Edelsbrunner, H. and Shi, W. 1991. An $O(n \log^2 h)$ time algorithm for the three-dimensional convex hull problem. *SIAM J. Comput.* 20(2):259–269.

Fortune, S. 1987. A sweepline algorithm for Voronoi diagrams. *Algorithmica* 2(2):153–174.

Fortune, S. 1993. Progress in computational geometry. In *Directions in Geom. Comput.*, pp. 81–128. R. Martin, ed. Information Geometers Ltd.

Ghosh, S. K. and Mount, D. M. 1991. An output-sensitive algorithm for computing visibility graphs. *SIAM J. Comput.* 20(5):888–910.

Guibas, L. J. and Hershberger, J. 1989. Optimal shortest path queries in a simple polygon. *J. Comput. Syst. Sci.* 39:126–152.

Guibas, L. J. and Seidel, R. 1987. Computing convolutions by reciprocal search. *Discrete Comput. Geom.* 2(2):175–193.

Handler, G. Y. and Mirchandani, P. B. 1979. *Location on Networks: Theory and Algorithm*. MIT Press, Cambridge, MA.

Hershberger, J. and Suri, S. 1993. Efficient computation of Euclidean shortest paths in the plane, pp. 508–517. In *Proc. 34th Ann. IEEE Symp. Found. Comput. Sci.*

Ho, J. M., Chang, C. H., Lee, D. T., and Wong, C. K. 1991. Minimum diameter spanning tree and related problems. *SIAM J. Comput.* 20(5):987–997.

Houle, M. E. and Toussaint, G. T. 1988. Computing the width of a set. *IEEE Trans. Pattern Anal. Machine Intelligence* PAMI-10(5):761–765.

Imai, H. and Asano, T. 1986. Efficient algorithms for geometric graph search problems. *SIAM J. Comput.* 15(2):478–494.

Imai, H. and Asano, T. 1987. Dynamic orthogonal segment intersection search. *J. Algorithms* 8(1): 1–18.

Janardan, R. and Lopez, M. 1993. Generalized intersection searching problems. *Int. J. Comput. Geom. Appl.* 3(1):39–69.

Kapoor, S. and Smid, M. 1996. New techniques for exact and approximate dynamic closest-point problems. *SIAM J. Comput.* 25(4):775–796.

Keil, J. M. 1985. Decomposing a polygon into simpler components. *SIAM J. Comput.* 14(4):799–817.

Kirkpatrick, D. G. and Seidel, R. 1986. The ultimate planar convex hull algorithm? *SIAM J. Comput.*, 15(1):287–299.

Klein, R. 1989. *Concrete and Abstract Voronoi Diagrams*. LNCS Vol. 400, Springer–Verlag.

Le, V. B. and Lee, D. T. 1991. Out-of-roundness problem revisited. *IEEE Trans. Pattern Anal. Machine Intelligence* 13(3):217–223.

Lee, D. T. 1978. *Proximity and Reachability in the Plan*. Ph.D. Thesis, *Tech. Rep.* R-831, Coordinated Science Lab., University of Illinois, Urbana.

Lee, D. T. and Drysdale, R. L., III 1981. Generalization of Voronoi diagrams in the plane. *SIAM J. Comput.* 10(1):73–87.

Lee, D. T. and Lin, A. K. 1986a. Computational complexity of art gallery problems. *IEEE Trans. Inf. Theory* 32(2):276–282.

Lee, D. T. and Lin, A. K. 1986b. Generalized Delaunay triangulation for planar graphs. *Discrete Comput. Geom.* 1(3):201–217.

Lee, D. T. and Preparata, F. P. 1982. An improved algorithm for the rectangle enclosure problem. *J. Algorithms* 3(3):218–224.

Lee, D. T. and Preparata, F. P. 1984. Computational geometry: a survey. *IEEE Trans. Comput.* C-33(12):1072–1101.

Lee, D. T. and Wu, V. B. 1993. Multiplicative weighted farthest neighbor Voronoi diagrams in the plane, pp. 154–168. In *Proc. Int. Workshop Discrete Math. and Algorithms*. Hong Kong, Dec.

Lee, D. T. and Wu, Y. F. 1986. Geometric complexity of some location problems. *Algorithmica* 1(2):193–211.

Lee, D. T., Yang, C. D., and Chen, T. H. 1991. Shortest rectilinear paths among weighted obstacles. *Int. J. Comput. Geom. Appl.* 1(2):109–124.

Lee, D. T., Yang, C. D., and Wong, C. K. 1996. Rectilinear paths among rectilinear obstacles. In *Perspectives in Discrete Applied Math.* K. Bogart, ed.

Lozano-Pérez, T. 1983. Spatial planning: a configuration space approach. *IEEE Trans. Comput.* C-32(2):108–120.

Mairson, H. G. and Stolfi, J. 1988. Reporting and counting intersections between two sets of line segments, pp. 307–325. In *Proc. Theor. Found. Comput. Graphics CAD*. Vol. F40, Springer–Verlag.

Matoušek, J. 1994. Geometric range searching. *ACM Computing Sur.* 26:421–461.

Matoušek, J., Sharir, M., and Welzl, E. 1992. A subexponential bound for linear programming, pp. 1–8. In *Proc. 8th Ann. ACM Symp. Comput. Geom.*

Megiddo, N. 1983a. Applying parallel computation algorithms in the design of serial algorithms. *J. ACM* 30(4):852–865.

Megiddo, N. 1983b. Linear time algorithm for linear programming in R^3 and related problems. *SIAM J. Comput.* 12(4):759–776.

Megiddo, N. 1983c. Towards a genuinely polynomial algorithm for linear programming. *SIAM J. Comput.* 12(2):347–353.

Megiddo, N. 1984. Linear programming in linear time when the dimension is fixed. *J. ACM* 31(1):114–127.

Megiddo, N. 1986. New approaches to linear programming. *Algorithmica* 1(4):387–394.

Mehlhorn, K. 1984. *Data Structures and Algorithms*, Vol. 3, Multi-dimensional searching and computational geometry. Springer–Verlag.

Mitchell, J. S. B. 1993. Shortest paths among obstacles in the plane, pp. 308–317. In *Proc. 9th ACM Symp. Comput. Geom.*, May.

Mitchell, J. S. B., Mount, D. M., and Papadimitriou, C. H. 1987. The discrete geodesic problem. *SIAM J. Comput.* 16(4):647–668.

Mitchell, J. S. B. and Papadimitriou, C. H. 1991. The weighted region problem: finding shortest paths through a weighted planar subdivision. *J. ACM* 38(1):18–73.

Mitchell, J. S. B., Rote, G., and Wōginger, G. 1992. Minimum link path among obstacles in the planes. *Algorithmica* 8(5/6):431–459.

Mulmuley, K. 1994. *Computational Geometry: An Introduction through Randomized Algorithms*. Prentice–Hall, Englewood Cliffs, NJ.

Nievergelt, J. and Preparata, F. P. 1982. Plane-sweep algorithms for intersecting geometric figures. *Commun. ACM* 25(10):739–747.

O'Rourke, J. 1987. *Art Gallery Theorems and Algorithms*. Oxford University Press, New York.

O'Rourke, J. 1994. *Computational Geometry in C*. Cambridge University Press, New York.

O'Rourke, J. and Supowit, K. J. 1983. Some NP-hard polygon decomposition problems. *IEEE Trans. Inform. Theory* IT-30(2):181–190,

Overmars, M. H. 1983. *The Design of Dynamic Data Structures*. LNCS Vol. 156, Springer–Verlag.

Preparata, F. P. and Shamos, M. I. 1985. In *Computational Geometry: An Introduction*. Springer–Verlag.

Ruppert, J. and Seidel, R. 1992. On the difficulty of triangulating three-dimensional non-convex polyhedra. *Discrete Comput. Geom.* 7(3):227–253.

Schaudt, B. F. and Drysdale, R. L. 1991. Multiplicatively weighted crystal growth Voronoi diagrams, pp. 214–223. In *Proc. 7th Ann. ACM Symp. Comput. Geom.*

Schwartz, C., Smid, M., and Snoeyink, J. 1994. An optimal algorithm for the on-line closest-pair problem. *Algorithmica* 12(1):18–29.

Seo, D. Y. and Lee, D. T. 1995. On the complexity of bicriteria spanning tree problems for a set of points in the plane. *Tech. Rep. Dept.* EE/CS, Northwestern University, June.

Sharir, M. 1985. Intersection and closest-pair problems for a set of planar discs. *SIAM J. Comput.* 14(2):448–468.

Sharir, M. 1987. On shortest paths amidst convex polyhedra. *SIAM J. Comput.* 16(3):561–572.

Shermer, T. C. 1992. Recent results in art galleries. *Proc. IEEE* 80(9):1384–1399.

Sifrony, S. and Sharir, M. 1987. A new efficient motion planning algorithm for a rod in two-dimensional polygonal space. *Algorithmica* 2(4):367–402.

Smid, M. 1992. Maintaining the minimal distance of a point set in polylogarithmic time. *Discrete Comput. Geom.* 7:415–431.

Suri, S. 1990. On some link distance problems in a simple polygon. *IEEE Trans. Robotics Automation* 6(1):108–113.

Swanson, K., Lee, D. T., and Wu, V. L. 1995. An optimal algorithm for roundness determination on convex polygons. *Comput. Geom. Theory Appl.* 5(4):225–235.

Toussaint, G. T., ed. 1985. *Computational Geometry*. North–Holland.

Van der Stappen, A. F. and Overmars, M. H. 1994. Motion planning amidst fat obstacle, pp. 31–40. In *Proc. 10th Ann. ACM Comput. Geom.*, June.

van Leeuwen, J. and Wood, D. 1980. Dynamization of decomposable searching problems. *Inf. Proc. Lett.* 10:51–56.

Willard, D. E. 1982. Polygon retrieval. *SIAM J. Comput.* 11(1):149–165.

Willard, D. E. 1985. New data structures for orthogonal range queries. *SIAM J. Comput.* 14(1):232–253.

Willard, D. E. and Luecker, G. S. 1985. Adding range restriction capability to dynamic data structures. *J. ACM* 32(3):597–617.

Yao, F. F. 1994. Computational geometry. In *Handbook of Theoretical Computer Science*, Vol. A: Algorithms and Complexity, J. van Leeuwen, ed., pp. 343–389.

Yap, C. K. 1987a. An $O(n \log n)$ algorithm for the Voronoi diagram of a set of simple curve segments. *Discrete Comput. Geom.* 2(4):365–393.

Yap, C. K. 1987b. Algorithmic motion planning. In *Advances in Robotics, Vol I: Algorithmic and Geometric Aspects of Robotics*. J. T. Schwartz and C. K. Yap, eds., pp. 95–143. Lawrence Erlbaum, London.

Further Information

We remark that there are new efforts being made in the applied side of algorithm development. A library of geometric software including visualization tools and applications programs is under development at the Geometry Center, University of Minnesota, and a concerted effort is being put together by researchers in Europe and in the United States to organize a system library containing primitive geometric abstract data types useful for geometric algorithm developers and practitioners.

Those who are interested in the implementations or would like to have more information about available software may consult the Proceedings of the Annual ACM Symposium on Computational Geometry, which has a video session, or the WWW page on *Geometry in Action* by David Eppstein (http://www.ics.uci.edu/~eppstein/geom.html).

12

Randomized Algorithms

Rajeev Motwani*
Stanford University

Prabhakar Raghavan
Verity, Inc.

12.1 Introduction

A **randomized algorithm** is one that makes random choices during its execution. The behavior of such an algorithm may thus be random even on a fixed input. The design and analysis of a randomized algorithm focus on establishing that it is likely to behave well on *every* input; the likelihood in such a statement depends only on the probabilistic choices made by the algorithm during execution and not on any assumptions about the input. It is especially important to distinguish a randomized algorithm from the *average-case analysis* of algorithms, where one analyzes an algorithm assuming that its input is drawn from a fixed probability distribution. With a randomized algorithm, in contrast, no assumption is made about the input.

Two benefits of randomized algorithms have made them popular: simplicity and efficiency. For many applications, a randomized algorithm is the simplest algorithm available, or the fastest, or both. In the following, we make these notions concrete through a number of illustrative examples. We assume that the reader has had undergraduate courses in algorithms and complexity, and in probability theory. A comprehensive source for randomized algorithms is the book by Motwani and Raghavan [1995]. The articles

*Supported by an Alfred P. Sloan Research Fellowship, an IBM Faculty Partnership Award, an ARO MURI Grant DAAH04-96-1-0007, and NSF Young Investigator Award CCR-9357849, with matching funds from IBM, Schlumberger Foundation, Shell Foundation, and Xerox Corporation.

by Karp [1991], Maffioli et al. [1985], and Welsh [1983] are good surveys of randomized algorithms. The book by Mulmuley [1993] focuses on randomized geometric algorithms.

Throughout this chapter, we assume the random access memory (RAM) model of computation, in which we have a machine that can perform the following operations involving registers and main memory: input–output operations, memory–register transfers, indirect addressing, branching, and arithmetic operations. Each register or memory location may hold an integer that can be accessed as a unit, but an algorithm has no access to the representation of the number. The arithmetic instructions permitted are $+$, $-$, $\times$, and $/$. In addition, an algorithm can compare two numbers and evaluate the square root of a positive number. In this chapter, $\mathbf{E}[X]$ will denote the expectation of random variable X, and $\mathbf{Pr}[A]$ will denote the probability of event A.

12.2 Sorting and Selection by Random Sampling

Some of the earliest randomized algorithms included algorithms for sorting the set S of numbers and the related problem of finding the kth smallest element in S. The main idea behind these algorithms is the use of *random sampling*: a randomly chosen member of S is unlikely to be one of its largest or smallest elements; rather, it is likely to be near the middle. Extending this intuition suggests that a random sample of elements from S is likely to be spread roughly uniformly in S. We now describe randomized algorithms for sorting and selection based on these ideas.

Algorithm RQS

Input: A set of numbers, S.
Output: The elements of S sorted in increasing order.

1. *Choose element y uniformly at random from S: every element in S has equal probability of being chosen.*
2. *By comparing each element of S with y, determine the set S_1 of elements smaller than y and the set S_2 of elements larger than y.*
3. *Recursively sort S_1 and S_2. Output the sorted version of S_1, followed by y, and then the sorted version of S_2.*

Algorithm RQS is an example of a *randomized algorithm* — an algorithm that makes random choices during execution. It is inspired by the Quicksort algorithm due to Hoare [1962], and described in Motwani and Raghavan [1995]. We assume that the random choice in Step 1 can be made in unit time. What can we prove about the running time of RQS?

We now analyze the *expected* number of comparisons in an execution of RQS. Comparisons are performed in Step 2, in which we compare a randomly chosen element to the remaining elements. For $1 \leq i \leq n$, let $S_{(i)}$ denote the element of *rank i* (the ith smallest element) in the set S. Define the random variable X_{ij} to assume the value 1 if $S_{(i)}$ and $S_{(j)}$ are compared in an execution and the value 0 otherwise. Thus, the total number of comparisons is $\sum_{i=1}^{n} \sum_{j>i} X_{ij}$. By linearity of expectation, the expected number of comparisons is

$$\mathrm{E}\left[\sum_{i=1}^{n} \sum_{j>i} X_{ij}\right] = \sum_{i=1}^{n} \sum_{j>i} \mathbf{E}[X_{ij}] \tag{12.1}$$

Let p_{ij} denote the probability that $S_{(i)}$ and $S_{(j)}$ are compared during an execution. Then,

$$\mathrm{E}[X_{ij}] = p_{ij} \times 1 + (1 - p_{ij}) \times 0 = p_{ij} \tag{12.2}$$

To compute p_{ij}, we view the execution of RQS as binary tree T, each node of which is labeled with a distinct element of S. The root of the tree is labeled with the element y chosen in Step 1; the left subtree of

y contains the elements in S_1 and the right subtree of y contains the elements in S_2. The structures of the two subtrees are determined recursively by the executions of RQS on S_1 and S_2. The root y is compared to the elements in the two subtrees, but no comparison is performed between an element of the left subtree and an element of the right subtree. Thus, there is a comparison between $S_{(i)}$ and $S_{(j)}$ if and only if one of these elements is an ancestor of the other.

Consider the permutation π obtained by visiting the nodes of T in increasing order of the level numbers and in a left-to-right order within each level; recall that the ith level of the tree is the set of all nodes at a distance exactly i from the root. The following two observations lead to the determination of p_{ij}:

1. There is a comparison between $S_{(i)}$ and $S_{(j)}$ if and only if $S_{(i)}$ or $S_{(j)}$ occurs earlier in the permutation π than any element $S_{(\ell)}$ such that $i < \ell < j$. To see this, let $S_{(k)}$ be the earliest in π from among all elements of rank between i and j. If $k \notin \{i, j\}$, then $S_{(i)}$ will belong to the left subtree of $S_{(k)}$ and $S_{(j)}$ will belong to the right subtree of $S_{(k)}$, implying that there is no comparison between $S_{(i)}$ and $S_{(j)}$. Conversely, when $k \in \{i, j\}$, there is an ancestor–descendant relationship between $S_{(i)}$ and $S_{(j)}$, implying that the two elements are compared by RQS.
2. Any of the elements $S_{(i)}, S_{(i+1)}, \ldots, S_{(j)}$ is equally likely to be the first of these elements to be chosen as a partitioning element and hence to appear first in π. Thus, the probability that this first element is either $S_{(i)}$ or $S_{(j)}$ is exactly $2/(j - i + 1)$.

It follows that $p_{ij} = 2/(j - i + 1)$. By Eqs. (12.1) and (12.2), the expected number of comparisons is given by:

$$\begin{aligned}\sum_{i=1}^{n}\sum_{j>i} p_{ij} &= \sum_{i=1}^{n}\sum_{j>i}\frac{2}{j-i+1}\\ &\le \sum_{i=1}^{n-1}\sum_{k=1}^{n-i}\frac{2}{k+1}\\ &\le 2\sum_{i=1}^{n}\sum_{k=1}^{n}\frac{1}{k}\end{aligned}$$

It follows that the expected number of comparisons is bounded above by $2nH_n$, where H_n is the nth *harmonic number*, defined by $H_n = \sum_{k=1}^{n} 1/k$.

Theorem 12.1 *The expected number of comparisons in an execution of RQS is at most* $2nH_n$.

Now $H_n = \ell n\, n + \Theta(1)$, so that the expected running time of RQS is $O(n \log n)$. Note that this expected running time *holds for every input*. It is an expectation that depends only on the random choices made by the algorithm and *not* on any assumptions about the distribution of the input.

12.2.1 Randomized Selection

We now consider the use of random sampling for the problem of selecting the kth smallest element in set S of n elements drawn from a totally ordered universe. We assume that the elements of S are all distinct, although it is not very hard to modify the following analysis to allow for multisets. Let $r_S(t)$ denote the rank of element t (the kth smallest element has rank k) and recall that $S_{(i)}$ denotes the ith smallest element of S. Thus, we seek to identify $S_{(k)}$. We extend the use of this notation to subsets of S as well. The following algorithm is adapted from one due to Floyd and Rivest [1975].

Algorithm LazySelect

Input: A set, S, of n elements from a totally ordered universe and an integer, k, in $[1, n]$.
Output: The kth smallest element of S, $S_{(k)}$.

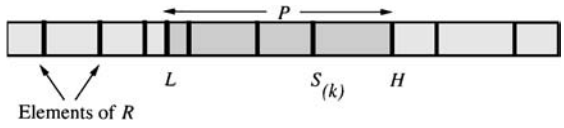


FIGURE 12.1 The LazySelect algorithm.

1. *Pick $n^{3/4}$ elements from S, chosen independently and uniformly at random with replacement; call this multiset of elements R.*
2. *Sort R in $O(n^{3/4} \log n)$ steps using any optimal sorting algorithm.*
3. *Let $x = kn^{-1/4}$. For $\ell = \max\{\lfloor x - \sqrt{n} \rfloor, 1\}$ and $h = \min\{\lceil x + \sqrt{n} \rceil, n^{3/4}\}$, let $a = R_{(\ell)}$ and $b = R_{(h)}$. By comparing a and b to every element of S, determine $r_S(a)$ and $r_S(b)$.*
4. if $k < n^{1/4}$, *let $P = \{y \in S \mid y \leq b\}$ and $r = k$;*
   else if $k > n - n^{1/4}$, *let $P = \{y \in S \mid y \geq a\}$ and $r = k - r_S(a) + 1$;*
   else if $k \in [n^{1/4}, n - n^{1/4}]$, *let $P = \{y \in S \mid a \leq y \leq b\}$ and $r = k - r_S(a) + 1$;*
   Check whether $S_{(k)} \in P$ and $|P| \leq 4n^{3/4} + 2$. If not, repeat Steps 1–3 until such a set, P, is found.
5. *By sorting P in $O(|P| \log |P|)$ steps, identify P_r, which is $S_{(k)}$.*

Figure 12.1 illustrates Step 3, where small elements are at the left end of the picture and large ones are to the right. Determining (in Step 4) whether $S_{(k)} \in P$ is easy because we know the ranks $r_S(a)$ and $r_S(b)$ and we compare either or both of these to k, depending on which of the three *if* statements in Step 4 we execute. The sorting in Step 5 can be performed in $O(n^{3/4} \log n)$ steps.

Thus, the idea of the algorithm is to identify two elements a and b in S such that both of the following statements hold with high probability:

1. The element $S_{(k)}$ that we seek is in P, the set of elements between a and b.
2. The set P of elements is not very large, so that we can sort P inexpensively in Step 5.

As in the analysis of RQS, we measure the running time of LazySelect in terms of the number of comparisons performed by it. The following theorem is established using the *Chebyshev bound* from elementary probability theory; a full proof can be found in Motwani and Raghavan [1995].

Theorem 12.2 *With probability $1 - O(n^{-1/4})$, LazySelect finds $S_{(k)}$ on the first pass through Steps 1–5 and thus performs only $2n + o(n)$ comparisons.*

This adds to the significance of LazySelect — the best-known deterministic selection algorithms use $3n$ comparisons in the worst case and are quite complicated to implement.

12.3 A Simple Min-Cut Algorithm

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

$$\Pr[\mathcal{E}_1 \cap \mathcal{E}_2] = \Pr[\mathcal{E}_1] \times \Pr[\mathcal{E}_2] \tag{12.3}$$

More generally, when $\mathcal{E}_1$ and $\mathcal{E}_2$ are not necessarily independent,

$$\Pr[\mathcal{E}_1 \cap \mathcal{E}_2] = \Pr[\mathcal{E}_1 \mid \mathcal{E}_2] \times \Pr[\mathcal{E}_2] = \Pr[\mathcal{E}_2 \mid \mathcal{E}_1] \times \Pr[\mathcal{E}_1] \tag{12.4}$$

where $\Pr[\mathcal{E}_1 \mid \mathcal{E}_2]$ denotes the *conditional probability* of $\mathcal{E}_1$ given $\mathcal{E}_2$. When a collection of events is not independent, the probability of their intersection is given by the following generalization of Eq. (12.4):

$$\Pr\left[\bigcap_{i=1}^{k} \mathcal{E}_i\right] = \Pr[\mathcal{E}_1] \times \Pr[\mathcal{E}_2 \mid \mathcal{E}_1] \times \Pr[\mathcal{E}_3 \mid \mathcal{E}_1 \cap \mathcal{E}_2] \cdots \Pr\left[\mathcal{E}_k \,\middle|\, \bigcap_{i=1}^{k-1} \mathcal{E}_i\right] \tag{12.5}$$

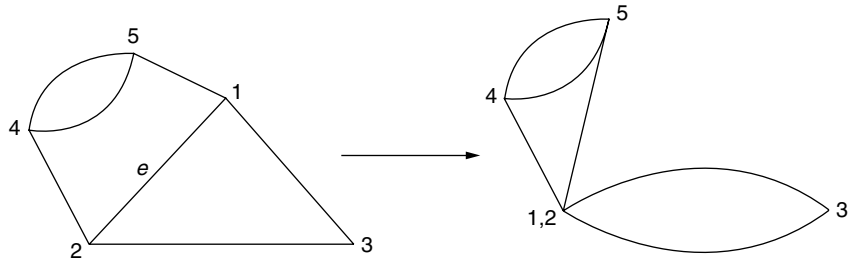


FIGURE 12.2 A step in the min-cut algorithm; the effect of contracting edge $e = (1, 2)$ is shown.

Let G be a connected, undirected multigraph with n vertices. A *multigraph* may contain multiple edges between any pair of vertices. A *cut* in G is a set of edges whose removal results in G being broken into two or more components. A *min-cut* is a cut of minimum cardinality. We now study a simple algorithm due to Karger [1993] for finding a min-cut of a graph.

We repeat the following step: Pick an edge uniformly at random and merge the two vertices at its end points. If as a result there are several edges between some pairs of (newly formed) vertices, retain them all. Remove edges between vertices that are merged, so that there are never any self-loops. This process of merging the two endpoints of an edge into a single vertex is called the *contraction* of that edge. See Figure 12.2. With each contraction, the number of vertices of G decreases by one. Note that as long as at least two vertices remain, an edge contraction does not reduce the min-cut size in G. The algorithm continues the contraction process until only two vertices remain; at this point, the set of edges between these two vertices is a cut in G and is output as a candidate min-cut. What is the probability that this algorithm finds a min-cut?

Definition 12.1 For any vertex v in the multigraph G, the *neighborhood* of G, denoted $\Gamma(v)$, is the set of vertices of G that are adjacent to v. The *degree* of v, denoted $d(v)$, is the number of edges incident on v. For the set S of vertices of G, the neighborhood of S, denoted $\Gamma(S)$, is the union of the neighborhoods of the constituent vertices.

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

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

For $1 \leq i \leq n-2$, let $\mathcal{E}_i$ denote the event of *not* picking an edge of C at the ith step. The probability that the edge randomly chosen in the first step is in C is at most $k/(nk/2) = 2/n$, so that $\Pr[\mathcal{E}_1] \geq 1-2/n$. Conditioned on the occurrence of $\mathcal{E}_1$, there are at least $k(n-1)/2$ edges during the second step so that $\Pr[\mathcal{E}_2 \mid \mathcal{E}_1] \geq 1 - 2/(n-1)$. Extending this calculation, $\Pr[\mathcal{E}_i \mid \cap_{j=1}^{i-1} \mathcal{E}_j] \geq 1 - 2/(n-i+1)$. We now invoke Eq. (12.5) to obtain

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

Our algorithm may err in declaring the cut it outputs to be a min-cut. But the probability of discovering a particular min-cut (which may in fact be the unique min-cut in G) is larger than $2/n^2$, so that the probability of error is at most $1 - 2/n^2$. Repeating the preceding algorithm $n^2/2$ times and making independent random choices each time, the probability that a min-cut is not found in any of the $n^2/2$

attempts is [by Eq. (12.3)], at most

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

By this process of repetition, we have managed to reduce the probability of failure from $1 - 2/n^2$ to less than $1/e$. Further executions of the algorithm will make the failure probability arbitrarily small (the only consideration being that repetitions increase the running time). Note the extreme simplicity of this randomized min-cut algorithm. In contrast, most deterministic algorithms for this problem are based on network flow and are considerably more complicated.

12.3.1 Classification of Randomized Algorithms

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

In contrast, the min-cut algorithm may sometimes produce a solution that is incorrect. However, we prove that the probability of such an error is bounded. Such an algorithm is called a **Monte Carlo algorithm**. We observe a useful property of a Monte Carlo algorithm: If the algorithm is run repeatedly with independent random choices each time, the failure probability can be made arbitrarily small, at the expense of running time. In some randomized algorithms, both the running time and the quality of the solution are random variables; sometimes these are also referred to as Monte Carlo algorithms. The reader is referred to Motwani and Raghavan [1995] for a detailed discussion of these issues.

12.4 Foiling an Adversary

A common paradigm in the design of randomized algorithms is that of *foiling an adversary*. Whereas an adversary might succeed in defeating a **deterministic algorithm** with a carefully constructed *bad* input, it is difficult for an adversary to defeat a randomized algorithm in this fashion. Due to the random choices made by the randomized algorithm, the adversary cannot, while constructing the input, predict the precise behavior of the algorithm. An alternative view of this process is to think of the randomized algorithm as first picking a series of random numbers, which it then uses in the course of execution as needed. In this view, we can think of the random numbers chosen at the start as *selecting* one of a family of deterministic algorithms. In other words, a randomized algorithm can be thought of as a probability distribution on deterministic algorithms. We illustrate these ideas in the setting of AND–OR *tree evaluation*; the following algorithm is due to Snir [1985].

For our purposes, an AND–OR tree is a rooted complete binary tree in which internal nodes at even distance from the root are labeled AND and internal nodes at odd distance are labeled OR. Associated with each leaf is a Boolean *value*. The *evaluation* of the game tree is the following process. Each leaf *returns* the value associated with it. Each OR node returns the Boolean OR of the values returned by its children, and each AND node returns the Boolean AND of the values returned by its children. At each step, an evaluation algorithm chooses a leaf and reads its value. We do not charge the algorithm for any other computation. We study the number of such steps taken by an algorithm for evaluating an AND–OR tree, the worst case being taken over all assignments of Boolean values of the leaves.

Let T_k denote an AND-OR tree in which every leaf is at distance $2k$ from the root. Thus, any root-to-leaf path passes through k AND nodes (including the root itself) and k OR nodes, and there are 2^{2k} leaves. An algorithm begins by specifying a leaf whose value is to be read at the first step. Thereafter, it specifies such a leaf at each step based on the values it has read on previous steps. In a deterministic algorithm, the choice of the next leaf to be read is a deterministic function of the values at the leaves read thus far. For a randomized algorithm, this choice may be randomized. It is not difficult to show that for any deterministic evaluation algorithm, there is an instance of T_k that forces the algorithm to read the values on all 2^{2k} leaves.

We now give a simple randomized algorithm and study the expected number of leaves it reads on any instance of T_k. The algorithm is motivated by the following simple observation. Consider a single AND node with two leaves. If the node were to return 0, at least one of the leaves must contain 0. A deterministic algorithm inspects the leaves in a fixed order, and an adversary can therefore always *hide* the 0 at the second of the two leaves inspected by the algorithm. Reading the leaves in a random order foils this strategy. With probability 1/2, the algorithm chooses the hidden 0 on the first step, so that its expected number of steps is 3/2, which is better than the worst case for any deterministic algorithm. Similarly, in the case of an OR node, if it were to return a 1, then a randomized order of examining the leaves will reduce the expected number of steps to 3/2. We now extend this intuition and specify the complete algorithm.

To evaluate an AND node, v, the algorithm chooses one of its children (a subtree rooted at an OR node) at random and evaluates it by recursively invoking the algorithm. If 1 is returned by the subtree, the algorithm proceeds to evaluate the other child (again by recursive application). If 0 is returned, the algorithm returns 0 for v. To evaluate an OR node, the procedure is the same with the roles of 0 and 1 interchanged. We establish by induction on k that the expected cost of evaluating any instance of T_k is at most 3^k.

The basis ($k = 0$) is trivial. Assume now that the expected cost of evaluating any instance of T_{k-1} is at most 3^{k-1}. Consider first tree T whose root is an OR node, each of whose children is the root of a copy of T_{k-1}. If the root of T were to evaluate to 1, at least one of its children returns 1. With probability 1/2, this child is chosen first, incurring (by the inductive hypothesis) an expected cost of at most 3^{k-1} in evaluating T. With probability 1/2 both subtrees are evaluated, incurring a net cost of at most $2 \times 3^{k-1}$. Thus, the expected cost of determining the value of T is

$$\leq \frac{1}{2} \times 3^{k-1} + \frac{1}{2} \times 2 \times 3^{k-1} = \frac{3}{2} \times 3^{k-1} \tag{12.6}$$

If, on the other hand, the OR were to evaluate to 0 both children must be evaluated, incurring a cost of at most $2 \times 3^{k-1}$.

Consider next the root of the tree T_k, an AND node. If it evaluates to 1, then both its subtrees rooted at OR nodes return 1. By the discussion in the previous paragraph and by linearity of expectation, the expected cost of evaluating T_k to 1 is at most $2 \times (3/2) \times 3^{k-1} = 3^k$. On the other hand, if the instance of T_k evaluates to 0, at least one of its subtrees rooted at OR nodes returns 0. With probability 1/2 it is chosen first, and so the expected cost of evaluating T_k is at most

$$2 \times 3^{k-1} + \frac{1}{2} \times \frac{3}{2} \times 3^{k-1} \leq 3^k$$

Theorem 12.3 *Given any instance of T_k, the expected number of steps for the preceding randomized algorithm is at most 3^k.*

Because $n = 4^k$, the expected running time of our randomized algorithm is $n^{\log_4 3}$, which we bound by $n^{0.793}$. Thus, the expected number of steps is smaller than the worst case for any deterministic algorithm. Note that this is a Las Vegas algorithm and always produces the correct answer.

12.5 The Minimax Principle and Lower Bounds

The randomized algorithm of the preceding section has an expected running time of $n^{0.793}$ on any uniform binary AND–OR tree with n leaves. Can we establish that *no randomized algorithm* can have a lower expected running time? We first introduce a standard technique due to Yao [1977] for proving such lower bounds. This technique applies only to algorithms that terminate in finite time on all inputs and sequences of random choices.

The crux of the technique is to relate the running times of randomized algorithms for a problem to the running times of deterministic algorithms for the problem *when faced with randomly chosen inputs*. Consider a problem where the number of distinct inputs of a fixed size is finite, as is the number of distinct

(deterministic, terminating, and always correct) algorithms for solving that problem. Let us define the **distributional complexity** of the problem at hand as the expected running time of the best deterministic algorithm for the worst distribution on the inputs. Thus, we envision an adversary choosing a probability distribution on the set of possible inputs and study the best deterministic algorithm for this distribution. Let $\boldsymbol{p}$ denote a probability distribution on the set $\mathcal{I}$ of inputs. Let the random variable $C(I_{\boldsymbol{p}}, A)$ denote the running time of deterministic algorithm $A \in \mathcal{A}$ on an input chosen according to $\boldsymbol{p}$. Viewing a randomized algorithm as a probability distribution $\boldsymbol{q}$ on the set $\mathcal{A}$ of deterministic algorithms, we let the random variable $C(I, A_{\boldsymbol{q}})$ denote the running time of this randomized algorithm on the worst-case input.

Proposition 12.1 (Yao's Minimax Principle) For all distributions $\boldsymbol{p}$ over $\mathcal{I}$ and $\boldsymbol{q}$ over $\mathcal{A}$,

$$\min_{A \in \mathcal{A}} \mathrm{E}[C(I_{\boldsymbol{p}}, A)] \leq \max_{I \in \mathcal{I}} \mathrm{E}[C(I, A_{\boldsymbol{q}})]$$

In other words, the expected running time of the optimal deterministic algorithm for an arbitrarily chosen input distribution $\boldsymbol{p}$ is a lower bound on the expected running time of the optimal (Las Vegas) randomized algorithm for Π. Thus, to prove a lower bound on the randomized complexity, it suffices to choose any distribution $\boldsymbol{p}$ on the input and prove a lower bound on the expected running time of deterministic algorithms for that distribution. The power of this technique lies in the flexibility in the choice of $\boldsymbol{p}$ and, more importantly, the reduction to a lower bound on deterministic algorithms. It is important to remember that the deterministic algorithm "knows" the chosen distribution $\boldsymbol{p}$.

The preceding discussion dealt only with lower bounds on the performance of Las Vegas algorithms. We briefly discuss Monte Carlo algorithms with error probability $\epsilon \in [0, 1/2]$. Let us define the distributional complexity with error ϵ, denoted $\min_{A \in \mathcal{A}} \mathrm{E}[C_{\epsilon}(I_{\boldsymbol{p}}, A)]$, to be the minimum expected running time of any deterministic algorithm that errs with probability at most ϵ under the input distribution $\boldsymbol{p}$. Similarly, we denote by $\max_{I \in \mathcal{I}} \mathrm{E}[C_{\epsilon}(I, A_{\boldsymbol{q}})]$ the expected running time (under the worst input) of any randomized algorithm that errs with probability at most ϵ (again, the randomized algorithm is viewed as probability distribution $\boldsymbol{q}$ on deterministic algorithms). Analogous to Proposition 12.1, we then have:

Proposition 12.2 For all distributions $\boldsymbol{p}$ over $\mathcal{I}$ and $\boldsymbol{q}$ over $\mathcal{A}$ and any $\epsilon \in [0, 1/2]$,

$$\frac{1}{2}\left(\min_{A \in \mathcal{A}} \mathrm{E}[C_{2\epsilon}(I_{\boldsymbol{p}}, A)] \right) \leq \max_{I \in \mathcal{I}} \mathrm{E}[C_{\epsilon}(I, A_{\boldsymbol{q}})]$$

12.5.1 Lower Bound for Game Tree Evaluation

We now apply Yao's minimax principle to the AND-OR tree evaluation problem. A randomized algorithm for AND-OR tree evaluation can be viewed as a probability distribution over deterministic algorithms, because the length of the computation as well as the number of choices at each step are both finite. We can imagine that all of these coins are tossed before the beginning of the execution.

The tree T_k is equivalent to a balanced binary tree, all of whose leaves are at distance $2k$ from the root and all of whose internal nodes compute the NOR function; a node returns the value 1 if both inputs are 0, and 0 otherwise. We proceed with the analysis of this tree of NORs of depth $2k$.

Let $p = (3 - \sqrt{5})/2$; each leaf of the tree is independently set to 1 with probability p. If each input to a NOR node is independently 1 with probability p, its output is 1 with probability

$$\left(\frac{\sqrt{5} - 1}{2} \right)^2 = \frac{3 - \sqrt{5}}{2} = p$$

Thus, the value of every node of NOR tree is 1 with probability p, and the value of a node is independent of the values of all of the other nodes on the same level. Consider a deterministic algorithm that is evaluating a tree furnished with such random inputs, and let v be a node of the tree whose value the algorithm is trying to determine. Intuitively, the algorithm should determine the value of one child of v before inspecting any leaf of the other subtree. An alternative view of this process is that the deterministic algorithm should inspect leaves visited in a depth-first search of the tree, except of course that it ceases to visit subtrees of node v when the value of v has been determined. Let us call such an algorithm a *depth-first pruning* algorithm, referring to the order of traversal and the fact that subtrees that supply no additional information are pruned away without being inspected. The following result is due to Tarsi [1983]:

Proposition 12.3 Let T be a NOR tree each of whose leaves is independently set to 1 with probability q for a fixed value $q \in [0,1]$. Let $W(T)$ denote the minimum, over all deterministic algorithms, of the expected number of steps to evaluate T. Then, there is a depth-first pruning algorithm whose expected number of steps to evaluate T is $W(T)$.

Proposition 12.3 tells us that for the purposes of our lower bound, we can restrict our attention to depth-first pruning algorithms. Let $W(h)$ be the expected number of leaves inspected by a depth-first pruning algorithm in determining the value of a node at distance h from the leaves, when each leaf is independently set to 1 with probability $(3-\sqrt{5})/2$. Clearly,

$$W(h) = W(h-1) + (1-p) \times W(h-1)$$

where the first term represents the work done in evaluating one of the subtrees of the node, and the second term represents the work done in evaluating the other subtree (which will be necessary if the first subtree returns the value 0, an event occurring with probability $1-p$). Letting h be $\log_2 n$ and solving, we get $W(h) \geq n^{0.694}$.

Theorem 12.4 *The expected running time of any randomized algorithm that always evaluates an instance of T_k correctly is at least $n^{0.694}$, where $n = 2^{2k}$ is the number of leaves.*

Why is our lower bound of $n^{0.694}$ less than the upper bound of $n^{0.793}$ that follows from Theorem 12.3? The reason is that we have not chosen the best possible probability distribution for the values of the leaves. Indeed, in the NOR tree if both inputs to a node are 1, no reasonable algorithm will read leaves of both subtrees of that node. Thus, to prove the best lower bound we have to choose a distribution on the inputs that precludes the event that both inputs to a node will be 1; in other words, the values of the inputs are chosen at random but not independently. This stronger (and considerably harder) analysis can in fact be used to show that the algorithm of section 12.4 is optimal; the reader is referred to the paper of Saks and Wigderson [1986] for details.

12.6 Randomized Data Structures

Recent research into data structures has strongly emphasized the use of randomized techniques to achieve increased efficiency without sacrificing simplicity of implementation. An illustrative example is the randomized data structure for dynamic dictionaries called *skip list* that is due to Pugh [1990].

The dynamic dictionary problem is that of maintaining the set of keys X drawn from a totally ordered universe so as to provide efficient support of the following operations: $\mathsf{find}(q, X)$ — decide whether the query key q belongs to X and return the information associated with this key if it does indeed belong to X; $\mathsf{insert}(q, X)$ — insert the key q into the set X, unless it is already present in X; $\mathsf{delete}(q, X)$ — delete the key q from X, unless it is absent from X. The standard approach for solving this problem involves the use of a binary search tree and gives worst-case time per operation that is $O(\log n)$, where n is the size of X at the time the operation is performed. Unfortunately, achieving this time bound requires the use of

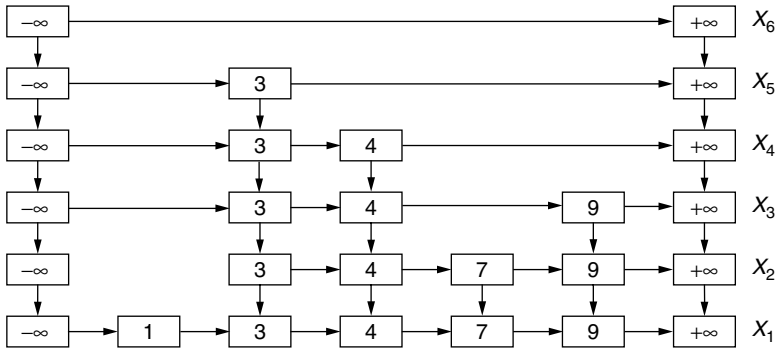


FIGURE 12.3 A skip list.

complex rebalancing strategies to ensure that the search tree remains balanced, that is, has depth $O(\log n)$. Not only does rebalancing require more effort in terms of implementation, but it also leads to significant overheads in the running time (at least in terms of the constant factors subsumed by the big-O notation). The skip list data structure is a rather pleasant alternative that overcomes both of these shortcomings.

Before getting into the details of randomized skip lists, we will develop some of the key ideas without the use of randomization. Suppose we have a totally ordered data set, $X = \{x_1 < x_2 < \cdots < x_n\}$. A *gradation of* X is a sequence of nested subsets (called *levels*)

$$X_r \subseteq X_{r-1} \subseteq \cdots \subseteq X_2 \subseteq X_1$$

such that $X_r = \emptyset$ and $X_1 = X$. Given an ordered set, X, and a gradation for it, the level of any element $x \in X$ is defined as

$$L(x) = \max\{i \mid x \in X_i\}$$

that is, $L(x)$ is the largest index i such that x belongs to the ith level of the gradation. In what follows, we will assume that two special elements $-\infty$ and $+\infty$ belong to each of the levels, where $-\infty$ is smaller than all elements in X and $+\infty$ is larger than all elements in X.

We now define an ordered list data structure with respect to a gradation of the set X. The first level, X_1, is represented as an ordered linked list, and each node x in this list has a stack of $L(x) - 1$ additional nodes directly above it. Finally, we obtain the skip list with respect to the gradation of X by introducing horizontal and vertical pointers between these nodes as illustrated in Figure 12.3. The skip list in Figure 12.3 corresponds to a gradation of the data set $X = \{1, 3, 4, 7, 9\}$ consisting of the following six levels:

$$\begin{aligned} X_6 &= \emptyset \\ X_5 &= \{3\} \\ X_4 &= \{3, 4\} \\ X_3 &= \{3, 4, 9\} \\ X_2 &= \{3, 4, 7, 9\} \\ X_1 &= \{1, 3, 4, 7, 9\} \end{aligned}$$

Observe that starting at the ith node from the bottom in the leftmost column of nodes and traversing the horizontal pointers in order yields a set of nodes corresponding to the elements of the ith level X_i.

Additionally, we will view each level i as defining a set of *intervals*, each of which is defined as the set of elements of X spanned by a horizontal pointer at level i. The sequence of levels X_i can be viewed as successively coarser partitions of X. In Figure 12.3, the levels determine the following

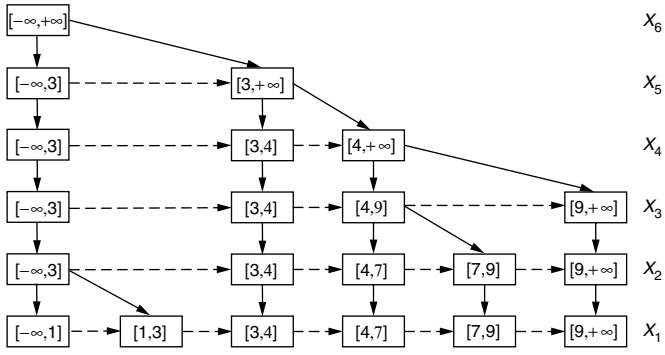


FIGURE 12.4 Tree representation of a skip list.

partitions of X into intervals:

$$X_6 = [-\infty, +\infty]$$
$$X_5 = [-\infty, 3] \cup [3, +\infty]$$
$$X_4 = [-\infty, 3] \cup [3, 4] \cup [4, +\infty]$$
$$X_3 = [-\infty, 3] \cup [3, 4] \cup [4, 9] \cup [9, +\infty]$$
$$X_2 = [-\infty, 3] \cup [3, 4] \cup [4, 7] \cup [7, 9] \cup [9, +\infty]$$
$$X_1 = [-\infty, 1] \cup [1, 3] \cup [3, 4] \cup [4, 7] \cup [7, 9] \cup [9, +\infty]$$

An alternative view of the skip list is in terms of a tree defined by the interval partition structure, as illustrated in Figure 12.4 for the preceding example. In this tree, each node corresponds to an interval, and the intervals at a given level are represented by nodes at the corresponding level of the tree. When the interval J at level $i+1$ is a superset of the interval I at level i, then the corresponding node J has the node I as a child in this tree. Let $C(I)$ denote the number of children in the tree of a node corresponding to the interval I; that is, it is the number of intervals from the previous level that are subintervals of I. Note that the tree is not necessarily binary because the value of $C(I)$ is arbitrary. We can view the skip list as a threaded version of this tree, where each thread is a sequence of (horizontal) pointers linking together the nodes at a level into an ordered list. In Figure 12.4, the broken lines indicate the threads, and the full lines are the actual tree pointers.

Finally, we need some notation concerning the membership of element x in the intervals already defined, where x is not necessarily a member of X. For each possible x, let $I_j(x)$ be the interval at level j containing x. In the degenerate case where x lies on the boundary between two intervals, we assign it to the leftmost such interval. Observe that the nested sequence of intervals containing y,

$$I_r(y) \subseteq I_{r-1}(y) \subseteq \cdots \subseteq I_1(y),$$

corresponds to a root-leaf path in the tree corresponding to the skip list.

It remains to specify the choice of the gradation that determines the structure of a skip list. This is precisely where we introduce randomization into the structure of a skip list. The idea is to define a random gradation. Our analysis will show that, with high probability, the search tree corresponding to a random skip list is balanced, and then the dictionary operations can be efficiently implemented.

We define the *random gradation* for X as follows. Given level X_i, the next level X_{i+1} is determined by independently choosing to retain each element $x \in X_i$ with probability 1/2. The random selection process begins with $X_1 = X$ and terminates when for the first time the resulting level is empty. Alternatively, we may view the choice of the gradation as follows. For each $x \in X$, choose the level $L(x)$ independently from the geometric distribution with parameter $p = 1/2$ and place x in the levels $X_1, \ldots, X_{L(x)}$. We define r to be one more than the maximum of these level numbers. Such a random level is chosen for every element of X upon its insertion and remains fixed unit until its deletion.

We omit the proof of the following theorem bounding the space complexity of a randomized skip list. The proof is a simple exercise, and it is recommended that the reader verify this to gain some insight into the behavior of this data structure.

Theorem 12.5 *A random skip list for a set, X, of size n has expected space requirement $O(n)$.*

We will go into more detail about the time complexity of this data structure. The following lemma underlies the running time analysis.

Lemma 12.1 *The number of levels r in a random gradation of a set, X, of size n has expected value $\mathrm{E}[r] = O(\log n)$. Further, $r = O(\log n)$ with high probability.*

***Proof* 12.1** We will prove the high probability result; the bound on the expected value follows immediately from this. Recall that the level numbers $L(x)$ for $x \in X$ are independent and identically distributed (i.i.d.) random variables distributed geometrically with parameter $p = 1/2$; notationally, we will denote these random variables by $Z_1, \ldots, Z_n$. Now, the total number of levels in the skip list can be determined as

$$r = 1 + \max_{x \in X} L(x) = 1 + \max_{1 \le i \le n} Z_i$$

that is, as one more than the maximum of n i.i.d. geometric random variables.

For such geometric random variables with parameter p, it is easy to verify that for any positive real t, $\Pr[Z_i > t] \le (1-p)^t$. It follows that

$$\Pr[\max_i Z_i > t] \le n(1-p)^t = \frac{n}{2^t}$$

because $p = 1/2$ in this case. For any $\alpha > 1$, setting $t = \alpha \log n$, we obtain

$$\Pr[r > \alpha \log n] \le \frac{1}{n^{\alpha - 1}}$$ □

We can now infer that the tree representing the skip list has height $O(\log n)$ with high probability. To show that the overall search time in a skip list is similarly bounded, we must first specify an efficient implementation of the find operation. We present the implementation of the dictionary operations in terms of the tree representation; it is fairly easy to translate this back into the skip list representation.

To implement find (y, X), we must walk down the path

$$I_r(y) \subseteq I_{r-1}(y) \subseteq \cdots \subseteq I_1(y)$$

For this, at level j, starting at the node $I_j(y)$, we use the vertical pointer to descend to the leftmost child of the current interval; then, via the horizontal pointers, we move rightward until the node $I_j(y)$ is reached. Note that it is easily determined whether y belongs to a given interval or to an interval to its right. Further, in the skip list, the vertical pointers allow access only to the leftmost child of an interval, and therefore we must use the horizontal pointers to scan its children.

To determine the expected cost of find(y, X) operation, we must take into account both the number of levels and the number of intervals/nodes scanned at each level. Clearly, at level j, the number of nodes visited is no more than the number of children of $I_{j+1}(y)$. It follows that the cost of find can be bounded by

$$O\left(\sum_{j=1}^{r} (1 + C(I_j(y)))\right)$$

The following lemma shows that this quantity has expectation bounded by $O(\log n)$.

Lemma 12.2 *For any y, let $I_r(y), \ldots, I_1(y)$ be the search path followed by* find(y, X) *in a random skip list for a set, X, of size n. Then,*

$$\mathrm{E}\left[\sum_{j=1}^{r}(1 + C(I_j(y)))\right] = O(\log n)$$

***Proof* 12.2** We begin by showing that for any interval I in a random skip list, $\mathrm{E}[C(I)] = O(1)$. By Lemma 12.1, we are guaranteed that $r = O(\log n)$ with his probability, and so we will obtain the desired bound. It is important to note that we really do need the high-probability bound on Lemma 12.1 because it is incorrect to multiply the expectation of r with that of $1 + C(I)$ (the two random variables need not be independent). However, in the approach we will use, because $r > \alpha \log n$ with probability at most $1/n^{\alpha-1}$ and $\sum_j (1 + C(I_j(y))) = O(n)$, it can be argued that the case $r > \alpha \log n$ does not contribute significantly to the expectation of $\sum_j C(I_j(y))$.

To show that the expected number of children of interval J at level i is bounded by a constant, we will show that the expected number of siblings of J (children of its parent) is bounded by a constant; in fact, we will bound only the number of right siblings because the argument for the number of left siblings is identical. Let the intervals to the right of J be the following:

$$J_1 = [x_1, x_2]; J_2 = [x_2, x_3]; \ldots; J_k = [x_k, +\infty]$$

Because these intervals exist at level i, each of the elements $x_1, \ldots, x_k$ belongs to X_i. If J has s right siblings, then it must be the case that $x_1, \ldots, x_s \notin X_{i+1}$, and $x_{s+1} \in X_{i+1}$. The latter event occurs with probability $1/2^{s+1}$ because each element of X_i is independently chosen to be in X_{i+1} with probability 1/2. Clearly, the number of right siblings of J can be viewed as a random variable that is geometrically distributed with parameter 1/2. It follows that the expected number of right siblings of J is at most 2. □

Consider now the implementation of the insert and delete operations. In implementing the operation insert(y, X), we assume that a random level, $L(y)$, is chosen for y as described earlier. If $L(y) > r$, then we start by creating new levels from $r + 1$ to $L(y)$ and then redefine r to be $L(y)$. This requires $O(1)$ time per level because the new levels are all empty prior to the insertion of y. Next we perform find(y, X) and determine the search path $I_r(y), \ldots, I_1(y)$, where r is updated to its new value if necessary. Given this search path, the insertion can be accomplished in time $O(L(y))$ by splitting around y the intervals $I_1(y), \ldots, I_{L(y)}(y)$ and updating the pointers as appropriate. The delete operation is the converse of the insert operation; it involves performing find(y, X) followed by collapsing the intervals that have y as an endpoint. Both operations incur costs that are the cost of a find operation and additional cost proportional to $L(y)$. By Lemmas 12.1 and 12.2, we obtain the following theorem.

Theorem 12.6 *In a random skip list for a set, X, of size n, the operations* find, insert, *and* delete *can be performed in expected time $O(\log n)$.*

12.7 Random Reordering and Linear Programming

The *linear programming problem* is a particularly notable example of the two main benefits of randomization: simplicity and speed. We now describe a simple algorithm for linear programming based on a paradigm for randomized algorithms known as *random reordering*. For many problems, it is possible to design natural algorithms based on the following idea. Suppose that the input consists of n elements. Given any subset of these n elements, there is a solution to the partial problem defined by these elements. If we start with the empty set and add the n elements of the input one at a time, maintaining a partial solution after each addition, we will obtain a solution to the entire problem when all of the elements have been added. The usual difficulty with this approach is that the running time of the algorithm depends

heavily on the order in which the input elements are added; for any fixed ordering, it is generally possible to force this algorithm to behave badly. The key idea behind random reordering is to *add the elements in a random order*. This simple device often avoids the pathological behavior that results from using a fixed order.

The linear programming problem is to find the extremum of a linear objective function of d real variables subject to set H of n constraints that are linear functions of these variables. The intersection of the n half-spaces defined by the constraints is a polyhedron in d-dimensional space (which may be empty, or possibly unbounded). We refer to this polyhedron as the *feasible region*. Without loss of generality [Schrijver 1986] we assume that the feasible region is nonempty and bounded. (Note that we are not assuming that we can *test* an arbitrary polyhedron for nonemptiness or boundedness; this is known to be equivalent to solving a linear program.) For a set of constraints, S, let $\mathcal{B}(S)$ denote the optimum of the linear program defined by S; we seek $\mathcal{B}(S)$.

Consider the following algorithm due to Seidel [1991]: Add the n constraints in random order, one at a time. After adding each constraint, determine the optimum subject to the constraints added so far. This algorithm also may be viewed in the following backwards manner, which will prove useful in the sequel.

Algorithm SLP

Input: A set of constraints H, and the dimension d.
Output: The optimum $\mathcal{B}(H)$.

0. *If there are only d constraints, output $\mathcal{B}(H) = H$.*
1. *Pick a random constraint $h \in H$;*
   Recursively find $\mathcal{B}(H\backslash\{h\})$.

2.1. If *$\mathcal{B}(H\backslash\{h\})$ does not violate h, output $\mathcal{B}(H\backslash\{h\})$ to be the optimum $\mathcal{B}(H)$.*

2.2. Else *project all of the constraints of $H\backslash\{h\})$ onto h and recursively solve this new linear programming problem of one lower dimension.*

The idea of the algorithm is simple. Either h (the constraint chosen randomly in Step 1) is redundant (in which case we execute Step 2.1), or it is not. In the latter case, we know that the vertex formed by $\mathcal{B}(H)$ must lie on the hyperplane bounding h. In this case, we project all of the constraints of $H\backslash\{h\}$ onto h and solve this new linear programming problem (which has dimension $d-1$).

The optimum $\mathcal{B}(H)$ is defined by d constraints. At the top level of recursion, the probability that random constraint h violates $\mathcal{B}(H\backslash\{h\})$ is at most d/n. Let $T(n,d)$ denote an upper bound on the expected running time of the algorithm for any problem with n constraints in d dimensions. Then, we may write

$$T(n,d) \leq T(n-1,d) + O(d) + \frac{d}{n}[O(dn) + T(n-1,d-1)] \tag{12.7}$$

In Equation (12.7), the first term on the right denotes the cost of recursively solving the linear program defined by the constraints in $H\backslash\{h\}$. The second accounts for the cost of checking whether h violates $\mathcal{B}(H\backslash\{h\})$. With probability d/n it does, and this is captured by the bracketed expression, whose first term counts the cost of projecting all of the constraints onto h. The second counts the cost of (recursively) solving the projected problem, which has one fewer constraint and dimension. The following theorem may be verified by substitution and proved by induction.

Theorem 12.7 *There is a constant b such that the recurrence (12.7) satisfies the solution $T(n,d) \leq bnd!$.*

In contrast, if the choice in Step 1 of SLP were not random, the recurrence (12.7) would be

$$T(n,d) \leq T(n-1,d) + O(d) + O(dn) + T(n-1,d-1) \tag{12.8}$$

whose solution contains a term that grows quadratically in n.

12.8 Algebraic Methods and Randomized Fingerprints

Some of the most notable randomized results in theoretical computer science, particularly in complexity theory, have involved a nontrivial combination of randomization and algebraic methods. In this section, we describe a fundamental randomization technique based on algebraic ideas. This is the randomized fingerprinting technique, originally due to Freivalds [1977], for the verification of identities involving matrices, polynomials, and integers. We also describe how this generalizes to the so-called Schwartz–Zippel technique for identities involving multivariate polynomials (independently due to Schwartz [1987] and Zippel [1979]; see also DeMillo and Lipton [1978]. Finally, following Lovász [1979], we apply the technique to the problem of detecting the existence of perfect matchings in graphs.

The *fingerprinting* technique has the following general form. Suppose we wish to decide the equality of two elements x and y drawn from some large universe U. Assuming any reasonable model of computation, this problem has a deterministic complexity $\Omega(\log|U|)$. Allowing randomization, an alternative approach is to choose a random function from U into a smaller space V such that with high probability x and y are identical if and only if their images in V are identical. These images of x and y are said to be their *fingerprints*, and the equality of fingerprints can be verified in time $O(\log|V|)$. Of course, for any fingerprint function the average number of elements of U mapped to an element of V is $|U|/|V|$; thus, it would appear impossible to find good fingerprint functions that work for arbitrary or worst-case choices of x and y. However, as we will show subsequently, when the identity checking is required to be correct only for x and y chosen from the small subspace S of U, particularly a subspace with some algebraic structure, it is possible to choose good fingerprint functions without any a priori knowledge of the subspace, provided the size of V is chosen to be comparable to the size of S.

Throughout this section, we will be working over some unspecified field $\mathcal{F}$. Because the randomization will involve uniform sampling from a finite subset of the field, we do not even need to specify whether the field is finite. The reader may find it helpful in the infinite case to assume that $\mathcal{F}$ is the field $\mathcal{Q}$ of rational numbers and in the finite case to assume that $\mathcal{F}$ is $\mathcal{Z}_p$, the field of integers modulo some prime number p.

12.8.1 Freivalds' Technique and Matrix Product Verification

We begin by describing a fingerprinting technique for verifying matrix product identities. Currently, the fastest algorithm for matrix multiplication (due to Coppersmith and Winograd [1990]) has running time $O(n^{2.376})$, improving significantly on the obvious $O(n^3)$ time algorithm; however, the fast matrix multiplication algorithm has the disadvantage of being extremely complicated. Suppose we have an implementation of the fast matrix multiplication algorithm and, given its complex nature, are unsure of its correctness. Because program verification appears to be an intractable problem, we consider the more reasonable goal of verifying the correctness of the output produced by executing the algorithm on specific inputs. (This notion of verifying programs on specific inputs is the basic tenet of the theory of *program checking* recently formulated by Blum and Kannan [1989].) More concretely, suppose we are given three $n \times n$ matrices $\boldsymbol{X}, \boldsymbol{Y}$, and $\boldsymbol{Z}$ over field $\mathcal{F}$, and would like to verify that $\boldsymbol{XY} = \boldsymbol{Z}$. Clearly, it does not make sense to use a simpler but slower matrix multiplication algorithm for the verification, as that would defeat the whole purpose of using the fast algorithm in the first place. Observe that, in fact, there is no need to recompute $\boldsymbol{Z}$; rather, we are merely required to verify that the product of $\boldsymbol{X}$ and $\boldsymbol{Y}$ is indeed equal to $\boldsymbol{Z}$. Freivalds' technique gives an elegant solution that leads to an $O(n^2)$ time randomized algorithm with bounded error probability.

The idea is to first pick the random vector $\boldsymbol{r} \in \{0,1\}^n$, that is, each component of $\boldsymbol{r}$ is chosen independently and uniformly at random from the set $\{0,1\}$ consisting of the additive and multiplicative identities of the field $\mathcal{F}$. Then, in $O(n^2)$ time, we can compute $\boldsymbol{y} = \boldsymbol{Yr}, \boldsymbol{x} = \boldsymbol{Xy} = \boldsymbol{XYr}$, and $\boldsymbol{z} = \boldsymbol{Zr}$. We would like to claim that the identity $\boldsymbol{XY} = \boldsymbol{Z}$ can be verified merely by checking that $\boldsymbol{x} = \boldsymbol{z}$. Quite clearly, if $\boldsymbol{XY} = \boldsymbol{Z}$, then $\boldsymbol{x} = \boldsymbol{z}$; unfortunately, the converse is not true in general. However, given the random choice of $\boldsymbol{r}$, we can show that for $\boldsymbol{XY} \neq \boldsymbol{Z}$, the probability that $\boldsymbol{x} \neq \boldsymbol{z}$ is at least 1/2. Observe that the fingerprinting algorithm errs only if $\boldsymbol{XY} \neq \boldsymbol{Z}$ but $\boldsymbol{x}$ and $\boldsymbol{z}$ turn out to be equal, and this has a bounded probability.

Theorem 12.8 *Let $\boldsymbol{X}, \boldsymbol{Y}$, and $\boldsymbol{Z}$ be $n \times n$ matrices over some field $\mathcal{F}$ such that $\boldsymbol{XY} \neq \boldsymbol{Z}$; further, let $\boldsymbol{r}$ be chosen uniformly at random from $\{0,1\}^n$ and define $\boldsymbol{x} = \boldsymbol{XYr}$ and $\boldsymbol{z} = \boldsymbol{Zr}$. Then,*

$$\Pr[\boldsymbol{x} = \boldsymbol{z}] \leq 1/2$$

***Proof* 12.3** Define $\boldsymbol{W} = \boldsymbol{XY} - \boldsymbol{Z}$ and observe that $\boldsymbol{W}$ is not the all-zeroes matrix. Because $\boldsymbol{Wr} = \boldsymbol{XYr} - \boldsymbol{Zr} = \boldsymbol{x} - \boldsymbol{z}$, the event $\boldsymbol{x} = \boldsymbol{z}$ is equivalent to the event that $\boldsymbol{Wr} = 0$. Assume, without loss of generality, that the first row of $\boldsymbol{W}$ has a nonzero entry and that the nonzero entries in that row precede all of the zero entries. Define the vector $\boldsymbol{w}$ as the first row of $\boldsymbol{W}$, and assume that the first $k > 0$ entries in $\boldsymbol{w}$ are nonzero. Because the first component of $\boldsymbol{Wr}$ is $\boldsymbol{w}^T\boldsymbol{r}$, giving an upper bound on the probability that the inner product of $\boldsymbol{w}$ and $\boldsymbol{r}$ is zero will give an upper bound on the probability that $\boldsymbol{x} = \boldsymbol{z}$.

Observe that $\boldsymbol{w}^T\boldsymbol{r} = 0$ if and only if

$$r_1 = \frac{-\sum_{i=2}^{k} w_i r_i}{w_1} \tag{12.9}$$

Suppose that while choosing the random vector $\boldsymbol{r}$, we choose $r_2, \ldots, r_n$ before choosing r_1. After the values for $r_2, \ldots, r_n$ have been chosen, the right-hand side of Equation (12.9) is fixed at some value $v \in \mathcal{F}$. If $v \notin \{0,1\}$, then r_1 will never equal v; conversely, if $v \in \{0,1\}$, then the probability that $r_1 = v$ is 1/2. Thus, the probability that $\boldsymbol{w}^T\boldsymbol{r} = 0$ is at most 1/2, implying the desired result. □

We have reduced the matrix multiplication verification problem to that of verifying the equality of two vectors. The reduction itself can be performed in $O(n^2)$ time and the vector equality can be checked in $O(n)$ time, giving an overall running time of $O(n^2)$ for this Monte Carlo procedure. The error probability can be reduced to $1/2^k$ via k independent iterations of the Monte Carlo algorithm. Note that there was nothing magical about choosing the components of the random vector $\boldsymbol{r}$ from $\{0,1\}$, because any two distinct elements of $\mathcal{F}$ would have done equally well. This suggests an alternative approach toward reducing the error probability, as follows: Each component of $\boldsymbol{r}$ is chosen independently and uniformly at random from some subset $\mathcal{S}$ of the field $\mathcal{F}$; then, it is easily verified that the error probability is no more than $1/|\mathcal{S}|$.

Finally, note that Freivalds' technique can be applied to the verification of any matrix identity $\boldsymbol{A} = \boldsymbol{B}$. Of course, given $\boldsymbol{A}$ and $\boldsymbol{B}$, just comparing their entries takes only $O(n^2)$ time. But there are many situations where, just as in the case of matrix product verification, computing $\boldsymbol{A}$ explicitly is either too expensive or possibly even impossible, whereas computing $\boldsymbol{Ar}$ is easy. The random fingerprint technique is an elegant solution in such settings.

12.8.2 Extension to Identities of Polynomials

The fingerprinting technique due to Freivalds is fairly general and can be applied to many different versions of the identity verification problem. We now show that it can be easily extended to identity verification for symbolic polynomials, where two polynomials $P_1(x)$ and $P_2(x)$ are deemed identical if they have identical coefficients for corresponding powers of x. Verifying integer or string equality is a special case because we can represent any string of length n as a polynomial of degree n by using the kth element in the string to determine the coefficient of the kth power of a symbolic variable.

Consider first the polynomial product verification problem: Given three polynomials $P_1(x)$, $P_2(x)$, $P_3(x) \in \mathcal{F}[x]$, we are required to verify that $P_1(x) \times P_2(x) = P_3(x)$. We will assume that $P_1(x)$ and $P_2(x)$ are of degree at most n, implying that $P_3(x)$ has degree at most $2n$. Note that degree n polynomials can be multiplied in $O(n \log n)$ time via fast Fourier transforms and that the evaluation of a polynomial can be done in $O(n)$ time.

The randomized algorithm we present for polynomial product verification is similar to the algorithm for matrix product verification. It first fixes set $\mathcal{S} \subseteq \mathcal{F}$ of size at least $2n + 1$ and chooses $r \in \mathcal{S}$ uniformly at random. Then, after evaluating $P_1(r)$, $P_2(r)$, and $P_3(r)$ in $O(n)$ time, the algorithm declares the identity $P_1(x)P_2(x) = P_3(x)$ to be correct if and only if $P_1(r)P_2(r) = P_3(r)$. The algorithm makes an error only

in the case where the polynomial identity is false but the value of the three polynomials at r indicates otherwise. We will show that the error event has a bounded probability.

Consider the degree $2n$ polynomial $Q(x) = P_1(x)P_2(x) - P_3(x)$. The polynomial $Q(x)$ is said to be *identically zero*, denoted by $Q(x) \equiv 0$, if each of its coefficients equals zero. Clearly, the polynomial identity $P_1(x)P_2(x) = P_3(x)$ holds if and only if $Q(x) \equiv 0$. We need to establish that if $Q(x) \not\equiv 0$, then with high probability $Q(r) = P_1(r)P_2(r) - P_3(r) \neq 0$. By elementary algebra we know that $Q(x)$ has at most $2n$ distinct roots. It follows that unless $Q(x) \equiv 0$, not more that $2n$ different choices of $r \in \mathcal{S}$ will cause $Q(r)$ to evaluate to 0. Therefore, the error probability is at most $2n/|\mathcal{S}|$. The probability of error can be reduced either by using independent iterations of this algorithm or by choosing a larger set $\mathcal{S}$. Of course, when $\mathcal{F}$ is an infinite field (e.g., the reals), the error probability can be made 0 by choosing r uniformly from the entire field $\mathcal{F}$; however, that requires an infinite number of random bits!

Note that we could also use a deterministic version of this algorithm where each choice of $r \in \mathcal{S}$ is tried once. But this involves $2n + 1$ different evaluations of each polynomial, and the best known algorithm for multiple evaluations needs $\Theta(n \log^2 n)$ time, which is more than the $O(n \log n)$ time requirement for actually performing a multiplication of the polynomials $P_1(x)$ and $P_2(x)$.

This verification technique is easily extended to a generic procedure for testing any polynomial identity of the form $P_1(x) = P_2(x)$ by converting it into the identity $Q(x) = P_1(x) - P_2(x) \equiv 0$. Of course, when P_1 and P_2 are explicitly provided, the identity can be deterministically verified in $O(n)$ time by comparing corresponding coefficients. Our randomized technique will take just as long to merely evaluate $P_1(x)$ and $P_2(x)$ at a random value. However, as in the case of verifying matrix identities, the randomized algorithm is quite useful in situations where the polynomials are implicitly specified, for example, when we have only a *black box* for computing the polynomials with no information about their coefficients, or when they are provided in a form where computing the actual coefficients is expensive. An example of the latter situation is provided by the following problem concerning the determinant of a symbolic matrix. In fact, the determinant problem will require a technique for the verification of polynomial identities of *multivariate* polynomials that we will discuss shortly.

Consider the $n \times n$ matrix $\boldsymbol{M}$. Recall that the determinant of the matrix M is defined as follows:

$$\det(\boldsymbol{M}) = \sum_{\pi \in \mathcal{S}_n} \operatorname{sgn}(\pi) \prod_{i=1}^{n} M_{i,\pi(i)} \tag{12.10}$$

where $\mathcal{S}_n$ is the symmetric group of permutations of order n, and $\operatorname{sgn}(\pi)$ is the sign of a permutation π. [The sign function is defined to be $\operatorname{sgn}(\pi) = (-1)^t$, where t is the number of pairwise exchanges required to convert the identity permutation into π.] Although the determinant is defined as a summation with $n!$ terms, it is easily evaluated in polynomial time provided that the matrix entries M_{ij} are explicitly specified. Consider the Vandermonde matrix $\boldsymbol{M}(x_1, \ldots, x_n)$, which is defined in terms of the indeterminates $x_1, \ldots, x_n$ such that $M_{ij} = x_i^{j-1}$, that is,

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

It is known that for the Vandermonde matrix, $\det(\boldsymbol{M}) = \prod_{i<j}(x_i - x_j)$. Consider the problem of verifying this identity without actually devising a formal proof. Computing the determinant of a symbolic matrix is infeasible as it requires dealing with a summation over $n!$ terms. However, we can formulate the identity verification problem as the problem of verifying that the polynomial $Q(x_1, \ldots, x_n) = \det(\boldsymbol{M}) - \prod_{i<j}(x_i - x_j)$ is identically zero. Based on our discussion of Freivalds' technique, it is natural to consider the substitution of random values for each x_i. Because the determinant can be computed in polynomial time for any

specific assignment of values to the symbolic variables $x_1, \ldots, x_n$, it is easy to evaluate the polynomial Q for random values of the variables. The only issue is that of bounding the error probability for this randomized test.

We now extend the analysis of Freivalds' technique for univariate polynomials to the multivariate case. But first, note that in a multivariate polynomial $Q(x_1, \ldots, x_n)$, the degree of a term is the sum of the exponents of the variable powers that define it, and the total degree of Q is the maximum over all terms of the degrees of the terms.

Theorem 12.9 *Let $Q(x_1, \ldots, x_n) \in \mathcal{F}[x_1, \ldots, x_n]$ be a multivariate polynomial of total degree m. Let $\mathcal{S}$ be a finite subset of the field $\mathcal{F}$, and let $r_1, \ldots, r_n$ be chosen uniformly and independently from $\mathcal{S}$. Then*

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

***Proof* 12.4** We will proceed by induction on the number of variables n. The basis of the induction is the case $n = 1$, which reduces to verifying the theorem for a univariate polynomial $Q(x_1)$ of degree m. But we have already seen for $Q(x_1) \not\equiv 0$ the probability that $Q(r_1) = 0$ is at most $m/|\mathcal{S}|$, taking care of the basis.

We now assume that the induction hypothesis holds for multivariate polynomials with at most $n - 1$ variables, where $n > 1$. In the polynomial $Q(x_1, \ldots, x_n)$ we can factor out the variable x_1 and thereby express Q as

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

where $k \leq m$ is the largest exponent of x_1 in Q. Given our choice of k, the coefficient $P_k(x_2, \ldots, x_n)$ of x_1^k cannot be identically zero. Note that the total degree of P_k is at most $m - k$. Thus, by the induction hypothesis, we conclude that the probability that $P_k(r_2, \ldots, r_n) = 0$ is at most $(m - k)/|\mathcal{S}|$.

Consider now the case where $P_k(r_2, \ldots, r_n)$ is indeed not equal to 0. We define the following univariate polynomial over x_1 by substituting the random values for the other variables in Q:

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

Quite clearly, the resulting polynomial $q(x_1)$ has degree k and is not identically zero (because the coefficient of x_i^k is assumed to be nonzero). As in the basis case, we conclude that the probability that $q(r_1) = Q(r_1, r_2, \ldots, r_n)$ evaluates to 0 is bounded by $k/|\mathcal{S}|$.

By the preceding arguments, we have established the following two inequalities:

$$\Pr[\, P_k(r_2, \ldots, r_n) = 0] \leq \frac{m - k}{|\mathcal{S}|}$$

$$\Pr[\, Q(r_1, r_2, \ldots, r_n) = 0 \mid P_k(r_2, \ldots, r_n) \neq 0] \leq \frac{k}{|\mathcal{S}|}$$

Using the elementary observation that for any two events $\mathcal{E}_1$ and $\mathcal{E}_2$, $\Pr[\mathcal{E}_1] \leq \Pr[\mathcal{E}_1 \mid \bar{\mathcal{E}}_2] + \Pr[\mathcal{E}_2]$, we obtain that the probability that $Q(r_1, r_2, \ldots, r_n) = 0$ is no more than the sum of the two probabilities on the right-hand side of the two obtained inequalities, which is $m/|S|$. This implies the desired results. □

This randomized verification procedure has one serious drawback: when working over large (or possibly infinite) fields, the evaluation of the polynomials could involve large intermediate values, leading to inefficient implementation. One approach to dealing with this problem in the case of integers is to perform all computations modulo some small random prime number; it can be shown that this does not have any adverse effect on the error probability.

12.8.3 Detecting Perfect Matchings in Graphs

We close by giving a surprising application of the techniques from the preceding section. Let $G(U, V, E)$ be a bipartite graph with two independent sets of vertices $U = \{u_1, \ldots, u_n\}$ and $V = \{v_1, \ldots, v_n\}$ and edges E that have one endpoint in each of U and V. We define a matching in G as a collection of edges $M \subseteq E$ such that each vertex is an endpoint of at most one edge in M; further, a perfect matching is defined to be a matching of size n, that is, where each vertex occurs as an endpoint of exactly one edge in M. Any perfect matching M may be put into a one-to-one correspondence with the permutations in $\mathcal{S}_n$, where the matching corresponding to a permutation $\pi \in \mathcal{S}_n$ is given by the collection of edges $\{(u_i, v_{\pi(i)} \mid 1 \leq i \leq n\}$. We now relate the matchings of the graph to the determinant of a matrix obtained from the graph.

Theorem 12.10 *For any bipartite graph $G(U, V, E)$, define a corresponding $n \times n$ matrix $\boldsymbol{A}$ as follows:*

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

Let the multivariate polynomial $Q(x_{11}, x_{12}, \ldots, x_{nn})$ denote the determinant $\det(\boldsymbol{A})$. *Then G has a perfect matching if and only if $Q \not\equiv 0$.*

***Proof* 12.5** We can express the determinant of $\boldsymbol{A}$ as follows:

$$\det(\boldsymbol{A}) = \sum_{\pi \in S_n} \operatorname{sgn}(\pi) A_{1,\pi(1)} A_{2,\pi(2)} \ldots A_{n,\pi(n)}$$

Note that there cannot be any cancellation of the terms in the summation because each indeterminate x_{ij} occurs at most once in $\boldsymbol{A}$. Thus, the determinant is not identically zero if and only if there exists some permutation π for which the corresponding term in the summation is nonzero. Clearly, the term corresponding to a permutation π is nonzero if and only if $A_{i,\pi(i)} \neq 0$ for each $i, 1 \leq i \leq n$; this is equivalent to the presence in G of the perfect matching corresponding to π. □

The matrix of indeterminates is sometimes referred to as the *Edmonds matrix* of a bipartite graph. The preceding result can be extended to the case of nonbipartite graphs, and the corresponding matrix of indeterminates is called the Tutte matrix. Tutte [1947] first pointed out the close connection between matchings in graphs and matrix determinants; the simpler relation between bipartite matchings and matrix determinants was given by Edmonds [1967].

We can turn the preceding result into a simple randomized procedure for testing the existence of perfect matchings in a bipartite graph (due to Lovász [1979]) — using the algorithm from the preceding subsection, determine whether the determinant is identically zero. The running time of this procedure is dominated by the cost of computing a determinant, which is essentially the same as the time required to multiply two matrices. Of course, there are algorithms for *constructing* a maximum matching in a graph with m edges and n vertices in time $O(m\sqrt{n})$ (see Hopcroft and Karp [1973], Micali and Vazirani [1980], Vazirani [1994], and Feder and Motwani [1991]). Unfortunately, the time required to compute the determinant exceeds $m\sqrt{n}$ for small m, and so the benefit in using this randomized *decision* procedure appears marginal at best. However, this technique was extended by Rabin and Vazirani [1984, 1989] to obtain simple algorithms for the actual *construction* of maximum matchings; although their randomized algorithms for matchings are simple and elegant, they are still slower than the deterministic $O(m\sqrt{n})$ time algorithms known earlier. Perhaps more significantly, this randomized decision procedure proved to be an essential ingredient in devising fast *parallel* algorithms for computing maximum matchings [Karp et al. 1988, Mulmuley et al. 1987].

Defining Terms

Deterministic algorithm: An algorithm whose execution is completely determined by its input.

Distributional complexity: The expected running time of the best possible deterministic algorithm over the worst possible probability distribution of the inputs.

Las Vegas algorithm: A randomized algorithm that always produces correct results, with the only variation from one run to another being in its running time.

Monte Carlo algorithm: A randomized algorithm that may produce incorrect results but with bounded error probability.

Randomized algorithm: An algorithm that makes random choices during the course of its execution.

Randomized complexity: The expected running time of the best possible randomized algorithm over the worst input.

References

Aleliunas, R., Karp, R. M., Lipton, R. J., Lovász, L., and Rackoff, C. 1979. Random walks, universal traversal sequences, and the complexity of maze problems. In *Proc. 20th Ann. Symp. Found. Comput. Sci.*, pp. 218–223. San Juan, Puerto Rico, Oct.

Aragon, C. R. and Seidel, R. G. 1989. Randomized search trees. In *Proc. 30th Ann. IEEE Symp. Found. Comput. Sci.*, pp. 540–545.

Ben-David, S., Borodin, A., Karp, R. M., Tardos, G., and Wigderson, A. 1994. On the power of randomization in on-line algorithms. *Algorithmica* 11(1):2–14.

Blum, M. and Kannan, S. 1989. Designing programs that check their work. In *Proc. 21st Annu. ACM Symp. Theory Comput.*, pp. 86–97. ACM.

Coppersmith, D. and Winograd, S. 1990. Matrix multiplication via arithmetic progressions. *J. Symbolic Comput.* 9:251–280.

DeMillo, R. A. and Lipton, R. J. 1978. A probabilistic remark on algebraic program testing. *Inf. Process. Lett.* 7:193–195.

Edmonds, J. 1967. Systems of distinct representatives and linear algebra. *J. Res. Nat. Bur. Stand. 71B,* 4:241–245.

Feder, T. and Motwani, R. 1991. Clique partitions, graph compression and speeding-up algorithms. In *Proc. 25th Annu. ACM Symp. Theory Comput.*, pp. 123–133.

Floyd, R. W. and Rivest, R. L. 1975. Expected time bounds for selection. *Commun. ACM* 18:165–172.

Freivalds, R. 1977. Probabilistic machines can use less running time. In *Inf. Process. 77, Proc. IFIP Congress 77,* B. Gilchrist, Ed., pp. 839–842, North-Holland, Amsterdam, Aug.

Goemans, M. X. and Williamson, D. P. 1994. 0.878-approximation algorithms for MAX-CUT and MAX-2SAT. In *Proc. 26th Annu. ACM Symp. Theory Comput.*, pp. 422–431.

Hoare, C. A. R. 1962. Quicksort. *Comput. J.* 5:10–15.

Hopcroft, J. E. and Karp, R. M. 1973. An $n^{5/2}$ algorithm for maximum matching in bipartite graphs. *SIAM J. Comput.* 2:225–231.

Karger, D. R. 1993. Global min-cuts in $\mathcal{RNC}$, and other ramifications of a simple min-cut algorithm. In *Proc. 4th Annu. ACM–SIAM Symp. Discrete Algorithms.*

Karger, D. R., Klein, P. N., and Tarjan, R. E. 1995. A randomized linear-time algorithm for finding minimum spanning trees. *J. ACM* 42:321–328.

Karger, D., Motwani, R., and Sudan, M. 1994. Approximate graph coloring by semidefinite programming. In *Proc. 35th Annu. IEEE Symp. Found. Comput. Sci.*, pp. 2–13.

Karp, R. M. 1991. An introduction to randomized algorithms. *Discrete Appl. Math.* 34:165–201.

Karp, R. M., Upfal, E., and Wigderson, A. 1986. Constructing a perfect matching is in random $\mathcal{NC}$. *Combinatorica* 6:35–48.

Karp, R. M., Upfal, E., and Wigderson, A. 1988. The complexity of parallel search. *J. Comput. Sys. Sci.* 36:225–253.

Lovász, L. 1979. On determinants, matchings and random algorithms. In *Fundamentals of Computing Theory.* L. Budach, Ed. Akademia-Verlag, Berlin.

Maffioli, F., Speranza, M. G., and Vercellis, C. 1985. Randomized algorithms. In *Combinatorial Optimization: Annotated Bibliographies*, M. O'Eigertaigh, J. K. Lenstra, and A. H. G. Rinooy Kan, Eds., pp. 89–105. Wiley, New York.

Micali, S. and Vazirani, V. V. 1980. An $O(\sqrt{|V|}|e|)$ algorithm for finding maximum matching in general graphs. In *Proc. 21st Annu. IEEE Symp. Found. Comput. Sci.*, pp. 17–27.

Motwani, R., Naor, J., and Raghavan, P. 1996. Randomization in approximation algorithms. In *Approximation Algorithms*, D. Hochbaum, Ed. PWS.

Motwani, R. and Raghavan, P. 1995. *Randomized Algorithms.* Cambridge University Press, New York.

Mulmuley, K. 1993. *Computational Geometry: An Introduction through Randomized Algorithms.* Prentice Hall, New York.

Mulmuley, K., Vazirani, U. V., and Vazirani, V. V. 1987. Matching is as easy as matrix inversion. *Combinatorica* 7:105–113.

Pugh, W. 1990. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM* 33(6):668–676.

Rabin, M. O. 1980. Probabilistic algorithm for testing primality. *J. Number Theory* 12:128–138.

Rabin, M. O. 1983. Randomized Byzantine generals. In *Proc. 24th Annu. Symp. Found. Comput. Sci.*, pp. 403–409.

Rabin, M. O. and Vazirani, V. V. 1984. Maximum matchings in general graphs through randomization. *Aiken Computation Lab. Tech. Rep.* TR-15-84, Harvard University, Cambridge, MA.

Rabin, M. O. and Vazirani, V. V. 1989. Maximum matchings in general graphs through randomization. *J. Algorithms* 10:557–567.

Raghavan, P. and Snir, M. 1994. Memory versus randomization in on-line algorithms. *IBM J. Res. Dev.* 38:683–707.

Saks, M. and Wigderson, A. 1986. Probabilistic Boolean decision trees and the complexity of evaluating game trees. In *Proc. 27th Annu. IEEE Symp. Found. Comput. Sci.*, pp. 29–38. Toronto, Ontario.

Schrijver, A. 1986. *Theory of Linear and Integer Programming.* Wiley, New York.

Schwartz, J. T. 1987. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM* 27(4):701–717.

Seidel, R. G. 1991. Small-dimensional linear programming and convex hulls made easy. *Discrete Comput. Geom.* 6:423–434.

Sinclair, A. 1992. *Algorithms for Random Generation and Counting: A Markov Chain Approach, Progress in Theoretical Computer Science.* Birkhauser, Boston, MA.

Snir, M. 1985. Lower bounds on probabilistic linear decision trees. *Theor. Comput. Sci.* 38:69–82.

Solovay, R. and Strassen, V. 1977. A fast Monte-Carlo test for primality. *SIAM J. Comput.* 6(1):84–85. *See also* 1978. *SIAM J. Comput.* 7(Feb.):118.

Tarsi, M. 1983. Optimal search on some game trees. *J. ACM* 30:389–396.

Tutte, W. T. 1947. The factorization of linear graphs. *J. London Math. Soc.* 22:107–111.

Valiant, L. G. 1982. A scheme for fast parallel communication. *SIAM J. Comput.* 11:350–361.

Vazirani, V. V. 1994. A theory of alternating paths and blossoms for proving correctness of $O(\sqrt{VE})$ graph maximum matching algorithms. *Combinatorica* 14(1):71–109.

Welsh, D. J. A. 1983. Randomised algorithms. *Discrete Appl. Math.* 5:133–145.

Yao, A. C.-C. 1977. Probabilistic computations: towards a unified measure of complexity. In *Proc. 17th Annu. Symp. Found. Comput. Sci.*, pp. 222–227.

Zippel, R. E. 1979. Probabilistic algorithms for sparse polynomials. In *Proc. EUROSAM 79,* Vol. 72, Lecture Notes in Computer Science., pp. 216–226. Marseille, France.

Further Information

In this section we give pointers to a plethora of randomized algorithms not covered in this chapter. The reader should also note that the examples in the text are but a (random!) sample of the many randomized

algorithms for each of the problems considered. These algorithms have been chosen to illustrate the main ideas behind randomized algorithms rather than to represent the state of the art for these problems. The reader interested in other algorithms for these problems is referred to Motwani and Raghavan [1995].

Randomized algorithms also find application in a number of other areas: in load balancing [Valiant 1982], approximation algorithms and combinatorial optimization [Goemans and Williamson 1994, Karger et al. 1994, Motwani et al. 1996], graph algorithms [Aleliunas et al. 1979, Karger et al. 1995], data structures [Aragon and Seidel 1989], counting and enumeration [Sinclair 1992], parallel algorithms [Karp et al. 1986, 1988], distributed algorithms [Rabin 1983], geometric algorithms [Mulmuley 1993], on-line algorithms [Ben-David et al. 1994, Raghavan and Snir 1994], and number-theoretic algorithms [Rabin 1983, Solovay and Strassen 1977]. The reader interested in these applications may consult these articles or Motwani and Raghavan [1995].

13

Pattern Matching and Text Compression Algorithms

Maxime Crochemore
University of Marne-la-Vallée and King's College London

Thierry Lecroq
University of Rouen

13.1 Processing Texts Efficiently

The present chapter describes a few standard algorithms used for processing texts. They apply, for example, to the manipulation of texts (text editors), to the storage of textual data (text compression), and to data retrieval systems. The algorithms of this chapter are interesting in different respects. First, they are basic components used in the implementations of practical software. Second, they introduce programming methods that serve as paradigms in other fields of computer science (system or software design). Third, they play an important role in theoretical computer science by providing challenging problems.

Although data is stored in various ways, text remains the main form of exchanging information. This is particularly evident in literature or linguistics where data is composed of huge corpora and dictionaries. This applies as well to computer science, where a large amount of data is stored in linear files. And this is also the case in molecular biology where biological molecules can often be approximated as sequences of

nucleotides or amino acids. Moreover, the quantity of available data in these fields tends to double every 18 months. This is the reason why algorithms should be efficient even if the speed of computers increases at a steady pace.

Pattern matching is the problem of locating a specific pattern inside raw data. The pattern is usually a collection of strings described in some formal language. Two kinds of textual patterns are presented: single strings and approximated strings. We also present two algorithms for matching patterns in images that are extensions of string-matching algorithms.

In several applications, texts need to be structured before being searched. Even if no further information is known about their syntactic structure, it is possible and indeed extremely efficient to build a data structure that supports searches. From among several existing data structures equivalent to represent indexes, we present the suffix tree, along with its construction.

The comparison of strings is implicit in the approximate pattern searching problem. Because it is sometimes required to compare just two strings (files or molecular sequences), we introduce the basic method based on longest common subsequences.

Finally, the chapter contains two classical text compression algorithms. Variants of these algorithms are implemented in practical compression software, in which they are often combined together or with other elementary methods. An example of mixing different methods is presented there.

The efficiency of algorithms is evaluated by their running times, and sometimes by the amount of memory space they require at runtime as well.

13.2 String-Matching Algorithms

String matching is the problem of finding one or, more generally, all the **occurrences** of a pattern in a text. The pattern and the text are both strings built over a finite alphabet (a finite set of symbols). Each algorithm of this section outputs all occurrences of the pattern in the text. The pattern is denoted by $x = x[0\,..\,m-1]$; its length is equal to m. The text is denoted by $y = y[0\,..\,n-1]$; its length is equal to n. The alphabet is denoted by Σ and its size is equal to σ.

String-matching algorithms of the present section work as follows: they first align the left ends of the pattern and the text, then compare the aligned symbols of the text and the pattern — this specific work is called an attempt or a scan, and after a whole match of the pattern or after a mismatch, they shift the pattern to the right. They repeat the same procedure again until the right end of the pattern goes beyond the right end of the text. This is called the scan and shift mechanism. We associate each attempt with the position j in the text, when the pattern is aligned with $y[j\,..\,j+m-1]$.

The brute-force algorithm consists of checking, at all positions in the text between 0 and $n-m$, whether an occurrence of the pattern starts there or not. Then, after each attempt, it shifts the pattern exactly one position to the right. This is the simplest algorithm, which is described in Figure 13.1.

The time complexity of the brute-force algorithm is $O(mn)$ in the worst case but its behavior in practice is often linear on specific data.

```
BF(x, m, y, n)
1  ▷ Searching
2  for j ← 0 to n − m
3       do i ← 0
4          while i < m and x[i] = y[i + j]
5                do i ← i + 1
6          if i ≥ m
7            then OUTPUT(j)
```

FIGURE 13.1 The brute-force string-matching algorithm.

13.2.1 Karp–Rabin Algorithm

Hashing provides a simple method for avoiding a quadratic number of symbol comparisons in most practical situations. Instead of checking at each position of the text whether the pattern occurs, it seems to be more efficient to check only if the portion of the text aligned with the pattern "looks like" the pattern. To check the resemblance between these portions, a hashing function is used. To be helpful for the string-matching problem, the hashing function should have the following properties:

- Efficiently computable
- Highly discriminating for strings
- $hash(y[j+1\,..\,j+m])$ must be easily computable from $hash(y[j\,..\,j+m-1])$; $hash(y[j+1\,..\,j+m]) = \text{REHASH}(y[j], y[j+m], hash(y[j\,..\,j+m-1]))$

For a word w of length k, its symbols can be considered as digits, and we define $hash(w)$ by:

$$hash(w[0\,..\,k-1]) = (w[0] \times 2^{k-1} + w[1] \times 2^{k-2} + \cdots + w[k-1]) \bmod q$$

where q is a large number. Then, REHASH has a simple expression

$$\text{REHASH}(a, b, h) = ((h - a \times d) \times 2 + b) \bmod q$$

where $d = 2^{k-1}$ and q is the computer word-size (see Figure 13.2).

During the search for the pattern x, $hash(x)$ is compared with $hash(y[j-m+1\,..\,j])$ for $m-1 \leq j \leq n-1$. If an equality is found, it is still necessary to check the equality $x = y[j-m+1\,..\,j]$ symbol by symbol.

In the algorithms of Figures 13.2 and 13.3, all multiplications by 2 are implemented by shifts (operator $<<$). Furthermore, the computation of the modulus function is avoided by using the implicit modular

```
REHASH(a, b, h)
1  return ((h − a × d) << 1) + b
```

FIGURE 13.2 Function REHASH

```
KR(x, m, y, n)
 1  ▷ Preprocessing
 2  d ← 1
 3  for i ← 1 to m − 1
 4       do d ← d << 1
 5  h_x ← 0
 6  h_y ← 0
 7  for i ← 0 to m − 1
 8       do h_x ← (h_x << 1) + x[i]
 9          h_y ← (h_y << 1) + y[i]
10  ▷ Searching
11  if h_x = h_y and x = y[0 .. m − 1]
12    then OUTPUT(0)
13  j ← m
14  while j < n
15       do h_y ← REHASH(y[j − m], y[j], h_y)
16          if h_x = h_y and x = y[j − m + 1 .. j]
17            then OUTPUT(j − m + 1)
18          j ← j + 1
```

FIGURE 13.3 The Karp–Rabin string-matching algorithm.

arithmetic given by the hardware that forgets carries in integer operations. Thus, q is chosen as the maximum value of an integer of the system.

The worst-case time complexity of the Karp–Rabin algorithm (Figure 13.3) is quadratic (as it is for the brute-force algorithm), but its expected running time is $O(m+n)$.

Example 13.1

Let $x =$ **ing**. Then, $hash(x) = 105 \times 2^2 + 110 \times 2 + 103 = 743$ (symbols are assimilated with their ASCII codes):

y =	**s**	**t**	**r**	**i**	**n**	**g**		**m**	**a**	**t**	**c**	**h**	**i**	**n**	**g**
hash =			806	797	776	743	678	585	443	746	719	766	709	736	743

13.2.2 Knuth–Morris–Pratt Algorithm

This section presents the first discovered linear-time string-matching algorithm. Its design follows a tight analysis of the brute-force algorithm, and especially the way this latter algorithm wastes the information gathered during the scan of the text.

Let us look more closely at the brute-force algorithm. It is possible to improve the length of shifts and simultaneously remember some portions of the text that match the pattern. This saves comparisons between characters of the text and of the pattern, and consequently increases the speed of the search.

Consider an attempt at position j, that is, when the pattern $x[0\,..\,m-1]$ is aligned with the segment $y[j\,..\,j+m-1]$ of the text. Assume that the first mismatch (during a left-to-right scan) occurs between symbols $x[i]$ and $y[i+j]$ for $0 \le i < m$. Then, $x[0\,..\,i-1] = y[j\,..\,i+j-1] = u$ and $a = x[i] \neq y[i+j] = b$. When shifting, it is reasonable to expect that a **prefix** v of the pattern matches some **suffix** of the portion u of the text. Moreover, if we want to avoid another immediate mismatch, the letter following the prefix v in the pattern must be different from a. (Indeed, it should be expected that v matches a suffix of ub, but elaborating along this idea goes beyond the scope of the chapter.) The longest such prefix v is called the **border** of u (it occurs at both ends of u). This introduces the notation: let $next[i]$ be the length of the longest (proper) border of $x[0\,..\,i-1]$, followed by a character c different from $x[i]$. Then, after a shift, the comparisons can resume between characters $x[next[i]]$ and $y[i+j]$ without missing any occurrence of x in y and having to backtrack on the text (see Figure 13.4).

Example 13.2

Here,

y =	.	.	.	**a**	**b**	**a**	**b**	**a**	**a**	**b**	.	.	.	.	.
x =				**a**	**b**	**a**	**b**	**a**	**b**	**a**					
x =									**a**	**b**	**a**	**b**	**a**	**b**	**a**

Compared symbols are underlined. Note that the empty string is the suitable border of **ababa**. Other borders of **ababa** are **aba** and **a**.

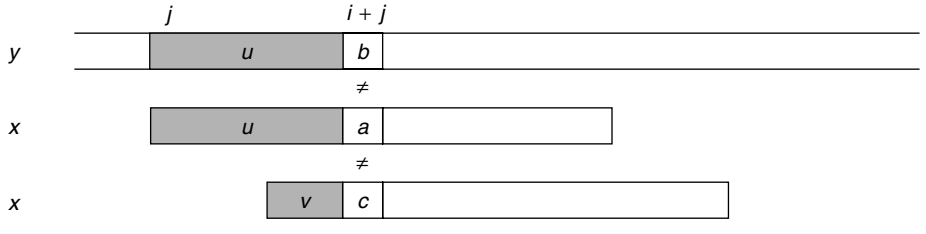


FIGURE 13.4 Shift in the Knuth–Morris–Pratt algorithm (v suffix of u).

```
KMP(x, m, y, n)
 1  ▷ Preprocessing
 2  next ← PREKMP(x, m)
 3  ▷ Searching
 4  i ← 0
 5  j ← 0
 6  while j < n
 7      do while i > −1 and x[i] ≠ y[j]
 8             do i ← next[i]
 9         i ← i + 1
10         j ← j + 1
11         if i ≥ m
12           then OUTPUT(j − i)
13                i ← next[i]
```

FIGURE 13.5 The Knuth–Morris–Pratt string-matching algorithm.

```
PREKMP(x, m)
 1  i ← −1
 2  j ← 0
 3  next[0] ← −1
 4  while j < m
 5      do while i > −1 and x[i] ≠ x[j]
 6             do i ← next[i]
 7         i ← i + 1
 8         j ← j + 1
 9         if x[i] = x[j]
10           then next[j] ← next[i]
11           else next[j] ← i
12  return next
```

FIGURE 13.6 Preprocessing phase of the Knuth–Morris–Pratt algorithm: computing *next*.

The Knuth–Morris–Pratt algorithm is displayed in Figure 13.5. The table *next* it uses is computed in $O(m)$ time before the search phase, applying the same searching algorithm to the pattern itself, as if $y = x$ (see Figure 13.6). The worst-case running time of the algorithm is $O(m + n)$ and it requires $O(m)$ extra space. These quantities are independent of the size of the underlying alphabet.

13.2.3 Boyer–Moore Algorithm

The Boyer–Moore algorithm is considered the most efficient string-matching algorithm in usual applications. A simplified version of it, or the entire algorithm, is often implemented in text editors for the search and substitute commands.

The algorithm scans the characters of the pattern from right to left, beginning with the rightmost symbol. In case of a mismatch (or a complete match of the whole pattern), it uses two precomputed functions to shift the pattern to the right. These two shift functions are called the *bad-character shift* and the *good-suffix shift*. They are based on the following observations.

Assume that a mismatch occurs between the character $x[i] = a$ of the pattern and the character $y[i+j] = b$ of the text during an attempt at position j. Then, $x[i+1\,..\,m-1] = y[i+j+1\,..\,j+m-1] = u$ and $x[i] \neq y[i+j]$. The good-suffix shift consists in aligning the **segment** $y[i+j+1\,..\,j+m-1]$ with its rightmost occurrence in x that is preceded by a character different from $x[i]$ (see Figure 13.7). If there exists no such segment, the shift consists in aligning the longest suffix v of $y[i+j+1\,..\,j+m-1]$ with a matching prefix of x (see Figure 13.8).

FIGURE 13.7 The good-suffix shift, when u reappears, preceded by a character different from a.

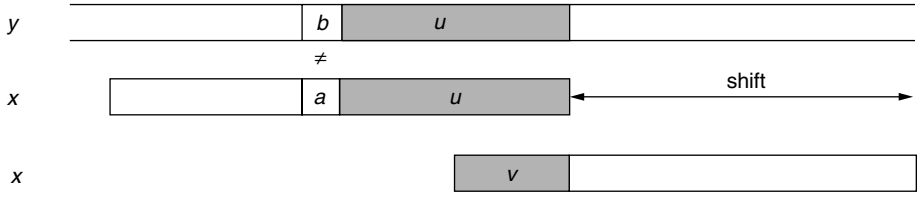


FIGURE 13.8 The good-suffix shift, when the situation of Figure 13.7 does not happen, only a suffix of u reappears as a prefix of x.

Example 13.3

Here,

$$
\begin{array}{lcccccccccccccccc}
y = & . & . & . & \mathbf{a} & \mathbf{b} & \mathbf{b} & \mathbf{a} & \mathbf{a} & \mathbf{b} & \mathbf{b} & \mathbf{a} & \mathbf{b} & \mathbf{b} & \mathbf{a} & . & . & . \\
x = & \mathbf{a} & \mathbf{b} & \mathbf{b} & \mathbf{a} & \mathbf{a} & \mathbf{b} & \underline{\mathbf{b}} & \underline{\mathbf{a}} & \underline{\mathbf{b}} & \underline{\mathbf{b}} & \underline{\mathbf{a}} & & & & & & \\
x = & & & & \underline{\mathbf{a}} & \underline{\mathbf{b}} & \underline{\mathbf{b}} & \underline{\mathbf{a}} & \underline{\mathbf{a}} & \underline{\mathbf{b}} & \underline{\mathbf{b}} & \underline{\mathbf{a}} & \underline{\mathbf{b}} & \underline{\mathbf{b}} & \underline{\mathbf{a}} & & &
\end{array}
$$

The shift is driven by the suffix **abba** of x found in the text. After the shift, the segment **abba** in the middle of y matches a segment of x as in Figure 13.7. The same mismatch does not recur.

Example 13.4

Here,

$$
\begin{array}{lcccccccccccccccccc}
y = & . & . & . & \mathbf{a} & \mathbf{b} & \mathbf{b} & \mathbf{a} & \mathbf{a} & \mathbf{b} & \mathbf{b} & \mathbf{a} & \mathbf{b} & \mathbf{b} & \mathbf{a} & \mathbf{b} & \mathbf{b} & \mathbf{a} & . & . \\
x = & & & \mathbf{b} & \mathbf{b} & \mathbf{a} & \mathbf{b} & \underline{\mathbf{b}} & \underline{\mathbf{a}} & \underline{\mathbf{b}} & \underline{\mathbf{b}} & \underline{\mathbf{a}} & & & & & & & & \\
x = & & & & & & & & \underline{\mathbf{b}} & \underline{\mathbf{b}} & \underline{\mathbf{a}} & \underline{\mathbf{b}} & \underline{\mathbf{b}} & \underline{\mathbf{a}} & \underline{\mathbf{b}} & \underline{\mathbf{b}} & \underline{\mathbf{a}} & & &
\end{array}
$$

The segment **abba** found in y partially matches a prefix of x after the shift, as in Figure 13.8.

The bad-character shift consists in aligning the text character $y[i+j]$ with its rightmost occurrence in $x[0\,..\,m-2]$ (see Figure 13.9). If $y[i+j]$ does not appear in the pattern x, no occurrence of x in y can overlap the symbol $y[i+j]$, then the left end of the pattern is aligned with the character at position $i+j+1$ (see Figure 13.10).

Example 13.5

Here,

$$
\begin{array}{lcccccccccccccc}
y = & . & . & . & . & . & . & \mathbf{a} & \mathbf{b} & \mathbf{c} & \mathbf{d} & . & . & . & . \\
x = & \mathbf{c} & \mathbf{d} & \mathbf{a} & \mathbf{h} & \mathbf{g} & \mathbf{f} & \underline{\mathbf{e}} & \underline{\mathbf{b}} & \underline{\mathbf{c}} & \underline{\mathbf{d}} & & & & & \\
x = & & & & & \mathbf{c} & \mathbf{d} & \mathbf{a} & \mathbf{h} & \mathbf{g} & \mathbf{f} & \mathbf{e} & \mathbf{b} & \mathbf{c} & \underline{\mathbf{d}}
\end{array}
$$

The shift aligns the symbol **a** in x with the mismatch symbol **a** in the text y (Figure 13.9).

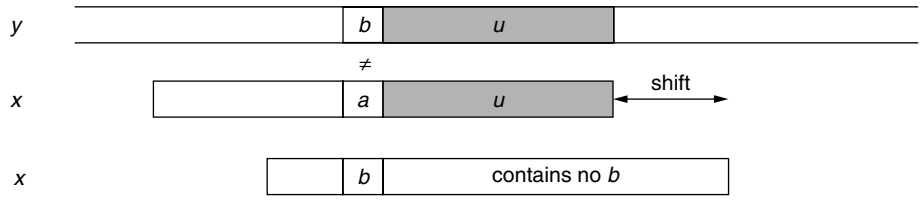


FIGURE 13.9 The bad-character shift, b appears in x.

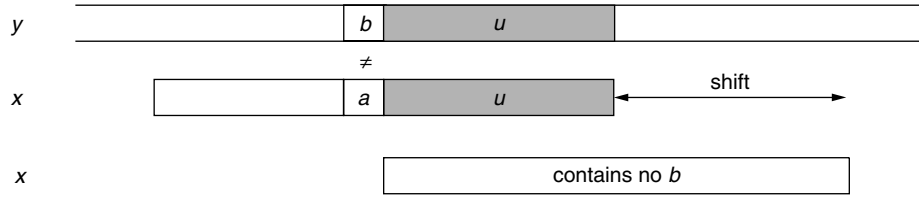


FIGURE 13.10 The bad-character shift, b does not appear in x (except possibly at $m-1$).

```
BM(x, m, y, n)
 1  ▷ Preprocessing
 2  gs ← PREGS(x, m)
 3  bc ← PREBC(x, m)
 4  ▷ Preprocessing
 5  j ← 0
 6  while j ≤ n − m
 7      do i ← m − 1
 8         while i ≥ 0 and x[i] = y[i + j]
 9             do i ← i − 1
10         if i < 0
11           then OUTPUT(j)
12         j ← max{gs[i + 1], bc[y[i + j] − m + i + 1]}
```

FIGURE 13.11 The Boyer–Moore string-matching algorithm.

Example 13.6

Here,

$$
\begin{array}{lcccccccccccccc}
y = & . & . & . & . & . & \mathbf{a} & \mathbf{b} & \mathbf{c} & \mathbf{d} & . & . & . & . & . & . \\
x = & \mathbf{c} & \mathbf{d} & \mathbf{h} & \mathbf{g} & \mathbf{f} & \underline{\mathbf{e}} & \underline{\mathbf{b}} & \underline{\mathbf{c}} & \underline{\mathbf{d}} & & & & & & \\
x = & & & & & & & \mathbf{c} & \mathbf{d} & \mathbf{h} & \mathbf{g} & \mathbf{f} & \mathbf{e} & \mathbf{b} & \mathbf{c} & \underline{\mathbf{d}}
\end{array}
$$

The shift positions the left end of x right after the symbol $\mathbf{a}$ of y (Figure 13.10).

The Boyer–Moore algorithm is shown in Figure 13.11. For shifting the pattern, it applies the maximum between the bad-character shift and the good-suffix shift. More formally, the two shift functions are defined as follows. The bad-character shift is stored in a table bc of size σ and the good-suffix shift is stored in a table gs of size $m+1$. For $a \in \Sigma$

$$
bc[a] = \begin{cases} \min\{i \mid 1 \le i < m \text{ and } x[m-1-i] = a\} & \text{if } a \text{ appears in } x, \\ m & \text{otherwise.} \end{cases}
$$

```
PREBC(x, m)
1  for a ← firstLetter to lastLetter
2      do bc[a] ← m
3  for i ← 0 to m − 2
4      do bc[x[i]] ← m − 1 − i
5  return bc
```

FIGURE 13.12 Computation of the bad-character shift.

```
SUFFIXES(x, m)
 1  suff[m − 1] ← m
 2  g ← m − 1
 3  for i ← m − 2 downto 0
 4      do if i > g and suff[i + m − 1 − f] ≠ i − g
 5             then suff[i] ← min{suff[i + m − 1 − f], i − g}
 6             else if i < g
 7                      then g ← i
 8                  f ← i
 9                  while g ≥ 0 and x[g] = x[g + m − 1 − f]
10                      do g ← g − 1
11                  suff[i] ← f − g
12  return suff
```

FIGURE 13.13 Computation of the table *suff*.

Let us define two conditions,

$$\begin{cases} cond_1(i,s): \text{ for each } k \text{ such that } i < k < m, s \geq k \text{ or } x[k-s] = x[k], \\ cond_2(i,s): \text{ if } s < i \text{ then } x[i-s] \neq x[i]. \end{cases}$$

Then, for $0 \leq i < m$,

$$gs[i+1] = \min\{s > 0 \mid cond_1(i,s) \text{ and } cond_2(i,s) \text{ hold}\}$$

and we define $gs[0]$ as the length of the smallest period of x.

To compute the table *gs*, a table *suff* is used. This table can be defined as follows: for $i = 0, 1, \ldots, m-1$,

$$suff[i] = \text{longest common suffix between } x[0\,..\,i] \text{ and } x\,.$$

It is computed in linear time and space by the function SUFFIXES (see Figure 13.13).

Tables *bc* and *gs* can be precomputed in time $O(m + \sigma)$ before the search phase and require an extra space in $O(m + \sigma)$ (see Figure 13.12 and Figure 13.14). The worst-case running time of the algorithm is quadratic. However, on large alphabets (relative to the length of the pattern), the algorithm is extremely fast. Slight modifications of the strategy yield linear-time algorithms (see the bibliographic notes). When searching for a^m in $(a^{m-1}b)^{\lfloor n/m \rfloor}$, the algorithm makes only $O(n/m)$ comparisons, which is the absolute minimum for any string-matching algorithm in the model where the pattern only is preprocessed.

13.2.4 Quick Search Algorithm

The bad-character shift used in the Boyer–Moore algorithm is not very efficient for small alphabets; but when the alphabet is large compared with the length of the pattern, as is often the case with the ASCII table and ordinary searches made under a text editor, it becomes very useful. Using it alone produces a practically very efficient algorithm that is described now.

After an attempt where x is aligned with $y[j\,..\,j+m-1]$, the length of the shift is at least equal to one. Thus, the character $y[j+m]$ is necessarily involved in the next attempt, and thus can be used for

```
PREGS(x, m)
 1  gs ← SUFFIXES(x, m)
 2  for i ← 0 to m − 1
 3      do gs[i] ← m
 4  j ← 0
 5  for i ← m − 1 downto −1
 6      do if i = −1 or suff[i] = i + 1
 7            then while j < m − 1 − i
 8                       do if gs[j] = m
 9                             then gs[j] ← m − 1 − i
10                          j ← j + 1
11  for i ← 0 to m − 2
12      do gs[m − 1 − suff[i]] ← m − 1 − i
13  return gs
```

FIGURE 13.14 Computation of the good-suffix shift.

```
QS(x, m, y, n)
 1  ▷ Preprocessing
 2  for a ← firstLetter to lastLetter
 3      do bc[a] ← m + 1
 4  for i ← 0 to m − 1
 5      do bc[x[i]] ← m − i
 6  ▷ Searching
 7  j ← 0
 8  while j ≤ n − m
 9      do i ← 0
10         while i ≥ 0 and x[i] = y[i + j]
11            do i ← i + 1
12         if i ≥ m
13           then OUTPUT(j)
14         j ← bc[y[j + m]]
```

FIGURE 13.15 The Quick Search string-matching algorithm.

the bad-character shift of the current attempt. In the present algorithm, the bad-character shift is slightly modified to take into account the observation as follows ($a \in \Sigma$):

$$bc[a] = 1 + \begin{cases} \min\{i \mid 0 \le i < m \text{ and } x[m-1-i] = a\} & \text{if } a \text{ appears in } x, \\ m & \text{otherwise.} \end{cases}$$

Indeed, the comparisons between text and pattern characters during each attempt can be done in any order. The algorithm of Figure 13.15 performs the comparisons from left to right. It is called Quick Search after its inventor and has a quadratic worst-case time complexity but good practical behavior.

Example 13.7

Here,

```
y =  s t r i n g - m a t c h i n g
x =  i n g
x =      i n g
x =          i n g
x =              i n g
x =                  i n g
```

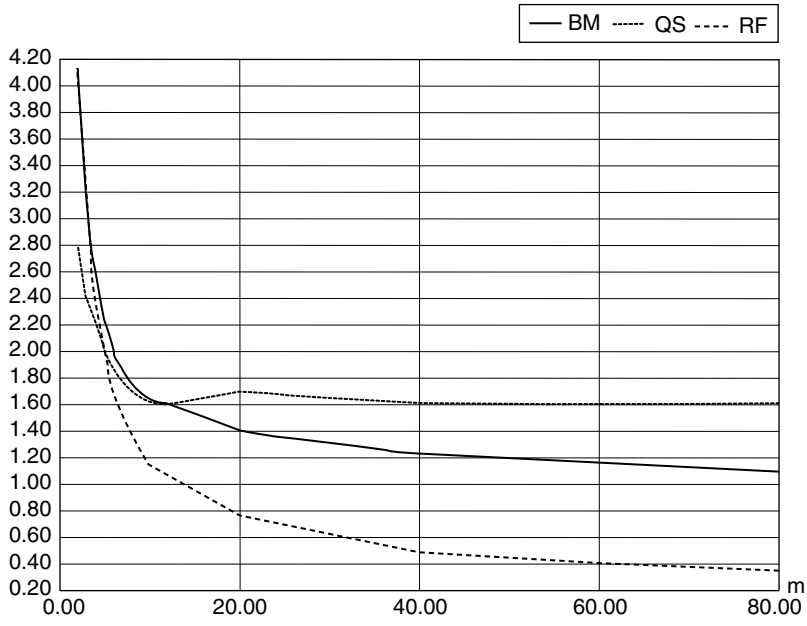


FIGURE 13.16 Running times for a DNA sequence.

The Quick Search algorithm makes only nine comparisons to find the two occurrences of **ing** inside the text of length 15.

13.2.5 Experimental Results

In Figure 13.16 and Figure 13.17, we present the running times of three string-matching algorithms: the Boyer–Moore algorithm (BM), the Quick Search algorithm (QS), and the Reverse-Factor algorithm (RF). The Reverse–Factor algorithm can be viewed as a variation of the Boyer–Moore algorithm where factors (segments) rather than suffixes of the pattern are recognized. The RF algorithm uses a data structure to store all the factors of the reversed pattern: a suffix automaton or a **suffix tree**.

Tests have been performed on various types of texts. In Figure 13.16 we show the results when the text is a DNA sequence on the four-letter alphabet of nucleotides **A**, **C**, **G**, **T**. In Figure 13.17 English text is considered.

For each pattern length, we ran a large number of searches with random patterns. The average time according to the length is shown in the two figures. The running times of both preprocessing and searching phases are added. The three algorithms are implemented in a homogeneous way in order to keep the comparison significant.

For the genome, as expected, the QS algorithm is the best for short patterns. But for long patterns it is less efficient than the BM algorithm. In this latter case, the RF algorithm achieves the best results. For rather large alphabets, as is the case for an English text, the QS algorithm remains better than the BM algorithm whatever the pattern length is. In this case, the three algorithms have similar behaviors; however, the QS is better for short patterns (which is typical of search under a text editor) and the RF is better for large patterns.

13.2.6 Aho–Corasick Algorithm

The Unix operating system provides standard text (or file) facilities. Among them is the series of **grep** commands that locate patterns in files. We describe in this section the algorithm underlying the **fgrep**

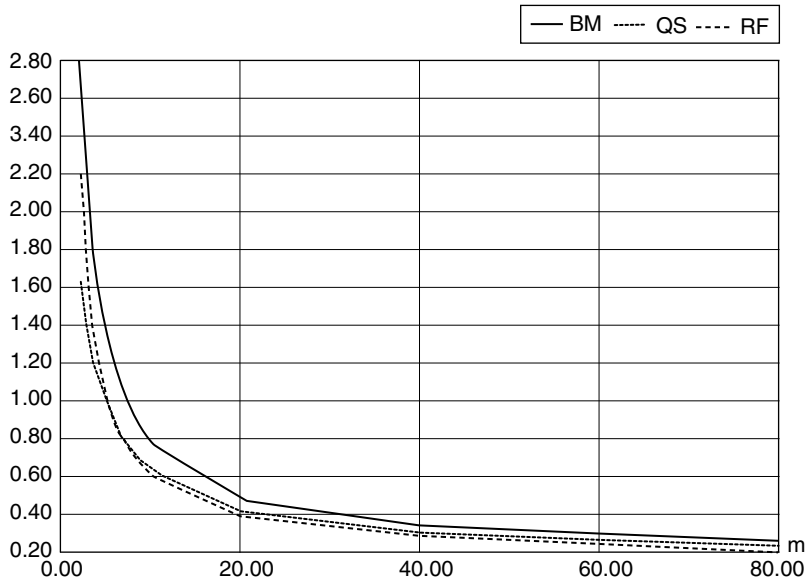


FIGURE 13.17 Running times for an English text.

```
PREAC(X, k)
 1  Create a new node root
 2  ▷ creates a loop on the root of the trie
 3  for a ∈ Σ
 4      do child(root, a) ← root
 5  ▷ enters each pattern in the trie
 6  for i ← 0 to k − 1
 7      do ENTER(X[i], root)
 8  ▷ completes the trie with failure links
 9  COMPLETE(root)
10  return root
```

FIGURE 13.18 Preprocessing phase of the Aho–Corasick algorithm.

command of Unix. It searches files for a finite set of strings, and can, for instance, output lines containing at least one of the strings.

If we are interested in searching for all occurrences of all patterns taken from a finite set of patterns, a first solution consists in repeating some string-matching algorithm for each pattern. If the set contains k patterns, this search runs in time $O(kn)$. The solution described in the present section and designed by Aho and Corasick runs in time $O(n \log \sigma)$. The algorithm is a direct extension of the Knuth–Morris–Pratt algorithm, and the running time is independent of the number of patterns.

Let $X = \{x_0, x_1, \ldots, x_{k-1}\}$ be the set of patterns, and let $|X| = |x_0| + |x_1| + \cdots + |x_{k-1}|$ be the total size of the set X. The Aho–Corasick algorithm first builds a **trie** $T(X)$, a digital tree recognizing the patterns of X. The trie $T(X)$ is a tree in which edges are labeled by letters and in which branches spell the patterns of X. We identify a node p in the trie $T(X)$ with the unique word w spelled by the path of $T(X)$ from its root to p. The root itself is identified with the empty word ε. Notice that if w is a node in $T(X)$ then w is a prefix of some $x_i \in X$. If w is a node in $T(X)$ and $a \in \Sigma$ then $child(w, a)$ is equal to wa if wa is a node in $T(X)$; it is equal to UNDEFINED otherwise.

The function PREAC in Figure 13.18 returns the trie of all patterns. During the second phase, where patterns are entered in the trie, the algorithm initializes an output function out. It associates the singleton

```
ENTER(x, root)
 1  r ← root
 2  i ← 0
 3  ▷ follows the existing edges
 4  while i < |x| and child(r, x[i]) ≠ UNDEFINED and child(r, x[i]) ≠ root
 5      do r ← child(r, x[i])
 6         i ← i + 1
 7  ▷ creates new edges
 8  while i < |x|
 9      do Create a new node s
10         child(r, x[i]) ← s
11         r ← s
12         i ← i + 1
13  out(r) ← {x}
```

FIGURE 13.19 Construction of the trie.

```
COMPLETE(root)
 1  q ← empty queue
 2  ℓ ← list of the edges (root, a, p) for any character a ∈ Σ and any node p ≠ root
 3  while the list ℓ is not empty
 4      do (r, a, p) ← FIRST(ℓ)
 5         ℓ ← NEXT(ℓ)
 6         ENQUEUE(q, p)
 7         fail(p) ← root
 8  while the queue q is not empty
 9      do r ← DEQUEUE(q)
10         ℓ ← list of the edges (r, a, p) for any character a ∈ Σ and any node p
11         while the list ℓ is not empty
12             do (r, a, p) ← FIRST(ℓ)
13                ℓ ← NEXT(ℓ)
14                ENQUEUE(q, p)
15                s ← fail(r)
16                while child(s, a) = UNDEFINED
17                    do s ← fail(s)
18                fail(p) ← child(s, a)
19                out(p) ← out(p) ∪ out(child(s, a))
```

FIGURE 13.20 Completion of the output function and construction of failure links.

$\{x_i\}$ with the nodes x_i $(0 \leq i < k)$, and associates the empty set with all other nodes of $T(X)$ (see Figure 13.19).

Finally, the last phase of function PREAC (Figure 13.18) consists in building the failure link of each node of the trie, and simultaneously completing the output function. This is done by the function COMPLETE in Figure 13.20. The failure function *fail* is defined on nodes as follows (w is a node):

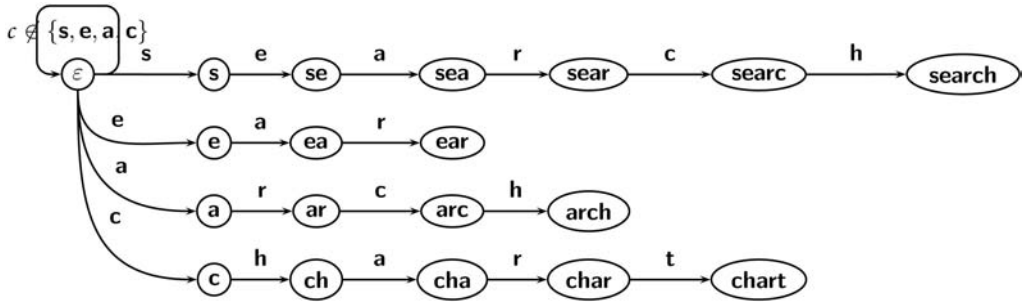
$$fail(w) = u \quad \text{where } u \text{ is the longest proper suffix of } w \text{ that belongs to } T(X).$$

Computation of failure links is done during a breadth-first traversal of $T(X)$. Completion of the output function is done while computing the failure function *fail* using the following rule:

$$\text{if } fail(w) = u \text{ then } out(w) = out(w) \cup out(u).$$

Example 13.8

Here, $X = \{\textbf{search}, \textbf{ear}, \textbf{arch}, \textbf{chart}\}$



nodes	ε	s	se	sea	sear	searc	search	e	ea	ear
fail	ε	ε	e	ea	ear	arc	arch	ε	a	ar

nodes	a	ar	arc	arch	c	ch	cha	char	chart	
fail	ε	ε	c	ch	ε	ε	a	ar	ε	

nodes	sear	search	ear	arch	chart
out	ear	{search, arch}	ear	arch	chart

To stop going back with failure links during the computation of the failure links, and also to overpass text characters for which no transition is defined from the root, a loop is added on the root of the trie for these symbols. This is done at the first phase of function PREAC.

After the preprocessing phase is completed, the searching phase consists in parsing all the characters of the text y with $T(X)$. This starts at the root of $T(X)$ and uses failure links whenever a character in y does not match any label of outgoing edges of the current node. Each time a node with a nonempty output is encountered, this means that the patterns of the output have been discovered in the text, ending at the current position. Then, the position is output.

An implementation of the Aho–Corasick algorithm from the previous discussion is shown in Figure 13.21. Note that the algorithm processes the text in an on-line way, so that the buffer on the text can be limited to only one symbol. Also note that the instruction $r \leftarrow \mathit{fail}(r)$ in Figure 13.21 is the exact analogue of instruction $i \leftarrow \mathit{next}[i]$ in Figure 13.5. A unified view of both algorithms exists but is beyond the scope of the chapter.

The entire algorithm runs in time $O(|X| + n)$ if the *child* function is implemented to run in constant time. This is the case for any fixed alphabet. Otherwise, a $\log \sigma$ multiplicative factor comes from access to the children nodes.

```
AC(X, k, y, n)
1  ▷ Preprocessing
2  r ← PREAC(X, k)
3  ▷ Searching
4  for j ← 0 to n − 1
5      do while child(r, y[j]) = UNDEFINED
6             do r ← fail(r)
7         r ← child(r, y[j])
8         if out(r) ≠ ∅
9           then OUTPUT((out(r), j))
```

FIGURE 13.21 The complete Aho–Corasick algorithm.

13.3 Two-Dimensional Pattern Matching Algorithms

In this section we consider only two-dimensional arrays. Arrays can be thought of as bit map representations of images, where each cell of arrays contains the codeword of a pixel. The string-matching problem finds an equivalent formulation in two dimensions (and even in any number of dimensions), and algorithms of Section 13.2 can be extended to operate on arrays.

The problem now is to locate all occurrences of a two-dimensional pattern $X = X[0\,..\,m_1-1, 0\,..\,m_2-1]$ of size $m_1 \times m_2$ inside a two-dimensional text $Y = Y[0\,..\,n_1 - 1, 0\,..\,n_2 - 1]$ of size $n_1 \times n_2$. The brute-force algorithm for this problem is given in Figure 13.22. It consists in checking at all positions of $Y[0\,..\,n_1 - m_1, 0\,..\,n_2 - m_2]$ if the pattern occurs. This algorithm has a quadratic (with respect to the size of the problem) worst-case time complexity in $O(m_1 m_2 n_1 n_2)$. We present in the next sections two more efficient algorithms. The first one is an extension of the Karp–Rabin algorithm (previous section). The second one solves the problem in linear time on a fixed alphabet; it uses both the Aho–Corasick and the Knuth–Morris–Pratt algorithms.

13.3.1 Zhu–Takaoka Algorithm

As for one-dimensional string matching, it is possible to check if the pattern occurs in the text only if the *aligned* portion of the text looks like the pattern. To do that, the idea is to use vertically the hash function method proposed by Karp and Rabin. To initialize the process, the two-dimensional arrays X and Y are translated into one-dimensional arrays of numbers x and y. The translation from X to x is done as follows ($0 \le i < m_2$):

$$x[i] = hash(X[0,i]X[1,i]\cdots X[m_1 - 1, i])$$

and the translation from Y to y is done by ($0 \le i < m_2$):

$$y[i] = hash(Y[0,i]Y[1,i]\cdots Y[m_1 - 1, i]).$$

The fingerprint y helps to find occurrences of X starting at row $j = 0$ in Y. It is then updated for each new row in the following way ($0 \le i < m_2$):

$$\begin{aligned} &hash(Y[j+1,i]Y[j+2,i]\cdots Y[j+m_1,i]) \\ &\quad = \text{REHASH}(Y[j,i], Y[j+m_1,i], hash(Y[j,i]Y[j+1,i]\cdots Y[j+m_1-1,i])) \end{aligned}$$

(functions *hash* and REHASH are described in the section on the Karp–Rabin algorithm).

```
BF2D(X, m1, m2, Y, n1, n2)
1  ▷ Searching
2  for j1 ← 0 to n1 − m1
3      do for j2 ← 0 to n2 − m2
4             do i ← 0
5                while i < m1 and x[i, 0 .. m2 − 1] = y[j1 + i, j2 .. j2 + m2 − 1]
6                      do i ← i + 1
7                if i ≥ m1
8                  then OUTPUT(j1, j2)
```

FIGURE 13.22 The brute-force two-dimensional pattern matching algorithm.

```
KMP-IN-LINE(X, m1, m2, Y, n1, n2, x, y, next, j1)
 1  i2 ← 0
 2  j2 ← 0
 3  while j2 < n2
 4      do while i2 > −1 and x[i2] ≠ y[j2]
 5             do i2 ← next[i2]
 6         i2 ← i2 + 1
 7         j2 ← j2 + 1
 8         if i2 ≥ m2
 9           then DIRECT-COMPARE(X, m1, m2, Y, n1, n2, j1, j2 − 1)
10                i2 ← next[m2]
```

FIGURE 13.23 Search for x in y using KMP algorithm.

```
DIRECT-COMPARE(X, m1, m2, Y, row, column)
1  j1 ← row − m1 + 1
2  j2 ← column − m2 + 1
3  for i1 ← 0 to m1 − 1
4      do for i2 ← 0 to m2 − 1
5             do if X[i1, i2] ≠ Y[i1 + j1, i2 + j2]
6                  then return
7  OUTPUT(j1, j2)
```

FIGURE 13.24 Naive check of an occurrence of x in y at position (*row, column*).

Example 13.9

$X =$

a	**a**	**a**
b	**b**	**a**
a	**a**	**b**

$Y =$

a	**b**	**a**	**b**	**a**	**b**	**b**
a	**a**	**a**	**a**	**b**	**b**	**b**
b	**b**	**b**	**a**	**a**	**a**	**b**
a	**a**	**a**	**b**	**b**	**a**	**a**
b	**b**	**a**	**a**	**a**	**b**	**b**
a	**a**	**b**	**a**	**b**	**a**	**a**

$x =$

681	681	680

$y =$

680	684	680	683	681	685	686

Next value of y is

681	681	681	680	684	683	685

. The occurrence of x at position 1 on y corresponds to an occurrence of X at position $(1, 1)$ on Y.

Since the alphabet of x and y is large, searching for x in y must be done by a string-matching algorithm for which the running time is independent of the size of the alphabet: the Knuth–Morris–Pratt suits this application perfectly. Its adaptation is shown in Figure 13.23.

When an occurrence of x is found in y, then we still have to check if an occurrence of X starts in Y at the corresponding position. This is done naively by the procedure of Figure 13.24.

The Zhu–Takaoka algorithm as explained above is displayed in Figure 13.25. The search for the pattern is performed row by row starting at row 0 and ending at row $n_1 - m_1$.

13.3.2 Bird/Baker Algorithm

The algorithm designed independently by Bird and Baker for the two-dimensional pattern matching problem combines the use of the Aho–Corasick algorithm and the Knuth–Morris–Pratt (KMP) algorithm.

```
ZT(X, m1, m2, Y, n1, n2)
 1  ▷ Preprocessing
 2  ▷ Computes x
 3  for i2 ← 0 to m2 − 1
 4      do x[i2] ← 0
 5         for i1 ← 0 to m1 − 1
 6             do x[i2] ← (x[i2] << 1) + X[i1, i2]
 7  ▷ Computes the first value of y
 8  for j2 ← 0 to n2 − 1
 9      do y[j2] ← 0
10         for j1 ← 0 to m1 − 1
11             do y[j2] ← (y[j2] << 1) + Y[j1, j2]
12  d ← 1
13  for i ← 1 to m1 − 1
14      do d ← d << 1
15  next ← PREKMP(X′, m2)
16  ▷ Searching
17  j1 ← m1 − 1
18  while j1 < n1
19      do KMP-IN-LINE(X, m1, m2, Y, n1, n2, x, y, next, j2)
20         if j1 < n1 − 1
21           then for j2 ← 0 to n2 − 1
22                    do y[j2] ← REHASH(Y[j1 − m1 + 1, j2], Y[j1 + 1, j2], y[j2])
23         j1 ← j1 + 1
```

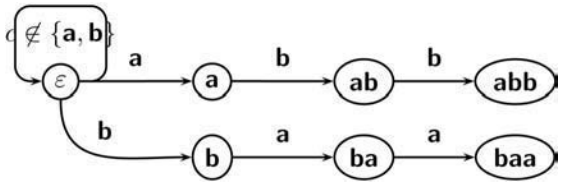
FIGURE 13.25 The Zhu-Takaoka two-dimensional pattern matching algorithm.

The pattern X is divided into its m_1 rows $R_0 = X[0, 0\,..\,m_2 - 1]$ to $R_{m_1-1} = x[m_1 - 1, 0\,..\,m_2 - 1]$. The rows are preprocessed into a trie as in the Aho–Corasick algorithm described earlier.

Example 13.10

Pattern X and the trie of its rows:

$$X = \begin{array}{|c|c|c|}\hline \mathbf{b} & \mathbf{a} & \mathbf{a} \\ \hline \mathbf{a} & \mathbf{b} & \mathbf{b} \\ \hline \mathbf{b} & \mathbf{a} & \mathbf{a} \\ \hline \end{array}$$



The search proceeds as follows. The text is read from the upper left corner to the bottom right corner, row by row. When reading the character $Y[j_1, j_2]$, the algorithm checks whether the portion $Y[j_1, j_2 - m_2 + 1\,..\,j_2] = R$ matches any of $R_0, \ldots, R_{m_1-1}$ using the Aho–Corasick machine. An additional one-dimensional array a of size n_1 is used as follows: $a[j_2] = k$ means that the $k - 1$ first rows $R_0, \ldots, R_{k-2}$ of the pattern match, respectively, the portions of the text: $Y[j_1 - k + 1, j_2 - m_2 + 1\,..\,j_2], \ldots, Y[j_1 - 1, j_2 - m_2 + 1\,..\,j_2]$. Then, if $R = R_{k-1}$, $a[j_2]$ is incremented to $k + 1$. If not, $a[j_2]$ is set to $s + 1$ where s is the maximum i such that

$$R_0 \cdots R_i = R_{k-s+1} \cdots R_{k-2} R.$$

```
PRE-KMP-FOR-B(X, m1, m2)
 1  i ← 0
 2  next[0] ← −1
 3  j ← −1
 4  while i < m1
 5      do while j > −1 and X[i, 0..m2 − 1] ≠ X[j, 0..m2 − 1]
 6             do j ← next[j]
 7         i ← i + 1
 8         j ← j + 1
 9         if X[i, 0..m2 − 1] ≠ X[j, 0..m2 − 1]
10           then next[i] ← next[j]
11           else next[i] ← j
12  return next
```

FIGURE 13.26 Computes the function *next* for rows of X.

```
B(X, m1, m2, Y, n1, n2)
 1  ▷ Preprocessing
 2  for i ← 0 to m2 − 1
 3      do a[i] ← 0
 4  root ← PREAC(m1)
 5  next ← PRE-KMP-FOR-B(X, m1, m2)
 6  for j1 ← 0 to n1 − 1
 7      do r ← root
 8         for j2 ← 0 to n2 − 1
 9             do while child(r, Y[j1, j2]) = UNDEFINED
10                    do r ← fail(r)
11                r ← child(r, Y[j1, j2])
12                if out(r) ≠ ∅
13                  then k ← a[j2]
14                       while k > 0 and X[k, 0..m2 − 1] = out(r)
15                             do k ← next[k]
16                       a[j2] ← k + 1
17                       if k ≥ m1 − 1
18                         then OUTPUT(j1 − m1 + 1, j2 − m2 + 1)
19                  else a[j2] ← 0
```

FIGURE 13.27 The Bird/Baker two-dimensional pattern matching algorithm.

The value s is computed using the KMP algorithm vertically (in columns). If there exists no such s, $a[j_2]$ is set to 0. Finally, if at some point $a[j_2] = m_1$, an occurrence of the pattern appears at position $(j_1 - m_1 + 1, j_2 - m_2 + 1)$ in the text.

The Bird/Baker algorithm is presented in Figure 13.26 and Figure 13.27. It runs in time $O((n_1 n_2 + m_1 m_2) \log \sigma)$.

13.4 Suffix Trees

The suffix tree $S(y)$ of a string y is a trie (as described earlier) containing all the suffixes of the string, and having the properties described subsequently. This data structure serves as an index on the string: it provides a direct access to all segments of the string, and gives the positions of all their occurrences in the string.

Once the suffix tree of a text y is built, searching for x in y remains to spell x along a branch of the tree. If this walk is successful, the positions of the pattern can be output. Otherwise, x does not occur in y.

```
SUFFIX-TREE(y, n)
1  T_{-1} ← one node tree
2  for j ← 0 to n − 1
3       do T_j ← INSERT(T_{j−1}, y[j . . n − 1])
4  return T_{n−1}
```

FIGURE 13.28 Construction of a suffix tree for y.

```
INSERT(T_{j−1}, y[j . . n − 1])
1  locate the node h associated with head_j in T_{j−1}, possibly breaking an edge
2  add a new edge labeled tail_j from h to a new leaf representing y[j . . n − 1]
3  return the modified tree
```

FIGURE 13.29 Insertion of a new suffix in the tree.

Any kind of trie that represents the suffixes of a string can be used to search it. But the suffix tree has additional features which imply that its size is linear. The suffix tree of y is defined by the following properties:

- All branches of $S(y)$ are labeled by all suffixes of y.
- Edges of $S(y)$ are labeled by strings.
- Internal nodes of $S(y)$ have at least two children (when y is not empty).
- Edges outgoing an internal node are labeled by segments starting with different letters.
- The preceding segments are represented by their starting positions on y and their lengths.

Moreover, it is assumed that y ends with a symbol occurring nowhere else in it (the dollar sign is used in examples). This avoids marking nodes, and implies that $S(y)$ has exactly n leaves (number of nonempty suffixes). The other properties then imply that the total size of $S(y)$ is $O(n)$, which makes it possible to design a linear-time construction of the trie. The algorithm described in the present section has this time complexity provided the alphabet is fixed, or with an additional multiplicative factor $\log \sigma$ otherwise.

The algorithm inserts all nonempty suffixes of y in the data structure from the longest to the shortest suffix, as shown in Figure 13.28. We introduce two definitions to explain how the algorithm works:

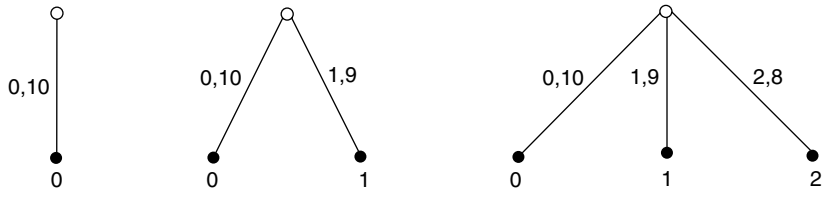
- $head_j$ is the longest prefix of $y[j \,..\, n-1]$ which is also a prefix of $y[i \,..\, n-1]$ for some $i < j$.
- $tail_j$ is the word such that $y[j \,..\, n-1] = head_j\, tail_j$.

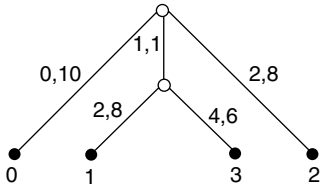
The strategy to insert the ith suffix in the tree is based on these definitions and described in Figure 13.29.

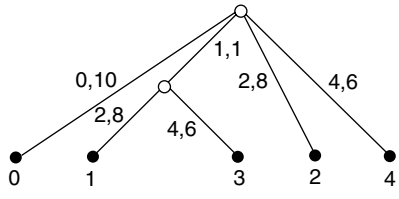
The second step of the insertion (Figure 13.29) is clearly performed in constant time. Thus, finding the node h is critical for the overall performance of the algorithm. A brute-force method to find it consists in spelling the current suffix $y[j \,..\, n-1]$ from the root of the tree, giving an $O(|head_j|)$ time complexity for the insertion at step j, and an $O(n^2)$ running time to build $S(y)$. Adding short-cut links leads to an overall $O(n)$ time complexity, although there is no guarantee that insertion at step j is realized in constant time.

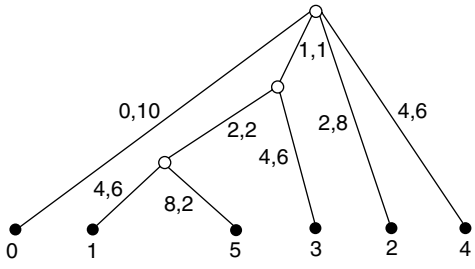
Example 13.11

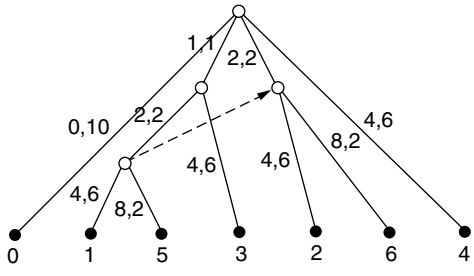
The different tries during the construction of the suffix tree of $y =$ **CAGATAGAG**. Leaves are black and labeled by the position of the suffix they represent. Plain arrows are labeled by pairs: the pair (j, ℓ) stands for the segment $y[j \,..\, j+\ell-1]$. Dashed arrows represent the nontrivial suffix links.

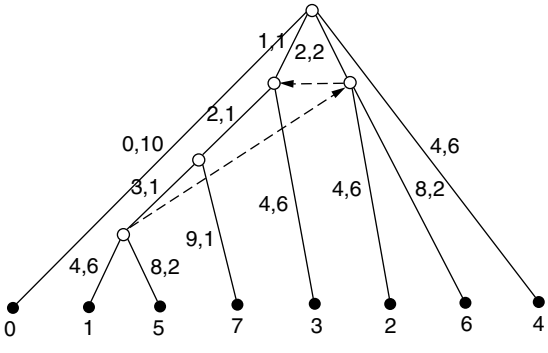


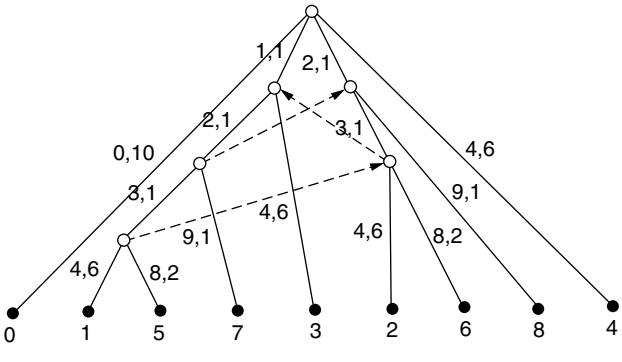


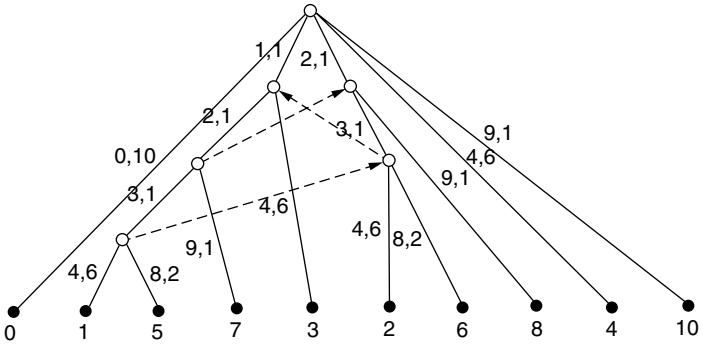












13.4.1 McCreight Algorithm

The key to get an efficient construction of the suffix tree $S(y)$ is to add links between nodes of the tree: they are called *suffix links*. Their definition relies on the relationship between $head_{j-1}$ and $head_j$: if $head_{j-1}$ is of the form az ($a \in \Sigma, z \in \Sigma^*$), then z is a prefix of $head_j$. In the suffix tree, the node associated with z is linked to the node associated with az. The suffix link creates a shortcut in the tree that helps with finding the next head efficiently. The insertion of the next suffix, namely, $head_j tail_j$, in the tree reduces to the insertion of $tail_j$ from the node associated with $head_j$.

The following property is an invariant of the construction: in T_j, only the node h associated with $head_j$ can fail to have a valid suffix link. This effectively happens when h has just been created at step j. The procedure to find the next head at step j is composed of two main phases:

A Rescanning: Assume that $head_{j-1} = az$ ($a \in \Sigma, z \in \Sigma^*$) and let d' be the associated node. If the suffix link on d' is defined, it leads to a node d from which the second step starts. Otherwise, the suffix link on d' is found by rescanning as follows. Let c' be the parent of d', and let (j, ℓ) be the label of edge (c', d'). For the ease of the description, assume that $az = av(y[j\,..\,j+\ell-1])$ (it may happen that $az = y[j\,..\,j+\ell-1]$). There is a suffix link defined on c' and going to some node c associated with v. The crucial observation here is that $y[j\,..\,j+\ell-1]$ is the prefix of the label of some branch starting at node c. Then, the algorithm rescans $y[j\,..\,j+\ell-1]$ in the tree: let e be the child of c along that branch, and let (k, m) be the label of edge (c, e). If $m < \ell$, then a recursive rescan of $q = y[j+m\,..\,j+\ell-1]$ starts from node e. If $m > \ell$, the edge (c, e) is broken to insert a new node d; labels are updated correspondingly. If $m = \ell$, d is simply set to e. If the suffix link of d' is currently undefined, it is set to d.

B Scanning: A downward search starts from d to find the node h associated with $head_j$. The search is dictated by the characters of $tail_{j-1}$ one at a time from left to right. If necessary a new internal node is created at the end of the scanning.

After the two phases A and B are executed, the node associated with the new head is known, and the tail of the current suffix can be inserted in the tree.

To analyze the time complexity of the entire algorithm we mainly have to evaluate the total time of all scannings, and the total time of all rescannings. We assume that the alphabet is fixed, so that branching from a node to one of its children can be implemented to take constant time. Thus, the time spent for all scannings is linear because each letter of y is scanned only once. The same holds true for rescannings because each step downward (through node e) increases strictly the position of the segment of y considered there, and this position never decreases.

An implementation of McCreight's algorithm is shown in Figure 13.30. The next figures (Figure 13.31 through Figure 13.34) give the procedures used by the algorithm, especially procedures RESCAN and SCAN.

We use the following notation:

- $parent(c)$ is the parent node of the node c
- $label(c)$ is the pair (i, l) if the edge from the parent node of c to c itself is associated with the factor $y[i\,..\,i+l-1]$
- $child(c, a)$ is the only node that can be reached from the node c with the character a
- $link(c)$ is the suffix node of the node c

13.5 Alignment

Alignments are used to compare strings. They are widely used in computational molecular biology. They constitute a mean to visualize resemblance between strings. They are based on notions of distance or similarity. Their computation is usually done by dynamic programming. A typical example of this method is the computation of the longest common subsequence of two strings. The reduction of the memory space presented on it can be applied to similar problems. We consider three different kinds of alignment of two

```
M(y, n)
 1  root ← INIT(y, n)
 2  head ← root
 3  tail ← child(root, y[0])
 4  n ← n − 1
 5  while n > 0
 6      do if head = root                          ▷ Phase A (rescanning)
 7           then d ← root
 8                (j, ℓ) ← label(tail)
 9                γ ← (j + 1, ℓ − 1)
10           else γ ← label(tail)
11                if link(head) ≠ UNDEFINED
12                  then d ← link(head)
13                  else (j, ℓ) ← label(head)
14                       if parent(head) = root
15                         then d ← RESCAN(root, j + 1, ℓ − 1))
16                         else d ← RESCAN(link(parent(head)), j, ℓ))
17                       link(head) ← d
18         (head, γ) ← SCAN(d, γ)                 ▷ Phase B (scanning)
19         create a new node tail
20         parent(tail) ← head
21         label(tail) ← γ
22         (j, ℓ) ← γ
23         child(head, y[ j]) ← tail
24         n ← n − 1
25  return root
```

FIGURE 13.30 Suffix tree construction.

```
INIT(y, n)
1  create a new node root
2  create a new node c
3  parent(root) ← UNDEFINED
4  parent(c) ← root
5  child(root, y[0]) ← c
6  label(root) ← UNDEFINED
7  label(c) ← (0, n)
8  return root
```

FIGURE 13.31 Initialization procedure.

```
RESCAN(c, j, ℓ)
1  (k, m) ← label(child(c, y[ j]))
2  while ℓ > 0 and ℓ ≥ m
3      do c ← child(c, y[ j])
4         ℓ ← ℓ − m
5         j ← j + m
6         (k, m) ← label(child(c, y[ j]))
7  if ℓ > 0
8    then return BREAK-EDGE(child(c, y[ j]), ℓ)
9    else return c
```

FIGURE 13.32 The crucial rescan operation.

```
BREAK-EDGE(c, k)
 1  create a new node g
 2  parent(g) ← parent(c)
 3  (j, ℓ) ← label(c)
 4  child(parent(c), y[j]) ← g
 5  label(g) ← (j, k)
 6  parent(c) ← g
 7  label(c) ← (j + k, ℓ − k)
 8  child(g, y[j + k]) ← c
 9  link(g) ← UNDEFINED
10  return g
```

FIGURE 13.33 Breaking an edge.

```
SCAN(d, γ)
 1  (j, ℓ) ← γ
 2  while child(d, y[j]) ≠ UNDEFINED
 3      do g ← child(d, y[j])
 4         k ← 1
 5         (s, lg) ← label(g)
 6         s ← s + 1
 7         ℓ ← ℓ − 1
 8         j ← j + 1
 9         while k < lg and y[j] = y[s]
10             do j ← j + 1
11                s ← s + 1
12                k ← k + 1
13                ℓ ← ℓ − 1
14         if k < lg
15           then return (BREAK-EDGE(g, k), (j, ℓ))
16         d ← g
17  return (d, (j, ℓ))
```

FIGURE 13.34 The scan operation.

strings x and y: global alignment (that consider the whole strings x and y), local alignment (that enable to find the segment of x that is closer to a segment of y), and the longest common subsequence of x and y.

An **alignment** of two strings x and y of length m and n, respectively, consists in aligning their symbols on vertical lines. Formally, an alignment of two strings $x, y \in \Sigma$ is a word w on the alphabet $(\Sigma \cup \{\varepsilon\}) \times (\Sigma \cup \{\varepsilon\}) \setminus (\{(\varepsilon, \varepsilon)\}$ (ε is the empty word) whose projection on the first component is x and whose projection of the second component is y.

Thus, an alignment $w = (\overline{x}_0, \overline{y}_0)(\overline{x}_1, \overline{y}_1) \cdots (\overline{x}_{p-1}, \overline{y}_{p-1})$ of length p is such that $x = \overline{x}_0\overline{x}_1 \cdots \overline{x}_{p-1}$ and $y = \overline{y}_0\overline{y}_1 \cdots \overline{y}_{p-1}$ with $\overline{x}_i \in \Sigma \cup \{\varepsilon\}$ and $\overline{y}_i \in \Sigma \cup \{\varepsilon\}$ for $0 \leq i \leq p - 1$. The alignment is represented as follows

$$\begin{array}{cccc} \overline{x}_0 & \overline{x}_1 & \cdots & \overline{x}_{p-1} \\ \overline{y}_0 & \overline{y}_1 & \cdots & \overline{y}_{p-1} \end{array}$$

with the symbol $-$ instead of the symbol ε.

Example 13.12

A	**C**	**G**	–	–	**A**
A	**T**	**G**	**C**	**T**	**A**

is an alignment of **ACGA** and **ATGCTA**.

13.5.1 Global alignment

A global alignment of two strings x and y can be obtained by computing the distance between x and y. The notion of distance between two strings is widely used to compare files. The **diff** command of Unix operating system implements an algorithm based on this notion, in which lines of the files are treated as symbols. The output of a comparison made by **diff** gives the minimum number of operations (substitute a symbol, insert a symbol, or delete a symbol) to transform one file into the other.

Let us define the edit distance between two strings x and y as follows: it is the minimum number of elementary edit operations that enable to transform x into y. The elementary edit operations are:

- The substitution of a character of x at a given position by a character of y
- The deletion of a character of x at a given position
- The insertion of a character of y in x at a given position

A cost is associated to each elementary edit operation. For $a, b \in \Sigma$:

- $Sub(a, b)$ denotes the cost of the substitution of the character a by the character b,
- $Del(a)$ denotes the cost of the deletion of the character a, and
- $Ins(a)$ denotes the cost of the insertion of the character a.

This means that the costs of the edit operations are independent of the positions where the operations occur. We can now define the edit distance of two strings x and y by

$$edit(x, y) = \min\{\text{cost of } \gamma \mid \gamma \in \Gamma_{x,y}\}$$

where $\Gamma_{x,y}$ is the set of all the sequences of edit operations that transform x into y, and the cost of an element $\gamma \in \Gamma_{x,y}$ is the sum of the costs of its elementary edit operations.

To compute $edit(x, y)$ for two strings x and y of length m and n, respectively, we make use of a two-dimensional table T of $m + 1$ rows and $n + 1$ columns such that

$$T[i, j] = edit(x[i], y[j])$$

for $i = 0, \ldots, m - 1$ and $j = 0, \ldots, n - 1$. It follows $edit(x, y) = T[m - 1, n - 1]$.

The values of the table T can be computed by the following recurrence formula:

$$\begin{aligned}
T[-1, -1] &= 0 \\
T[i, -1] &= T[i - 1, -1] + Del(x[i]) \\
T[-1, j] &= T[-1, j - 1] + Ins(y[j]) \\
T[i, j] &= \min \begin{cases} T[i - 1, j - 1] + Sub(x[i], y[j]) \\ T[i - 1, j] + Del(x[i]) \\ T[i, j - 1] + Ins(y[j]) \end{cases}
\end{aligned}$$

for $i = 0, 1, \ldots, m - 1$ and $j = 0, 1, \ldots, n - 1$.

```
GENERIC-DP(x, m, y, n, MARGIN, FORMULA)
1  MARGIN(T, x, m, y, n)
2  for j ← 0 to n − 1
3      do for i ← 0 to m − 1
4             do T[i, j] ← FORMULA(T, x, i, y, j)
5  return T
```

FIGURE 13.35 Computation of the table T by dynamic programming.

```
MARGIN-GLOBAL(T, x, m, y, n)
1  T[−1, −1] ← 0
2  for i ← 0 to m − 1
3      do T[i, −1] ← T[i − 1, −1] + Del(x[i])
4  for j ← 0 to n − 1
5      do T[−1, j] ← T[−1, j − 1] + Ins(y[j])
```

FIGURE 13.36 Margin initialization for the computation of a global alignment.

$$\text{FORMULA-GLOBAL}(T, x, i, y, j)$$
$$1 \quad \textbf{return} \min \begin{cases} T[i-1, j-1] + Sub(x[i], y[j]) \\ T[i-1, j] + Del(x[i]) \\ T[i, j-1] + Ins(y[j]) \end{cases}$$

FIGURE 13.37 Computation of $T[i, j]$ for a global alignment.

The value at position (i, j) in the table T only depends on the values at the three neighbor positions $(i-1, j-1)$, $(i-1, j)$, and $(i, j-1)$.

The direct application of the above recurrence formula gives an exponential time algorithm to compute $T[m-1, n-1]$. However, the whole table T can be computed in quadratic time technique known as "dynamic programming." This is a general technique that is used to solve the different kinds of alignments.

The computation of the table T proceeds in two steps. First it initializes the first column and first row of T; this is done by a call to a generic function MARGIN, which is a parameter of the algorithm and that depends on the kind of alignment considered. Second, it computes the remaining values of T, which is done by a call to a generic function FORMULA, which is a parameter of the algorithm and that depends on the kind of alignment considered. Computing a global alignment of x and y can be done by a call to GENERIC-DP with the following parameters $(x, m, y, n,$ MARGIN-GLOBAL, FORMULA-GLOBAL) (see Figure 13.35, Figure 13.36, and Figure 13.37). The computation of all the values of the table T can thus be done in quadratic space and time: $O(m \times n)$.

An optimal alignment (with minimal cost) can then be produced by a call to the function ONE-ALIGNMENT$(T, x, m-1, y, n-1)$ (see Figure 13.38). It consists in tracing back the computation of the values of the table T from position $[m-1, n-1]$ to position $[-1, -1]$. At each cell $[i, j]$, the algorithm determines among the three values $T[i-1, j-1]+Sub(x[i], y[j])$, $T[i-1, j]+Del(x[i])$, and $T[i, j-1]+Ins(y[j])$ which has been used to produce the value of $T[i, j]$. If $T[i-1, j-1] + Sub(x[i], y[j])$ has been used it adds $(x[i], y[j])$ to the optimal alignment and proceeds recursively with the cell at $[i-1, j-1]$. If $T[i-1, j] + Del(x[i])$ has been used, it adds $(x[i], -)$ to the optimal alignment and proceeds recursively with cell at $[i-1, j]$. If $T[i, j-1] + Ins(y[j])$ has been used, it adds $(-, y[j])$ to the optimal alignment

```
ONE-ALIGNMENT(T, x, i, y, j)
 1  if i = −1 and j = −1
 2    then return (ε, ε)
 3    else if i = −1
 4           then return ONE-ALIGNMENT(T, x, −1, y, j − 1) · (ε, y[j])
 5         elseif j = −1
 6           then return ONE-ALIGNMENT(T, x, i − 1, y, −1) · (x[i], ε)
 7         else    if T[i, j] = T[i − 1, j − 1] + Sub(x[i], y[j])
 8                   then return ONE-ALIGNMENT(T, x, i − 1, y, j − 1) · (x[i], y[j])
 9                 elseif T[i, j] = T[i − 1, j] + Del(x[i])
10                   then return ONE-ALIGNMENT(T, x, i − 1, y, j) · (x[i], ε)
11                 else    return ONE-ALIGNMENT(T, x, i, y, j − 1) · (ε, y[j])
```

FIGURE 13.38 Recovering an optimal alignment.

and proceeds recursively with cell at $[i, j-1]$. Recovering all the optimal alignments can be done by a similar technique.

Example 13.13

T	j	-1	0	1	2	3	4	5
i		$y[j]$	**A**	**T**	**G**	**C**	**T**	**A**
-1	$x[i]$	0	1	2	3	4	5	6
0	**A**	1	0	1	2	3	4	5
1	**C**	2	1	1	2	2	3	4
2	**G**	3	2	2	1	2	3	4
3	**A**	4	3	3	2	2	3	3

The values of the above table have been obtained with the following unitary costs: $Sub(a, b) = 1$ if $a \neq b$ and $Sub(a, a) = 0$, $Del(a) = Ins(a) = 1$ for $a, b \in \Sigma$.

13.5.2 Local Alignment

A local alignment of two strings x and y consists in finding the segment of x that is closer to a segment of y. The notion of distance used to compute global alignments cannot be used in that case because the segments of x closer to segments of y would only be the empty segment or individual characters. This is why a notion of similarity is used based on a scoring scheme for edit operations.

A score (instead of a cost) is associated to each elementary edit operation. For $a, b \in \Sigma$:

- $Sub_S(a, b)$ denotes the score of substituting the character b for the character a.
- $Del_S(a)$ denotes the score of deleting the character a.
- $Ins_S(a)$ denotes the score of inserting the character a.

This means that the scores of the edit operations are independent of the positions where the operations occur. For two characters a and b, a positive value of $Sub_S(a, b)$ means that the two characters are close to each other, and a negative value of $Sub_S(a, b)$ means that the two characters are far apart.

We can now define the edit score of two strings x and y by

$$sco(x, y) = \max\{\text{score of } \gamma \mid \gamma \in \Gamma_{x,y}\}$$

where $\Gamma_{x,y}$ is the set of all the sequences of edit operations that transform x into y and the score of an element $\sigma \in \Gamma_{x,y}$ is the sum of the scores of its elementary edit operations.

To compute $sco(x, y)$ for two strings x and y of length m and n, respectively, we make use of a two-dimensional table T of $m + 1$ rows and $n + 1$ columns such that

$$T[i, j] = sco(x[i], y[j])$$

for $i = 0, \ldots, m - 1$ and $j = 0, \ldots, n - 1$. Therefore, $sco(x, y) = T[m - 1, n - 1]$.

The values of the table T can be computed by the following recurrence formula:

$$\begin{aligned} T[-1, -1] &= 0\,, \\ T[i, -1] &= 0\,, \\ T[-1, j] &= 0\,, \\ T[i, j] &= \max \begin{cases} T[i-1, j-1] + Sub_S(x[i], y[j])\,, \\ T[i-1, j] + Del_S(x[i])\,, \\ T[i, j-1] + Ins_S(y[j])\,, \\ 0\,, \end{cases} \end{aligned}$$

for $i = 0, 1, \ldots, m - 1$ and $j = 0, 1, \ldots, n - 1$.

Computing the values of T for a local alignment of x and y can be done by a call to GENERIC-DP with the following parameters $(x, m, y, n,$ MARGIN-LOCAL, FORMULA-LOCAL$)$ in $O(mn)$ time and space complexity (see Figure 13.35, Figure 13.39, and Figure 13.40). Recovering a local alignment can be done in a way similar to what is done in the case of a global alignment (see Figure 13.38) but the trace back procedure must start at a position of a maximal value in T rather than at position $[m - 1, n - 1]$.

```
MARGIN-LOCAL(T, x, m, y, n)
1  T[−1, −1] ← 0
2  for i ← 0 to m − 1
3      do T[i, −1] ← 0
4  for j ← 0 to n − 1
5      do T[−1, j] ← 0
```

FIGURE 13.39 Margin initialization for computing a local alignment.

FORMULA-LOCAL(T, x, i, y, j)

$$\textbf{return} \max \begin{cases} T[i-1, j-1] + Sub_S(x[i], y[j]) \\ T[i-1, j] + Del_S(x[i]) \\ T[i, j-1] + Ins_S(y[j]) \\ 0 \end{cases}$$

FIGURE 13.40 Recurrence formula for computing a local alignment.

Example 13.14

Computation of an optimal local alignment of $x =$ **EAWACQGKL** and $y =$ **ERDAWCQPGKWY** with scores:

$Sub_S(a,a) = 1$, $Sub_S(a,b) = -3$ and $Del_S(a) = Ins_S(a) = -1$ for $a, b \in \Sigma$, $a \neq b$.

T	j	-1	0	1	2	3	4	5	6	7	8	9	10	11
i		$y[j]$	**E**	**R**	**D**	**A**	**W**	**C**	**Q**	**P**	**G**	**K**	**W**	**Y**
-1	$x[i]$	0	0	0	0	0	0	0	0	0	0	0	0	0
0	**E**	0	1	0	0	0	0	0	0	0	0	0	0	0
1	**A**	0	0	0	0	1	0	0	0	0	0	0	0	0
2	**W**	0	0	0	0	0	2	1	0	0	0	0	1	0
3	**A**	0	0	0	0	1	1	0	0	0	0	0	0	0
4	**C**	0	0	0	0	0	0	2	1	0	0	0	0	0
5	**Q**	0	0	0	0	0	0	1	3	2	1	0	0	0
6	**G**	0	0	0	0	0	0	0	2	1	3	2	1	0
7	**K**	0	0	0	0	0	0	0	1	0	2	4	3	2
8	**L**	0	0	0	0	0	0	0	0	0	1	3	2	1

The corresponding optimal local alignment is:

A W A C Q - G K
A W - C Q P G K

13.5.3 Longest Common Subsequence of Two Strings

A subsequence of a word x is obtained by deleting zero or more characters from x. More formally, $w[0\,..\,i-1]$ is a subsequence of $x[0\,..\,m-1]$ if there exists an increasing sequence of integers $(k_j \mid j = 0,\ldots,i-1)$ such that for $0 \leq j \leq i-1$, $w[j] = x[k_j]$. We say that a word is an $\mathrm{lcs}(x,y)$ if it is a **longest common subsequence** of the two words x and y. Note that two strings can have several longest common subsequences. Their common length is denoted by $\mathrm{llcs}(x,y)$.

A brute-force method to compute an $\mathrm{lcs}(x,y)$ would consist in computing all the subsequences of x, checking if they are subsequences of y, and keeping the longest one. The word x of length m has 2^m subsequences, and so this method could take $O(2^m)$ time, which is impractical even for fairly small values of m.

However, $\mathrm{llcs}(x,y)$ can be computed with a two-dimensional table T by the following recurrence formula:

$$
\begin{aligned}
T[-1,-1] &= 0\,,\\
T[i,-1] &= 0\,,\\
T[-1,j] &= 0\,,\\
T[i,j] &= \begin{cases} T[i-1,j-1]+1 & \text{if } x[i] = y[j],\\ \max(T[i-1,j], T[i,j-1]) & \text{otherwise,}\end{cases}
\end{aligned}
$$

for $i = 0,1,\ldots,m-1$ and $j = 0,1,\ldots,n-1$. Then, $T[i,j] = \mathrm{llcs}(x[0\,..\,i], y[0\,..\,j])$ and $\mathrm{llcs}(x,y) = T[m-1,n-1]$.

Computing $T[m-1,n-1]$ can be done by a call to GENERIC-DP with the following parameters $(x, m, y, n,$ MARGIN-LOCAL, FORMULA-LCS$)$ in $O(mn)$ time and space complexity (see Figure 13.35, Figure 13.39, and Figure 13.41).

```
FORMULA-LCS(T, x, i, y, j)
1  if x[i] = y[j]
2     then return T[i − 1, j − 1] + 1
3     else return max{T[i − 1, j], T[i, j − 1]}
```

FIGURE 13.41 Recurrence formula for computing an *lcs*.

It is possible afterward to trace back a path from position $[m-1, n-1]$ in order to exhibit an $\text{lcs}(x, y)$ in a similar way as for producing a global alignment (see Figure 13.38).

Example 13.15

The value $T[4, 8] = 4$ is $\text{llcs}(x, y)$ for $x =$ **AGCGA** and $y =$ **CAGATAGAG**. String **AGGA** is an lcs of x and y.

T	j	-1	0	1	2	3	4	5	6	7	8
i		$y[j]$	**C**	**A**	**G**	**A**	**T**	**A**	**G**	**A**	**G**
-1	$x[i]$	0	0	0	0	0	0	0	0	0	0
0	**A**	0	0	1	1	1	1	1	1	1	1
1	**G**	0	0	1	2	2	2	2	2	2	2
2	**C**	0	1	1	2	2	2	2	2	2	2
3	**G**	0	1	1	2	2	2	2	3	3	3
4	**A**	0	1	2	2	3	3	3	3	4	4

13.5.4 Reducing the Space: Hirschberg Algorithm

If only the length of an $\text{lcs}(x, y)$ is required, it is easy to see that only one row (or one column) of the table T needs to be stored during the computation. The space complexity becomes $O(\min(m, n))$, as can be checked on the algorithm of Figure 13.42. Indeed, the Hirschberg algorithm computes an $\text{lcs}(x, y)$ in linear space and not only the value $\text{llcs}(x, y)$. The computation uses the algorithm of Figure 13.43.

Let us define

$$T^*[i, n] = T^*[m, j] = 0, \quad \text{for } 0 \le i \le m \quad \text{and} \quad 0 \le j \le n$$

$$T^*[m-i, n-j] = \text{llcs}((x[i\,..\,m-1])^R, (y[j\,..\,n-1])^R)$$
$$\text{for } 0 \le i \le m-1 \quad \text{and} \quad 0 \le j \le n-1$$

and

$$M(i) = \max_{0 \le j < n} \{T[i, j] + T^*[m-i, n-j]\}$$

where the word w^R is the reverse (or mirror image) of the word w. The following property is the key observation to compute an $\text{lcs}(x, y)$ in linear space:

$$M(i) = T[m-1, n-1], \quad \text{for } 0 \le i < m\,.$$

In the algorithm shown in Figure 13.43, the integer j is chosen as $n/2$. After $T[i, j-1]$ and $T^*[m-i, n-j]$ $(0 \le i < m)$ are computed, the algorithm finds an integer k such that $T[i, k] + T^*[m-i, n-k] = T[m-1, n-1]$. Then, recursively, it computes an $\text{lcs}(x[0\,..\,k-1], y[0\,..\,j-1])$ and an $\text{lcs}(x[k\,..\,m-1], y[j\,..\,n-1])$, and concatenates them to get an $\text{lcs}(x, y)$.

```
LLCS(x, m, y, n)
 1  for i ← −1 to m − 1
 2      do C[i] ← 0
 3  for j ← 0 to n − 1
 4      do last ← 0
 5          for i ← −1 to m − 1
 6              do if last > C[i]
 7                    then C[i] ← last
 8                  elseif last < C[i]
 9                    then last ← C[i]
10                  elseif x[i] = y[j]
11                    then C[i] ← C[i] + 1
12                         last ← last + 1
13  return C
```

FIGURE 13.42 $O(m)$-space algorithm to compute llcs(x, y).

```
HIRSCHBERG(x, m, y, n)
 1  if m = 0
 2    then return ε
 3    else if m = 1
 4           then if x[0] ∈ y
 5                  then return x[0]
 6                  else return ε
 7           else j ← ⌊n/2⌋
 8                C ← LLCS(x, m, y[0 . . j − 1], j)
 9                C* ← LLCS(x^R, m, y[j . . n − 1]^R, n − j)
10                k ← m − 1
11                M ← C[m − 1] + C*[m − 1]
12                for j ← −1 to m − 2
13                    do if C[j] + C*[j] > M
14                          then M ← C[j] + C*[j]
15                               k ← j
16                return HIRSCHBERG(x[0 . . k − 1], k, y[0 . . j − 1], j)·
                         HIRSCHBERG(x[k . . m − 1], m − k, y[j . . n − 1], n − j)
```

FIGURE 13.43 $O(\min(m, n))$-space computation of lcs(x, y).

The running time of the Hirschberg algorithm is still $O(mn)$ but the amount of space required for the computation becomes $O(\min(m, n))$, instead of being quadratic when computed by dynamic programming.

13.6 Approximate String Matching

Approximate string matching is the problem of finding all approximate occurrences of a pattern x of length m in a text y of length n. Approximate occurrences of x are segments of y that are close to x according to a specific distance: the distance between segments and x must be not greater than a given integer k. We consider two distances in this section: the **Hamming distance** and the **Levenshtein distance**.

With the Hamming distance, the problem is also known as approximate string matching with k mismatches. With the Levenshtein distance (or edit distance), the problem is known as approximate string matching with k differences.

The Hamming distance between two words w_1 and w_2 of the same length is the number of positions with different characters. The Levenshtein distance between two words w_1 and w_2 (not necessarily of the same length) is the minimal number of differences between the two words. A difference is one of the following operations:

- A substitution: a character of w_1 corresponds to a different character in w_2.
- An insertion: a character of w_1 corresponds to no character in w_2.
- A deletion: a character of w_2 corresponds to no character in w_1.

The *Shift-Or algorithm* of the next section is a method that is both very fast in practice and very easy to implement. It solves the Hamming distance and the Levenshtein distance problems. We initially describe the method for the exact string-matching problem and then show how it can handle the cases of k mismatches and k differences. The method is flexible enough to be adapted to a wide range of similar approximate matching problems.

13.6.1 Shift-Or Algorithm

We first present an algorithm to solve the exact string-matching problem using a technique different from those developed previously, but which extends readily to the approximate string-matching problem.

Let $\mathbf{R}^0$ be a bit array of size m. Vector $\mathbf{R}^0_j$ is the value of the entire array $\mathbf{R}^0$ after text character $y[j]$ has been processed (see Figure 13.44). It contains information about all matches of prefixes of x that end at position j in the text. It is defined, for $0 \leq i \leq m-1$, by

$$\mathbf{R}^0_j[i] = \begin{cases} 0 & \text{if } x[0..i] = y[j-i..j] \\ 1 & \text{otherwise.} \end{cases}$$

Therefore, $\mathbf{R}^0_j[m-1] = 0$ is equivalent to saying that an (exact) occurrence of the pattern x ends at position j in y.

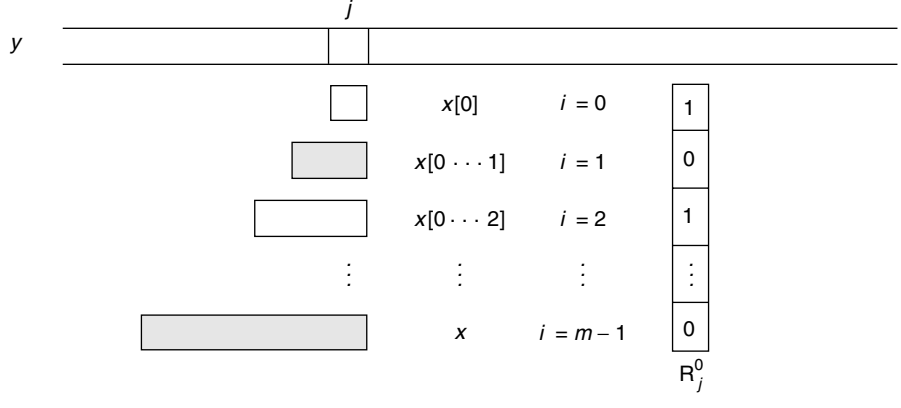


FIGURE 13.44 Meaning of vector $\mathbf{R}^0_j$.

The vector $\mathbf{R}_j^0$ can be computed after $\mathbf{R}_{j-1}^0$ by the following recurrence relation:

$$\mathbf{R}_j^0[i] = \begin{cases} 0 & \text{if } \mathbf{R}_{j-1}^0[i-1] = 0 \text{ and } x[i] = y[j], \\ 1 & \text{otherwise,} \end{cases}$$

and

$$\mathbf{R}_j^0[0] = \begin{cases} 0 & \text{if } x[0] = y[j], \\ 1 & \text{otherwise.} \end{cases}$$

The transition from $\mathbf{R}_{j-1}^0$ to $\mathbf{R}_j^0$ can be computed very fast as follows. For each $a \in \Sigma$, let S_a be a bit array of size m defined, for $0 \leq i \leq m-1$, by

$$S_a[i] = 0 \quad \text{if} \quad x[i] = a.$$

The array S_a denotes the positions of the character a in the pattern x. All arrays S_a are preprocessed before the search starts. And the computation of $\mathbf{R}_j^0$ reduces to two operations, SHIFT and OR:

$$\mathbf{R}_j^0 = \text{SHIFT}(\mathbf{R}_{j-1}^0) \quad \text{OR} \quad S_{y[j]}.$$

Example 13.16

String $x =$ **GATAA** occurs at position 2 in $y =$ **CAGATAAGAGAA**.

$S_{\mathbf{A}}$	$S_{\mathbf{C}}$	$S_{\mathbf{G}}$	$S_{\mathbf{T}}$
1	1	0	1
0	1	1	1
1	1	1	0
0	1	1	1
0	1	1	1

	C	**A**	**G**	**A**	**T**	**A**	**A**	**G**	**A**	**G**	**A**	**A**
G	1	1	0	1	1	1	1	0	1	0	1	1
A	1	1	1	0	1	1	1	1	0	1	0	1
T	1	1	1	1	0	1	1	1	1	1	1	1
A	1	1	1	1	1	0	1	1	1	1	1	1
A	1	1	1	1	1	1	0	1	1	1	1	1

13.6.2 String Matching with *k* Mismatches

The Shift-Or algorithm easily adapts to support approximate string matching with k mismatches. To simplify the description, we shall present the case where at most one substitution is allowed.

We use arrays $\mathbf{R}^0$ and S as before, and an additional bit array $\mathbf{R}^1$ of size m. Vector $\mathbf{R}_{j-1}^1$ indicates all matches with at most one substitution up to the text character $y[j-1]$. The recurrence on which the computation is based splits into two cases.

1. There is an exact match on the first i characters of x up to $y[j-1]$ (i.e., $\mathbf{R}_{j-1}^0[i-1] = 0$). Then, substituting $x[i]$ to $y[j]$ creates a match with one substitution (see Figure 13.45). Thus,

$$\mathbf{R}_j^1[i] = \mathbf{R}_{j-1}^0[i-1].$$

2. There is a match with one substitution on the first i characters of x up to $y[j-1]$ and $x[i] = y[j]$. Then, there is a match with one substitution of the first $i+1$ characters of x up to $y[j]$

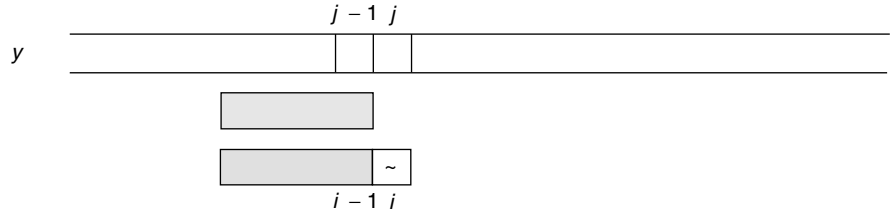


FIGURE 13.45 If $\mathbf{R}^0_{j-1}[i-1] = 0$, then $\mathbf{R}^1_j[i] = 0$.

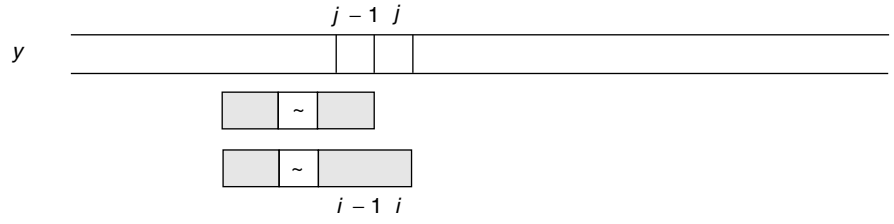


FIGURE 13.46 $\mathbf{R}^1_j[i] = \mathbf{R}^1_{j-1}[i-1]$ if $x[i] = y[j]$.

(see Figure 13.46). Thus,

$$\mathbf{R}^1_j[i] = \begin{cases} \mathbf{R}^1_{j-1}[i-1] & \text{if } x[i] = y[j], \\ 1 & \text{otherwise.} \end{cases}$$

This implies that $\mathbf{R}^1_j$ can be updated from $\mathbf{R}^1_{j-1}$ by the relation:

$$\mathbf{R}^1_j = \left(\text{SHIFT}\left(\mathbf{R}^1_{j-1}\right) \quad \text{OR} \quad S_{y[j]}\right) \quad \text{AND} \quad \text{SHIFT}\left(\mathbf{R}^0_{j-1}\right).$$

Example 13.17

String $x =$ **GATAA** occurs at positions 2 and 7 in $y =$ **CAGATAAGAGAA** with no more than one mismatch.

	C	**A**	**G**	**A**	**T**	**A**	**A**	**G**	**A**	**G**	**A**	**A**
G	0	0	0	0	0	0	0	0	0	0	0	0
A	1	0	1	0	1	0	0	1	0	1	0	0
T	1	1	1	1	0	1	1	1	1	0	1	0
A	1	1	1	1	1	0	1	1	1	1	0	1
A	1	1	1	1	1	1	0	1	1	1	1	0

13.6.3 String Matching with *k* Differences

We show in this section how to adapt the Shift-Or algorithm to the case of only one insertion, and then dually to the case of only one deletion. The method is based on the following elements.

One insertion is allowed: here, vector $\mathbf{R}^1_{j-1}$ indicates all matches with at most one insertion up to text character $y[j-1]$. $\mathbf{R}^1_{j-1}[i-1] = 0$ if the first i characters of x ($x[0\,..\,i-1]$) match i symbols of the last $i+1$ text characters up to $y[j-1]$. Array $\mathbf{R}^0$ is maintained as before, and we show how to maintain array $\mathbf{R}^1$. Two cases arise.

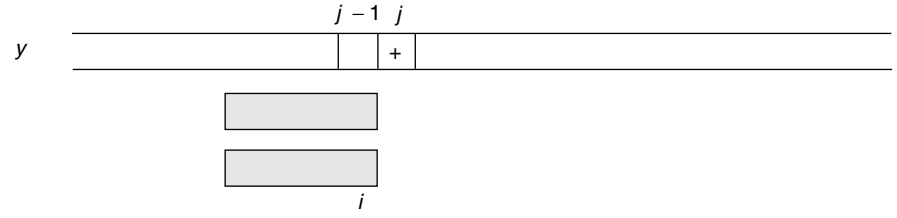


FIGURE 13.47 If $\mathbf{R}^0_{j-1}[i] = 0$, then $\mathbf{R}^1_j[i] = 0$.

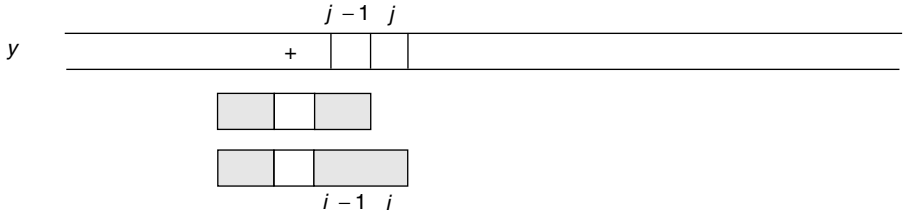


FIGURE 13.48 $\mathbf{R}^1_j[i] = \mathbf{R}^1_{j-1}[i-1]$ if $x[i] = y[j]$.

1. There is an exact match on the first $i+1$ characters of x ($x[0\,..\,i]$) up to $y[j-1]$. Then inserting $y[j]$ creates a match with one insertion up to $y[j]$ (see Figure 13.47). Thus,

$$\mathbf{R}^1_j[i] = \mathbf{R}^0_{j-1}[i]\,.$$

2. There is a match with one insertion on the i first characters of x up to $y[j-1]$. Then if $x[i] = y[j]$, there is a match with one insertion on the first $i+1$ characters of x up to $y[j]$ (see Figure 13.48). Thus,

$$\mathbf{R}^1_j[i] = \begin{cases} \mathbf{R}^1_{j-1}[i-1] & \text{if } x[i] = y[j], \\ 1 & \text{otherwise.} \end{cases}$$

This shows that $\mathbf{R}^1_j$ can be updated from $\mathbf{R}^1_{j-1}$ with the formula

$$\mathbf{R}^1_j = \left(\text{SHIFT}\left(\mathbf{R}^1_{j-1}\right) \quad \text{OR} \quad S_{y[j]}\right) \quad \text{AND} \quad \mathbf{R}^0_{j-1}\,.$$

Example 13.18

Here, **GATAAG** is an occurrence of $x =$ **GATAA** with exactly one insertion in $y =$ **CAGATAAGAGAA**

	C	**A**	**G**	**A**	**T**	**A**	**A**	**G**	**A**	**G**	**A**	**A**
G	1	1	1	0	1	1	1	1	0	1	0	1
A	1	1	1	1	0	1	1	1	1	0	1	0
T	1	1	1	1	1	0	1	1	1	1	1	1
A	1	1	1	1	1	1	0	1	1	1	1	1
A	1	1	1	1	1	1	1	0	1	1	1	1

One deletion is allowed: we assume here that $\mathbf{R}^1_{j-1}$ indicates all possible matches with at most one deletion up to $y[j-1]$. As in the previous solution, two cases arise.

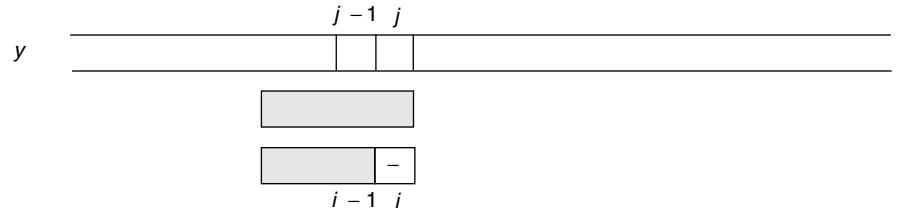


FIGURE 13.49 If $\mathbf{R}_j^0[i] = 0$, then $\mathbf{R}_j^1[i] = 0$.

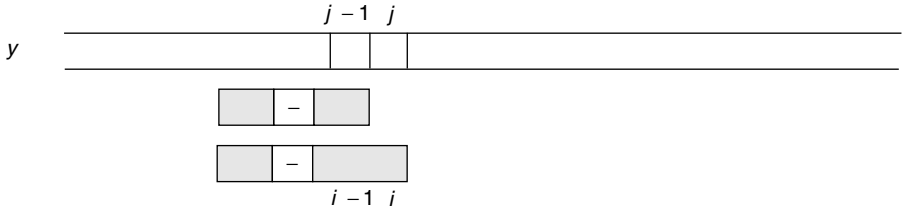


FIGURE 13.50 $\mathbf{R}_j^1[i] = \mathbf{R}_{j-1}^1[i-1]$ if $x[i] = y[j]$.

1. There is an exact match on the first $i+1$ characters of x ($x[0\,..\,i]$) up to $y[j]$ (i.e., $\mathbf{R}_j^0[i] = 0$). Then, deleting $x[i]$ creates a match with one deletion (see Figure 13.49). Thus,

$$\mathbf{R}_j^1[i] = \mathbf{R}_j^0[i].$$

2. There is a match with one deletion on the first i characters of x up to $y[j-1]$ and $x[i] = y[j]$. Then, there is a match with one deletion on the first $i+1$ characters of x up to $y[j]$ (see Figure 13.50). Thus,

$$\mathbf{R}_j^1[i] = \begin{cases} \mathbf{R}_{j-1}^1[i-1] & \text{if } x[i] = y[j], \\ 1 & \text{otherwise.} \end{cases}$$

The discussion provides the following formula used to update $\mathbf{R}_j^1$ from $\mathbf{R}_{j-1}^1$:

$$\mathbf{R}_j^1 = \left(\text{SHIFT}\left(\mathbf{R}_{j-1}^1\right) \quad \text{OR} \quad S_{y[j]}\right) \quad \text{AND} \quad \text{SHIFT}\left(\mathbf{R}_j^0\right).$$

Example 13.19

GATA and **ATAA** are two occurrences with one deletion of $x =$ **GATAA** in $y =$ **CAGATAAGAGAA**

	C	**A**	**G**	**A**	**T**	**A**	**A**	**G**	**A**	**G**	**A**	**A**
G	0	0	0	0	0	0	0	0	0	0	0	0
A	1	0	0	0	1	0	0	0	0	0	0	0
T	1	1	1	0	0	1	1	1	0	1	0	1
A	1	1	1	1	0	0	1	1	1	1	1	0
A	1	1	1	1	1	0	0	1	1	1	1	1

13.6.4 Wu–Manber Algorithm

We present in this section a general solution for the approximate string-matching problem with at most k differences of the types: insertion, deletion, and substitution. It is an extension of the problems presented

above. The following algorithm maintains $k+1$ bit arrays $\mathbf{R}^0, \mathbf{R}^1, \ldots, \mathbf{R}^k$ that are described now. The vector $\mathbf{R}^0$ is maintained similarly as in the exact matching case (Section 13.6.1). The other vectors are computed with the formula $(1 \leq \ell \leq k)$

$$\begin{aligned}\mathbf{R}_j^\ell = &\left(\text{SHIFT}\left(\mathbf{R}_{j-1}^\ell\right) \quad \text{OR} \quad S_{y[j]}\right)\\ &\text{AND} \quad \text{SHIFT}\left(\mathbf{R}_j^{\ell-1}\right)\\ &\text{AND} \quad \text{SHIFT}\left(\mathbf{R}_{j-1}^{\ell-1}\right)\\ &\text{AND} \quad \mathbf{R}_{j-1}^{\ell-1}\end{aligned}$$

which can be rewritten into

$$\begin{aligned}\mathbf{R}_j^\ell = &\left(\text{SHIFT}\left(\mathbf{R}_{j-1}^\ell\right) \quad \text{OR} \quad S_{y[j]}\right)\\ &\text{AND} \quad \text{SHIFT}\left(\mathbf{R}_j^{\ell-1} \quad \text{AND} \quad \mathbf{R}_{j-1}^{\ell-1}\right)\\ &\text{AND} \quad \mathbf{R}_{j-1}^{\ell-1}.\end{aligned}$$

Example 13.20

Here, $x =$ **GATAA** and $y =$ **CAGATAAGAGAA** and $k = 1$. The output 5, 6, 7, and 11 corresponds to the segments **GATA**, **GATAA**, **GATAAG**, and **GAGAA**, which approximate the pattern **GATAA** with no more than one difference.

	C	**A**	**G**	**A**	**T**	**A**	**A**	**G**	**A**	**G**	**A**	**A**
G	0	0	0	0	0	0	0	0	0	0	0	0
A	1	0	0	0	0	0	0	0	0	0	0	0
T	1	1	1	0	0	0	1	1	0	0	0	0
A	1	1	1	1	0	0	0	1	1	1	0	0
A	1	1	1	1	1	0	0	0	1	1	1	0

The method, called the Wu–Manber algorithm, is implemented in Figure 13.51. It assumes that the length of the pattern is no more than the size of the memory word of the machine, which is often the case in applications.

```
WM(x, m, y, n, k)
 1  for each character a ∈ Σ
 2      do S_a ← 1^m
 3  for i ← 0 to m − 1
 4      do S_{x[i]}[i] ← 0
 5  R^0 ← 1^m
 6  for ℓ ← 1 to k
 7      do R^ℓ ← SHIFT(R^{ℓ−1})
 8  for j ← 0 to n − 1
 9      do T ← R^0
10         R^0 ← SHIFT(R^0)  OR  S_{y[j]}
11         for ℓ ← 1 to k
12             do T' ← R^ℓ
13                R^ℓ ← (SHIFT(R^ℓ)  OR  S_{y[j]})  AND  (SHIFT((T  AND  R^{ℓ−1}))  AND  T
14                T ← T'
15         if R^k[m − 1] = 0
16           then OUTPUT(j)
```

FIGURE 13.51 Wu–Manber approximate string-matching algorithm.

The preprocessing phase of the algorithm takes $O(\sigma m + km)$ memory space, and runs in time $O(\sigma m + k)$. The time complexity of its searching phase is $O(kn)$.

13.7 Text Compression

In this section we are interested in algorithms that compress texts. Compression serves both to save storage space and to save transmission time. We shall assume that the uncompressed text is stored in a file. The aim of compression algorithms is to produce another file containing the compressed version of the same text. Methods in this section work with no loss of information, so that decompressing the compressed text restores exactly the original text.

We apply two main strategies to design the algorithms. The first strategy is a statistical method that takes into account the frequencies of symbols to build a uniquely decipherable code optimal with respect to the compression. The code contains new codewords for the symbols occurring in the text. In this method, fixed-length blocks of bits are encoded by different codewords. *A contrario*, the second strategy encodes variable-length segments of the text. To put it simply, the algorithm, while scanning the text, replaces some already read segments just by a pointer to their first occurrences.

Text compression software often use a mixture of several methods. An example of that is given in Section 13.7.3, which contains in particular two classical simple compression algorithms. They compress efficiently only a small variety of texts when used alone, but they become more powerful with the special preprocessing presented there.

13.7.1 Huffman Coding

The Huffman method is an optimal statistical coding. It transforms the original code used for characters of the text (ASCII code on 8 b, for instance). Coding the text is just replacing each symbol (more exactly, each occurrence of it) by its new codeword. The method works for any length of blocks (not only 8 b), but the running time grows exponentially with the length. In the following, we assume that symbols are originally encoded on 8 b to simplify the description.

The Huffman algorithm uses the notion of **prefix code**. A prefix code is a set of words containing no word that is a prefix of another word of the set. The advantage of such a code is that decoding is immediate. Moreover, it can be proved that this type of code does not weaken the compression.

A prefix code on the binary alphabet $\{0, 1\}$ can be represented by a trie (see section on the Aho–Corasick algorithm) that is a binary tree. In the present method codes are complete: they correspond to complete tries (internal nodes have exactly two children). The leaves are labeled by the original characters, edges are labeled by 0 or 1, and labels of branches are the words of the code. The condition on the code implies that codewords are identified with leaves only. We adopt the convention that, from an internal node, the edge to its left child is labeled by 0, and the edge to its right child is labeled by 1.

In the model where characters of the text are given new codewords, the Huffman algorithm builds a code that is optimal in the sense that the compression is the best possible (the length of the compressed text is minimum). The code depends on the text, and more precisely on the frequencies of each character in the uncompressed text. The more frequent characters are given short codewords, whereas the less frequent symbols have longer codewords.

13.7.1.1 Encoding

The coding algorithm is composed of three steps: count of character frequencies, construction of the prefix code, and encoding of the text.

The first step consists in counting the number of occurrences of each character in the original text (see Figure 13.52). We use a special end marker (denoted by END), which (virtually) appears only once at the end of the text. It is possible to skip this first step if fixed statistics on the alphabet are used. In this case, the method is optimal according to the statistics, but not necessarily for the specific text.

```
COUNT(fin)
1  for each character a ∈ Σ
2      do freq(a) ← 0
3  while not end of file fin and a is the next symbol
4      do freq(a) ← freq(a) + 1
5  freq(END) ← 1
```

FIGURE 13.52 Counts the character frequencies.

```
BUILD-TREE()
 1  for each character a ∈ Σ ∪ {END}
 2      do if freq(a) ≠ 0
 3           then create a new node t
 4                weight(t) ← freq(a)
 5                label(t) ← a
 6  lleaves ← list of all the nodes in increasing order of weight
 7  ltrees ← empty list
 8  while LENGTH(lleaves) + LENGTH(ltrees) > 1
 9      do (ℓ, r) ← extract the two nodes of smallest weight (among the two nodes at the
                    beginning of lleaves and the two nodes at the beginning of ltrees)
10         create a new node t
11         weight(t) ← weight(ℓ) + weight(r)
12         left(t) ← ℓ
13         right(t) ← r
14         insert t at the end of ltrees
15  return t
```

FIGURE 13.53 Builds the coding tree.

The second step of the algorithm builds the tree of a prefix code using the character frequency $freq(a)$ of each character a in the following way:

- Create a one-node tree t for each character a, setting $weight(t) = freq(a)$ and $label(t) = a$,
- Repeat (1), extract the two least weighted trees t_1 and t_2, and (2) create a new tree t_3 having left subtree t_1, right subtree t_2, and weight $weight(t_3) = weight(t_1) + weight(t_2)$,
- Until only one tree remains.

The tree is constructed by the algorithm BUILD-TREE in Figure 13.53. The implementation uses two linear lists. The first list contains the leaves of the future tree, each associated with a symbol. The list is sorted in the increasing order of the weight of the leaves (frequency of symbols). The second list contains the newly created trees. Extracting the two least weighted trees consists in extracting the two least weighted trees among the two first trees of the list of leaves and the two first trees of the list of created trees. Each new tree is inserted at the end of the list of the trees. The only tree remaining at the end of the procedure is the coding tree.

After the coding tree is built, it is possible to recover the codewords associated with characters by a simple depth-first search of the tree (see Figure 13.54); $codeword(a)$ is then the binary code associated with the character a.

```
BUILD-CODE(t, length)
1  if t is not a leaf
2    then temp[length] ← 0
3         BUILD-CODE(left(t), length + 1)
4         temp[length] ← 1
5         BUILD-CODE(right(t), length + 1)
6    else codeword(label(t)) ← temp[0 . . length − 1]
```

FIGURE 13.54 Builds the character codes from the coding tree.

```
CODE-TREE(fout, t)
1  if t is not a leaf
2    then write a 0 in the file fout
3         CODE-TREE(fout, left(t))
4         CODE-TREE(fout, right(t))
5    else write a 1 in the file fout
6         write the original code of label(t) in the file fout
```

FIGURE 13.55 Memorizes the coding tree in the compressed file.

```
CODE-TEXT(fin, fout)
1  while not end of file fin and a is the next symbol
2      do write codeword(a) in the file fout
3  write codeword(END) in the file fout
```

FIGURE 13.56 Encodes the characters in the compressed file.

```
CODING(fin, fout)
1  COUNT(fin)
2  t ← BUILD-TREE()
3  BUILD-CODE(t, 0)
4  CODE-TREE(fout, t)
5  CODE-TEXT(fin, fout)
```

FIGURE 13.57 Complete function for Huffman coding.

In the third step, the original text is encoded. Since the code depends on the original text, in order to be able to decode the compressed text, the coding tree and the original codewords of symbols must be stored with the compressed text. This information is placed in a header of the compressed file, to be read at decoding time just before the compressed text. The header is made via a depth-first traversal of the tree. Each time an internal node is encountered, a 0 is produced. When a leaf is encountered, a 1 is produced, followed by the original code of the corresponding character on 9 b (so that the end marker can be equal to 256 if all the characters appear in the original text). This part of the encoding algorithm is shown in Figure 13.55. After the header of the compressed file is computed, the encoding of the original text is realized by the algorithm of Figure 13.56.

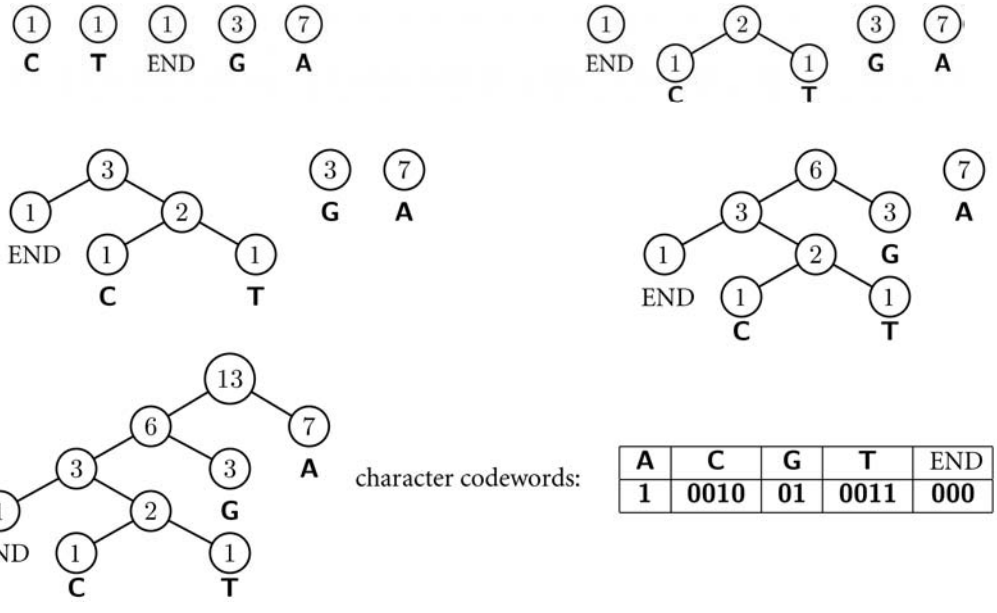
A complete implementation of the Huffman algorithm, composed of the three steps just described, is given in Figure 13.57.

Example 13.21

Here, $y =$ **CAGATAAGAGAA**. The length of $y = 12 \times 8 = 96$ b (assuming an 8-b code). The character frequencies are

A	**C**	**G**	**T**	END
7	1	3	1	1

The different steps during the construction of the coding tree are



character codewords:

A	**C**	**G**	**T**	END
1	**0010**	**01**	**0011**	**000**

The encoded tree is **0001** binary (END, 9)**01**binary (**C**, 9)**1**binary(**T**, 9) **1**binary (**G**, 9)**1**binary (**A**, 9), which produces a header of length 54 b,

0001 100000000 01 001000011 1 001010100 1 001000111 1 001000001

The encoded text

0010 1 01 1 0011 1 1 01 1 01 1 1 000

is of length 24 b. The total length of the compressed file is 78 b.

The construction of the tree takes $O(\sigma \log \sigma)$ time if the sorting of the list of the leaves is implemented efficiently. The rest of the encoding process runs in linear time in the sum of the sizes of the original and compressed texts.

13.7.1.2 Decoding

Decoding a file containing a text compressed by the Huffman algorithm is a mere programming exercise. First, the coding tree is rebuilt by the algorithm of Figure 13.58. Then, the uncompressed text is recovered by parsing the compressed text with the coding tree. The process begins at the root of the coding tree and follows a left edge when a 0 is read or a right edge when a 1 is read. When a leaf is encountered, the corresponding character (in fact the original codeword of it) is produced and the parsing phase resumes at the root of the tree. The parsing ends when the codeword of the end marker is read. An implementation of the decoding of the text is presented in Figure 13.59.

```
REBUILD-TREE(fin, t)
 1  b ← read a bit from the file fin
 2  if b = 1                          ▷ leaf
 3    then left(t) ← NIL
 4         right(t) ← NIL
 5         label(t) ← symbol corresponding to the 9 next bits in the file fin
 6    else create a new node ℓ
 7         left(t) ← ℓ
 8         REBUILD-TREE(fin, ℓ)
 9         create a new node r
10         right(t) ← r
11         REBUILD-TREE(fin, r)
```

FIGURE 13.58 Rebuilds the tree read from the compressed file.

```
DECODE-TEXT(fin, fout, root)
1  t ← root
2  while label(t) ≠ END
3      do if t is a leaf
4           then label(t) in the file fout
5                t ← root
6           else b ← read a bit from the file fin
7                if b = 1
8                  then t ← right(t)
9                  else t ← left(t)
```

FIGURE 13.59 Reads the compressed text and produces the uncompressed text.

```
DECODING(fin, fout)
1  create a new node root
2  REBUILD-TREE(fin, root)
3  DECODE-TEXT(fin, fout, root)
```

FIGURE 13.60 Complete function for Huffman decoding.

The complete decoding program is given in Figure 13.60. It calls the preceding functions. The running time of the decoding program is linear in the sum of the sizes of the texts it manipulates.

13.7.2 Lempel–Ziv–Welsh (LZW) Compression

Ziv and Lempel designed a compression method using encoding segments. These segments are stored in a dictionary that is built during the compression process. When a segment of the dictionary is encountered later while scanning the original text, it is substituted by its index in the dictionary. In the model where portions of the text are replaced by pointers on previous occurrences, the Ziv–Lempel compression scheme can be proved to be asymptotically optimal (on large enough texts satisfying good conditions on the probability distribution of symbols).

The dictionary is the central point of the algorithm. It has the property of being prefix closed (every prefix of a word of the dictionary is in the dictionary), so that it can be implemented as a tree. Furthermore, a hashing technique makes its implementation efficient. The version described in this section is called the Lempel–Ziv–Welsh method after several improvements introduced by Welsh. The algorithm is implemented by the **compress** command existing under the Unix operating system.

13.7.2.1 Compression Method

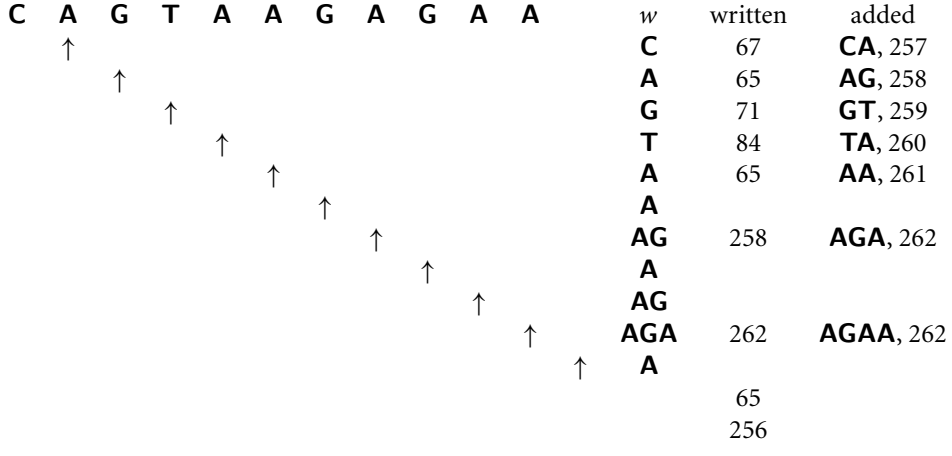
We describe the scheme of the compression method. The dictionary is initialized with all the characters of the alphabet. The current situation is when we have just read a segment w in the text. Let a be the next symbol (just following w). Then we proceed as follows:

- If wa is not in the dictionary, we write the index of w to the output file, and add wa to the dictionary. We then reset w to a and process the next symbol (following a).
- If wa is in the dictionary, we process the next symbol, with segment wa instead of w.

Initially, the segment w is set to the first symbol of the source text.

Example 13.22

Here $y =$ **CAGTAAGAGAA**



13.7.2.2 Decompression Method

The decompression method is symmetrical to the compression algorithm. The dictionary is recovered while the decompression process runs. It is basically done in this way:

- Read a code c in the compressed file.
- Write in the output file the segment w that has index c in the dictionary.
- Add to the dictionary the word wa where a is the first letter of the next segment.

In this scheme, a problem occurs if the next segment is the word that is being built. This arises only if the text contains a segment $azazax$ for which az belongs to the dictionary but aza does not. During the compression process, the index of az is written into the compressed file, and aza is added to the dictionary. Next, aza is read and its index is written into the file. During the decompression process, the index of aza is read while the word az has not been completed yet: the segment aza is not already in the dictionary. However, because this is the unique case where the situation arises, the segment aza is recovered, taking the last segment az added to the dictionary concatenated with its first letter a.

Example 13.23

Here, the decoding is 67, 65, 71, 84, 65, 258, 262, 65, 256

read	written	added
67	**C**	
65	**A**	**CA**, 257
71	**G**	**AG**, 258
84	**T**	**GT**, 259
65	**A**	**TA**, 260
258	**AG**	**AA**, 261
262	**AGA**	**AGA**, 262
65	**A**	**AGAA**, 263
256		

13.7.2.3 Implementation

For the compression algorithm shown in Figure 13.61, the dictionary is stored in a table D. The dictionary is implemented as a tree; each node z of the tree has the three following components:

- $parent(z)$ is a link to the parent node of z.
- $label(z)$ is a character.
- $code(z)$ is the code associated with z.

The tree is stored in a table that is accessed with a hashing function. This provides fast access to the children of a node. The procedure HASH-INSERT$((D,(p,a,c)))$ inserts a new node z in the dictionary D with $parent(z) = p, label(z) = a$, and $code(z) = c$. The function HASH-SEARCH$((D,(p,a)))$ returns the node z such that $parent(z) = p$ and $label(z) = a$.

```
COMPRESS(fin, fout)
 1  count ← −1
 2  for each character a ∈ Σ
 3      do count ← count + 1
 4         HASH-INSERT(D, (−1, a, count))
 5  count ← count + 1
 6  HASH-INSERT(D, (−1, END, count))
 7  p ← −1
 8  while not end of file fin
 9      do a ← next character of fin
10         q ← HASH-SEARCH(D, (p, a))
11         if q = NIL
12           then write code(p) on 1 + log(count) bits in fout
13                count ← count + 1
14                HASH-INSERT(D, (p, a, count))
15                p ← HASH-SEARCH(D, (−1, a))
16           else p ← q
17  write code(p) on 1 + log(count) bits in fout
18  write code(HASH-SEARCH(D, (−1, END))) on 1 + log(count) bits in fout
```

FIGURE 13.61 LZW compression algorithm.

```
UNCOMPRESS(fin, fout)
 1  count ← −1
 2  for each character a ∈ Σ
 3      do count ← count + 1
 4         HASH-INSERT(D, (−1, a, count))
 5  count ← count + 1
 6  HASH-INSERT(D, (−1, END, count))
 7  c ← first code on 1 + log(count) bits in fin
 8  write string(c) in fout
 9  a ← first(string(c))
10  while TRUE
11      do d ← next code on 1 + log(count) bits in fin
12         if d > count
13           then count ← count + 1
14                parent(count) ← c
15                label(count) ← a
16                write string(c)a in fout
17                c ← d
18           else a ← first(string(d))
19                if a ≠ END
20                  then count ← count + 1
21                       parent(count) ← c
22                       label(count) ← a
23                       write string(d) in fout
24                       c ← d
25                  else break
```

FIGURE 13.62 LZW decompression algorithm.

For the decompression algorithm, no hashing technique is necessary. Having the index of the next segment, a bottom-up walk in the trie implementing the dictionary produces the mirror image of the segment. A stack is used to reverse it. We assume that the function *string*(c) performs this specific work for a code c. The bottom-up walk follows the parent links of the data structure. The function *first*(w) gives the first character of the word w. These features are part of the decompression algorithm displayed in Figure 13.62.

The Ziv–Lempel compression and decompression algorithms run both in linear time in the sizes of the files provided a good hashing technique is chosen. Indeed, it is very fast in practice. Its main advantage compared to Huffman coding is that it captures long repeated segments in the source file.

13.7.3 Mixing Several Methods

We describe simple compression methods and then an example of a combination of several of them, the basis of the popular **bzip** software.

13.7.3.1 Run Length Encoding

The aim of Run Length Encoding (RLE) is to efficiently encode repetitions occurring in the input data. Let us assume that it contains a good quantity of repetitions of the form $aa \ldots a$ for some character a ($a \in \Sigma$). A repetition of k consecutive occurrences of letter a is replaced by $\&ak$, where the symbol & is a new character ($\& \notin \Sigma$).

The string &ak that encodes a repetition of k consecutive occurrences of a is itself encoded on the binary alphabet $\{0, 1\}$. In practice, letters are often represented by their ASCII code. Therefore, the codeword of a letter belongs to $\{0, 1\}^k$ with $k = 7$ or 8. Generally, there is no problem in choosing or encoding the special character &. The integer k of the string &ak is also encoded on the binary alphabet, but it is not sufficient to translate it by its binary representation, because we would be unable to recover it at decoding time inside the stream of bits. A simple way to cope with this is to encode k by the string $0^\ell \text{bin}(k)$, where $\text{bin}(k)$ is the binary representation of k, and ℓ is the length. This works well because the binary representation of k starts with a 1 so there is no ambiguity to recover ℓ by counting during the decoding phase. The size of the encoding of k is thus roughly $2 \log k$. More sophisticated integer representations are possible, but none is really suitable for the present situation. Simpler solution consists in encoding k on the same number of bits as other symbols, but this bounds values of ℓ and decreases the power of the method.

13.7.3.2 Move To Front

The Move To Front (MTF) method can be regarded as an extension of Run Length Encoding or a simplification of Ziv–Lempel compression. It is efficient when the occurrences of letters in the input text are localized into a relatively short segment of it. The technique is able to capture the proximity between occurrences of symbols and to turn it into a short encoded text.

Letters of the alphabet Σ of the input text are initially stored in a list that is managed dynamically. Letters are represented by their rank in the list, starting from 1, rank that is itself encoded as described above for RLE.

Letters of the input text are processed in an on-line manner. The clue of the method is that each letter is moved to the beginning of the list just after it is translated by the encoding of its rank.

The effect of MTF is to reduce the size of the encoding of a letter that reappears soon after its preceding occurrence.

13.7.3.3 Integrated Example

Most compression software combines several methods to be able to efficiently compress a large range of input data. We present an example of this strategy, implemented by the UNIX command **bzip**.

Let $y = y[0]y[1] \cdots y[n-1]$ be the input text. The k-th rotation (or conjugate) of y, $0 \leq k \leq n-1$, is the string $y_k = y[k]y[k+1] \cdots y[n-1]y[0]y[1] \cdots y[k-1]$.

We define the BW transformation as $BW(y) = y[p_0]y[p_1] \cdots y[p_{n-1}]$, where $p_i + 1$ is such that y_{p_i+1} has rank i in the sorted list of all rotations of y.

It is remarkable that y can be recovered from both $BW(y)$ and a position on it, starting position of the inverse transformation (see Figure 13.63). This is possible due to the following property of the transformation. Assume that $i < j$ and $y[p_i] = y[p_j] = a$. Since $i < j$, the definition implies $y_{p_i+1} < y_{p_j+1}$. Since $y[p_i] = y[p_j]$, transferring the last letters of y_{p_i+1} and y_{p_j+1} to the beginning of these words does not change the inequality. This proves that the two occurrences of a in $BW(y)$ are in the same relative order as in the sorted list of letters of y. Figure 13.63 illustrates the inverse transformation.

Transformation BW obviously does not compress the input text y. But $BW(y)$ is compressed more efficiently with simple methods. This is the strategy applied for the command **bzip**. It is a combination of the BW transformation followed by MTF encoding and RLE encoding. Arithmetic coding, a method providing compression ratios slightly better than Huffman coding, can also be used.

Table 13.1 contains a sample of experimental results showing the behavior of compression algorithms on different types of texts from the Calgary Corpus: `bib` (bibliography), `book1` (fiction book), `news` (USENET batch file), `pic` (black and white fax picture), `progc` (source code in C), and `trans` (transcript of terminal session).

The compression algorithms reported in the table are the Huffman coding algorithm implemented by **pack**, the Ziv–Lempel algorithm implemented by **gzip-b**, and the compression based on the BW transform implemented by **bzip2-1**.

Additional compression results can be found at `http://corpus.canterbury.ac.nz`.

TABLE 13.1 Compression Results with Three Algorithms. Huffman coding (**pack**), Ziv–Lempel coding (**gzip-b**) and Burrows-Wheeler coding (**bzip2-1**). Figures give the number of bits used per character (letter). They show that **pack** is the less efficient method and that **bzip2-1** compresses a bit more than **gzip-b**.

Sizes in bytes	111,261	768,771	377,109	513,216	39,611	93,695	
Source Texts	`bib`	`book1`	`news`	`pic`	`progc`	`trans`	Average
pack	5.24	4.56	5.23	1.66	5.26	5.58	4.99
gzip-b	2.51	3.25	3.06	0.82	2.68	1.61	2.69
bzip2-1	2.10	2.81	2.85	0.78	2.53	1.53	2.46

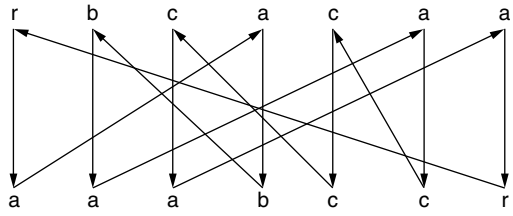


FIGURE 13.63 Example of text $y =$ **baccara**. Top line is $BW(y)$ and bottom line the sorted list of letters of it. Top-down arrows correspond to succession of occurrences in y. Each bottom-up arrow links the same occurrence of a letter in y. Arrows starting from equal letters do not cross. The circular path is associated with rotations of the string y. If the starting point is known, the only occurrence of letter **b** here, following the path produces the initial string y.

13.8 Research Issues and Summary

The algorithm for string searching by hashing was introduced by Harrison in 1971, and later fully analyzed by Karp and Rabin (1987).

The linear-time string-matching algorithm of Knuth, Morris, and Pratt is from 1976. It can be proved that, during the search, a character of the text is compared to a character of the pattern no more than $\log_\Phi(|x|+1)$ (where Φ is the golden ratio $(1+\sqrt{5})/2$). Simon (1993) gives an algorithm similar to the previous one but with a delay bounded by the size of the alphabet (of the pattern x). Hancart (1993) proves that the delay of Simon's algorithm is, indeed, no more than $1+\log_2|x|$. He also proves that this is optimal among algorithms searching the text through a window of size 1.

Galil (1981) gives a general criterion to transform searching algorithms of that type into real-time algorithms.

The Boyer–Moore algorithm was designed by Boyer and Moore (1977). The first proof on the linearity of the algorithm when restricted to the search of the first occurrence of the pattern is in Knuth et al. (1977). Cole (1994) proves that the maximum number of symbol comparisons is bounded by $3n$, and that this bound is tight.

Knuth et al. (1977) consider a variant of the Boyer–Moore algorithm in which all previous matches inside the current window are memorized. Each window configuration becomes the state of what is called the Boyer–Moore automaton. It is still unknown whether the maximum number of states of the automaton is polynomial or not.

Several variants of the Boyer–Moore algorithm avoid the quadratic behavior when searching for all occurrences of the pattern. Among the more efficient in terms of the number of symbol comparisons are the algorithm of Apostolico and Giancarlo (1986), Turbo–BM algorithm by Crochemore et al. (1992) (the two algorithms are analyzed in Lecroq (1995)), and the algorithm of Colussi (1994).

The general bound on the expected time complexity of string matching is $O(|y|\log|x|/|x|)$. The probabilistic analysis of a simplified version of the Boyer–Moore algorithm, similar to the Quick Search algorithm of Sunday (1990) described in the chapter, was studied by several authors.

String searching can be solved by a linear-time algorithm requiring only a constant amount of memory in addition to the pattern and the (window on the) text. This can be proved by different techniques presented in Crochemore and Rytter (2002).

The Aho–Corasick algorithm is from Aho and Corasick (1975). It is implemented by the **fgrep** command under the UNIX operating system. Commentz-Walter (1979) has designed an extension of the Boyer-Moore algorithm to several patterns. It is fully described in Aho (1990).

On general alphabets the two-dimensional pattern matching can be solved in linear time, whereas the running time of the Bird/Baker algorithm has an additional $\log \sigma$ factor. It is still unknown whether the problem can be solved by an algorithm working simultaneously in linear time and using only a constant amount of memory space (see Crochemore and Rytter 2002).

The suffix tree construction of Section 13.2 is by McCreight (1976). An on-line construction is given by Ukkonen (1995). Other data structures to represent indexes on text files are: direct acyclic word graph (Blumer et al., 1985), suffix automata (Crochemore, 1986), and suffix arrays (Manber and Myers, 1993). All these techniques are presented in (Crochemore and Rytter, 2002). The data structures implement full indexes with standard operations, whereas applications sometimes need only incomplete indexes. The design of compact indexes is still unsolved.

First algorithms for aligning two sequences are by Needleman and Wunsch (1970) and Wagner and Fischer (1974). Idea and algorithm for local alignment is by Smith and Waterman (1981). Hirschberg (1975) presents the computation of the lcs in linear space. This is an important result because the algorithm is classically run on large sequences. Another implementation is given in Durbin et al. (1998). The quadratic time complexity of the algorithm to compute the Levenshtein distance is a bottleneck in practical string comparison for the same reason.

Approximate string searching is a lively domain of research. It includes, for instance, the notion of regular expressions to represent sets of strings. Algorithms based on regular expression are commonly found in books related to compiling techniques. The algorithms of Section 13.6 are by Baeza-Yates and Gonnet (1992) and Wu and Manber (1992).

The statistical compression algorithm of Huffman (1951) has a dynamic version where symbol counting is done at coding time. The current coding tree is used to encode the next character and then updated. At decoding time, a symmetrical process reconstructs the same tree, so the tree does not need to be stored with the compressed text; see Knuth (1985). The command **compact** of UNIX implements this version.

Several variants of the Ziv and Lempel algorithm exist. The reader can refer to Bell et al. (1990) for further discussion. Nelson (1992) presents practical implementations of various compression algorithms. The *BW* transform is from Burrows and Wheeler (1994).

Defining Terms

Alignment: An alignment of two strings x and y is a word of the form $(\overline{x}_0, \overline{y}_0)(\overline{x}_1, \overline{y}_1)\cdots(\overline{x}_{p-1}, \overline{y}_{p-1})$ where each $(\overline{x}_i, \overline{y}_i) \in (\Sigma \cup \{\varepsilon\}) \times (\Sigma \cup \{\varepsilon\}) \setminus (\{(\varepsilon, \varepsilon)\}$ for $0 \le i \le p-1$ and both $x = \overline{x}_0\overline{x}_1\cdots\overline{x}_{p-1}$ and $y = \overline{y}_0\overline{y}_1\cdots\overline{y}_{p-1}$.

Border: A word $u \in \Sigma^*$ is a border of a word $w \in \Sigma^*$ if u is both a prefix and a suffix of w (there exist two words $v, z \in \Sigma^*$ such that $w = vu = uz$). The common length of v and z is a period of w.

Edit distance: The metric distance between two strings that counts the minimum number of insertions and deletions of symbols to transform one string into the other.

Hamming distance: The metric distance between two strings of same length that counts the number of mismatches.

Levenshtein distance: The metric distance between two strings that counts the minimum number of insertions, deletions, and substitutions of symbols to transform one string into the other.

Occurrence: An occurrence of a word $u \in \Sigma^*$, of length m, appears in a word $w \in \Sigma^*$, of length n, at position i if for $0 \le k \le m-1$, $u[k] = w[i+k]$.

Prefix: A word $u \in \Sigma^*$ is a prefix of a word $w \in \Sigma^*$ if $w = uz$ for some $z \in \Sigma^*$.

Prefix code: Set of words such that no word of the set is a prefix of another word contained in the set. A prefix code is represented by a coding tree.

Segment: A word $u \in \Sigma^*$ is a segment of a word $w \in \Sigma^*$ if u occurs in w (see occurrence); that is, $w = vuz$ for two words $v, z \in \Sigma^*$ (u is also referred to as a factor or a subword of w).

Subsequence: A word $u \in \Sigma^*$ is a subsequence of a word $w \in \Sigma^*$ if it is obtained from w by deleting zero or more symbols that need not be consecutive (u is sometimes referred to as a subword of w, with a possible confusion with the notion of segment).

Suffix: A word $u \in \Sigma^*$ is a suffix of a word $w \in \Sigma^*$ if $w = vu$ for some $v \in \Sigma^*$.

Suffix tree: Trie containing all the suffixes of a word.

Trie: Tree in which edges are labeled by letters or words.

References

Aho, A.V. 1990. Algorithms for finding patterns in strings. In *Handbook of Theoretical Computer Science*, Vol. A. *Algorithms and Complexity*, J. van Leeuwen, Ed., pp. 255–300. Elsevier, Amsterdam.

Aho, A.V. and Corasick, M.J. 1975. Efficient string matching: an aid to bibliographic search. *Comm. ACM*, 18(6):333–340.

Baeza-Yates, R.A. and Gonnet, G.H. 1992. A new approach to text searching. *Comm. ACM*, 35(10):74–82.

Baker, T.P. 1978. A technique for extending rapid exact-match string matching to arrays of more than one dimension. *SIAM J. Comput.*, 7(4):533–541.

Bell, T.C., Cleary, J.G., and Witten, I.H. 1990. *Text Compression*. Prentice Hall, Englewood Cliffs, NJ.

Bird, R.S. 1977. Two-dimensional pattern matching. *Inf. Process. Lett.*, 6(5):168–170.

Blumer, A., Blumer, J., Ehrenfeucht, A., Haussler, D., Chen, M.T., and Seiferas, J. 1985. The smallest automaton recognizing the subwords of a text. *Theor. Comput. Sci.*, 40:31–55.

Boyer, R.S. and Moore, J.S. 1977. A fast string searching algorithm. *Comm. ACM*, 20(10):762–772.

Breslauer, D., Colussi, L., and Toniolo, L. 1993. Tight comparison bounds for the string prefix matching problem. *Inf. Process. Lett.*, 47(1):51–57.

Burrows, M. and Wheeler, D. 1994. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation.

Cole, R. 1994. Tight bounds on the complexity of the Boyer-Moore pattern matching algorithm. *SIAM J. Comput.*, 23(5):1075–1091.

Colussi, L. 1994. Fastest pattern matching in strings. *J. Algorithms*, 16(2):163–189.

Crochemore, M. 1986. Transducers and repetitions. *Theor. Comput. Sci.*, 45(1):63–86.

Crochemore, M. and Rytter, W. 2002. *Jewels of Stringology*. World Scientific.

Durbin, R., Eddy, S., and Krogh, A., and Mitchison G. 1998. *Biological Sequence Analysis Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press.

Galil, Z. 1981. String matching in real time. *J. ACM*, 28(1):134–149.

Hancart, C. 1993. On Simon's string searching algorithm. *Inf. Process. Lett.*, 47(2):95–99.

Hirschberg, D.S. 1975. A linear space algorithm for computing maximal common subsequences. *Comm. ACM*, 18(6):341–343.

Hume, A. and Sunday, D.M. 1991. Fast string searching. *Software — Practice Exp.*, 21(11):1221–1248.

Karp, R.M. and Rabin, M.O. 1987. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260.

Knuth, D.E. 1985. Dynamic Huffman coding. *J. Algorithms*, 6(2):163–180.

Knuth, D.E., Morris, J.H., Jr, and Pratt, V.R. 1977. Fast pattern matching in strings. *SIAM J. Comput.*, 6(1):323–350.

Lecroq, T. 1995. Experimental results on string-matching algorithms. *Software — Practice Exp.* 25(7): 727–765.

McCreight, E.M. 1976. A space-economical suffix tree construction algorithm. *J. Algorithms*, 23(2): 262–272.

Manber, U. and Myers, G. 1993. Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948.
Needleman, S.B. and Wunsch, C.D. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.*, 48:443–453.
Nelson, M. 1992. *The Data Compression Book*. M&T Books.
Simon, I. 1993. String matching algorithms and automata. In *First American Workshop on String Processing*, Baeza-Yates and Ziviani, Eds., pp. 151–157. Universidade Federal de Minas Gerais.
Smith, T.F. and Waterman, M.S. 1981. Identification of common molecular sequences. *J. Mol. Biol.*, 147:195–197.
Stephen, G.A. 1994. *String Searching Algorithms*. World Scientific Press.
Sunday, D.M. 1990. A very fast substring search algorithm. *Commun. ACM* 33(8):132–142.
Ukkonen, E. 1995. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260.
Wagner, R.A. and Fischer, M. 1974. The string-to-string correction problem. *J. ACM*, 21(1):168–173.
Welch, T. 1984. A technique for high-performance data compression. *IEEE Comput.* 17(6):8–19.
Wu, S. and Manber, U. 1992. Fast text searching allowing errors. *Commun. ACM*, 35(10):83–91.
Zhu, R.F. and Takaoka, T. 1989. A technique for two-dimensional pattern matching. *Commun. ACM*, 32(9):1110–1120.

Further Information

Problems and algorithms presented in the chapter are just a sample of questions related to pattern matching. They share the formal methods used to design solutions and efficient algorithms. A wider panorama of algorithms on texts can be found in books, other including:

Apostolico, A. and Galil, Z., Editors. 1997. *Pattern Matching Algorithms*. Oxford University Press.
Bell, T.C., Cleary, J.G., and Witten, I.H. 1990. *Text Compression*. Prentice Hall, Englewood Cliffs, NJ.
Crochemore, M. and Rytter, W. 2002. *Jewels of Stringology.* World Scientific.
Gusfield D. 1997. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press.
Navarro, G. and Raffinot M. 2002. *Flexible Pattern Matching in Strings: Practical On-line Search Algorithms for Texts and Biological Sequences*. Cambridge University Press.
Nelson, M. 1992. *The Data Compression Book.* M&T Books.
Salomon, D. 2000. *Data Compression: the Complete Reference*. Springer-Verlag.
Stephen, G.A. 1994. *String Searching Algorithms.* World Scientific Press.

Research papers in pattern matching are disseminated in a few journals, among which are: *Communications of the ACM*, *Journal of the ACM*, *Theoretical Computer Science*, *Algorithmica*, *Journal of Algorithms*, *SIAM Journal on Computing*, and *Journal of Discrete Algorithms*.

Finally, three main annual conferences present the latest advances of this field of research and Combinatorial Pattern Matching, which started in 1990. Data Compression Conference, which is regularly held at Snowbird. The scope of SPIRE (String Processing and Information Retrieval) includes the domain of data retrieval.

General conferences in computer science often have sessions devoted to pattern matching algorithms.

Several books on the design and analysis of general algorithms contain chapters devoted to algorithms on texts. Here is a sample of these books:

Cormen, T.H., Leiserson, C.E., and Rivest, R.L. 1990. *Introduction to Algorithms*. MIT Press.
Gonnet, G.H. and Baeza-Yates, R.A. 1991. *Handbook of Algorithms and Data Structures*. Addison-Wesley.

Animations of selected algorithms can be found at:

`http://www-igm.univ-mlv.fr/~lecroq/string/` (Exact String Matching Algorithms),
`http://www-igm.univ-mlv.fr/~lecroq/seqcomp/` (Alignments).

14

Genetic Algorithms

Stephanie Forrest
University of New Mexico

14.1 Introduction

A genetic algorithm is a form of evolution that occurs in a computer. Genetic algorithms are useful, both as search methods for solving problems and for modeling evolutionary systems. This chapter describes how genetic algorithms work, gives several examples of genetic algorithm applications, and reviews some mathematical analysis of genetic algorithm behavior.

In genetic algorithms, strings of binary digits are stored in a computer's memory, and over time the properties of these strings evolve in much the same way that populations of individuals evolve under natural selection. Although the computational setting is highly simplified when compared with the natural world, genetic algorithms are capable of evolving surprisingly complex and interesting structures. These structures, called **individuals**, can represent solutions to problems, strategies for playing games, visual images, or computer programs. Thus, genetic algorithms allow engineers to use a computer to evolve problem solutions over time, instead of designing them by hand. Although genetic algorithms are known primarily as a problem-solving method, they can also be used to study and model evolution in various settings, including biological (such as ecologies, immunology, and population genetics), social (such as economies and political systems), and cognitive systems.

14.2 Underlying Principles

The basic idea of a genetic algorithm is quite simple. First, a population of individuals is created in a computer, and then the population is evolved using the principles of variation, selection, and inheritance. Random variations in the population result in some individuals being more fit than others (better suited to their environment). These individuals have more offspring, passing on successful variations to their children, and the cycle is repeated. Over time, the individuals in the population become better adapted to their environment. There are many ways of implementing this simple idea. Here I describe the one invented by Holland [1975, Goldberg 1989].

The idea of using selection and variation to evolve solutions to problems goes back at least to Box [1957], although his work did not use a computer. In the late 1950s and early 1960s there were several independent

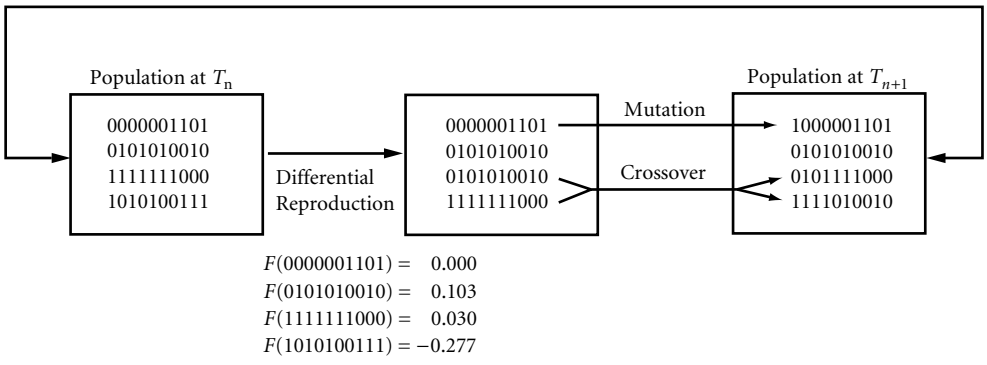


FIGURE 14.1 (See Plate 14.1 in the color insert following page **29**-22.) Genetic algorithm overview: A population of four individuals is shown. Each is assigned a fitness value by the function $F(x, y) = yx^2 - x^4$. (See Figure 14.3.) On the basis of these fitnesses, the selection phase assigns the first individual (0000001101) one copy, the second (0101010010) two copies, the third (1111111000) one copy, and the fourth (1010100111) zero copies. After selection, the genetic operators are applied probabilistically; the first individual has its first bit mutated from a 0 to a 1, and crossover combines the last two individuals into two new ones. The resulting population is shown in the box labeled $T_{(N+1)}$.

efforts to incorporate ideas from evolution in computation. Of these, the best known are genetic algorithms [Holland 1962], evolutionary programming [Fogel et al. 1966], and evolutionary strategies [Back and Schwefel 1993]. Rechenberg [Back and Schwefel 1993] emphasized the importance of selection and mutation as mechanisms for solving difficult real-valued optimization problems. Fogel et al. [1966] developed similar ideas for evolving intelligent agents in the form of finite state machines. Holland [1962, 1975] emphasized the adaptive properties of entire populations and the importance of recombination mechanisms such as **crossover**. In recent years, genetic algorithms have taken many forms, and in some cases bear little resemblance to Holland's original formulation. Researchers have experimented with different types of representations, crossover and mutation operators, special-purpose operators, and different approaches to reproduction and selection. However, all of these methods have a family resemblance in that they take some inspiration from biological evolution and from Holland's original genetic algorithm. A new term, *evolutionary computation*, has been introduced to cover these various members of the genetic algorithm family, evolutionary programming, and evolution strategies.

Figure 14.1 gives an overview of a simple genetic algorithm. In its simplest form, each individual in the population is a bit string. Genetic algorithms often use more complex representations, including richer alphabets, diploidy, redundant encodings, and multiple **chromosomes**. However, the binary case is both the simplest and the most general. By analogy with genetics, the string of bits is referred to as the **genotype**. Each individual consists only of its genetic material, and it is organized into one (haploid) chromosome. Each bit position (set to 1 or 0) represents one gene. I will use the term bit string to refer both to genotypes and the individuals that they define. A natural question is how genotypes built from simple strings of bits can specify a solution to a specific problem. In other words, how are the binary genes expressed? There are many techniques for mapping bit strings to different problem domains, some of which are described in the following subsections.

The initial population of individuals is usually generated randomly, although it need not be. For example, prior knowledge about the problem solution can be encoded directly into the initial population, as in Hillis [1990]. Each individual is tested empirically in an environment, receiving a numerical evaluation of its merit, assigned by a **fitness function** F. The environment can be almost anything: another computer simulation, interactions with other individuals in the population, actions in the physical world (by a robot for example), or a human's subjective judgment. The fitness function's evaluation typically returns a single number (usually, higher numbers are assigned to fitter individuals). This constraint is sometimes relaxed so that the fitness function returns a vector of numbers [Fonseca and Fleming 1995], which can be appropriate for problems with multiple objectives. The fitness function determines how each gene (bit)

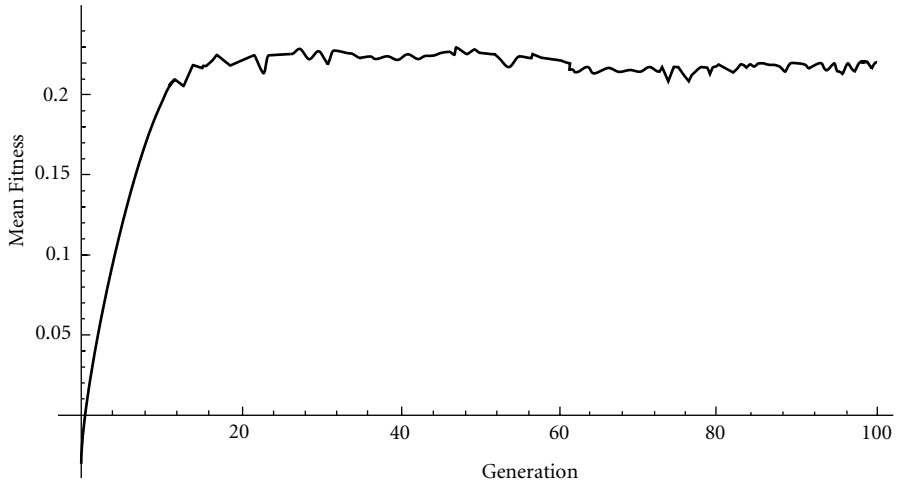


FIGURE 14.2 Mean fitness of a population evolving under the genetic algorithm. The population size is 100 individuals, each of which is 10 bits long (5 bits for x, 5 bits for y, as described in Figure 14.3), mutation probability is 0.0026/bit, crossover probability is 0.6 per pair of individuals, and the fitness function is $F = yx^2 - x^4$. Population mean is shown every generation for 100 generations.

of an individual will be interpreted and thus what specific problem the population will evolve to solve. The fitness function is the primary place where the traditional genetic algorithm is tailored to a specific problem.

Once all individuals in the population have been evaluated, their fitnesses form the basis for selection. **Selection** is implemented by eliminating low-fitness individuals from the population, and inheritance is implemented by making multiple copies of high-fitness individuals. Genetic operators such as **mutation** (flipping individual bits) and crossover (exchanging substrings of two individuals to obtain new offspring) are then applied probabilistically to the selected individuals to produce a new population (or **generation**) of individuals. The term crossover is used here to refer to the exchange of homologous substrings between individuals, although the biological term crossing over generally implies exchange within an individual. New generations can be produced either synchronously, so that the old generation is completely replaced, or asynchronously, so that generations overlap.

By transforming the previous set of good individuals to a new one, the operators generate a new set of individuals that ideally have a better than average chance of also being good. When this cycle of evaluation, selection, and genetic operations is iterated for many generations, the overall fitness of the population generally improves, as shown in Figure 14.2, and the individuals in the population represent improved solutions to whatever problem was posed in the fitness function.

There are many details left unspecified by this description. For example, selection can be performed in any of several ways — it could arbitrarily eliminate the least fit 50% of the population and make one copy of all of the remaining individuals, it could replicate individuals in direct proportion to their fitness (fitness-proportionate selection), or it could scale the fitnesses in any of several ways and replicate individuals in direct proportion to their scaled values (a more typical method). Similarly, the crossover operator can pass on both offspring to the new generation, or it can arbitrarily choose one to be passed on; the number of crossover points can be restricted to one per pair, two per pair, or N per pair. These and other variations of the basic algorithm have been discussed extensively in Goldberg [1989], in Davis [1991], and in the Proceedings of the International Conference on Genetic Algorithms. (See Further Information section.)

The genetic algorithm is interesting from a computational standpoint, at least in part, because of the claims that have been made about its effectiveness as a biased sampling algorithm. The classical argument

about genetic algorithm performance has three components [Holland 1975, Goldberg 1989]:

- Independent sampling is provided by large populations that are initialized randomly.
- High-fitness individuals are preserved through selection, and this biases the sampling process toward regions of high fitness.
- Crossover combines partial solutions, called building blocks, from different strings onto the same string, thus exploiting the parallelism provided by the population of candidate solutions.

A partial solution is taken to be a hyperplane in the search space of strings and is called a **schema** (see Section 14.4). A central claim about genetic algorithms is that schemas capture important regularities in the search space and that a form of *implicit parallelism* exists because one fitness evaluation of an individual comprising l bits implicitly gives information about the 2^l schemas, or hyperplanes, of which it is an instance. The Schema Theorem states that the genetic algorithm operations of reproduction, mutation, and crossover guarantee exponentially increasing samples of the observed best schemas in the next time step. By analogy with the k-armed bandit problem it can be argued that the genetic algorithm uses an optimal sampling strategy [Holland 1975]. See Section 14.4 for details.

14.3 Best Practices

The simple computational procedure just described can be applied in many different ways to solve a wide range of problems. In designing a genetic algorithm to solve a specific problem there are two major design decisions: (1) specifying the mapping between binary strings and candidate solutions (this is commonly referred to as the representation problem) and (2) defining a concrete measure of fitness. In some cases the best representation and fitness function are obvious, but in many cases they are not, and in all cases, the particular representation and fitness function that are selected will determine the ultimate success of the genetic algorithm on the chosen problem. Possibly the simplest representation is a *feature list* in which each bit, or gene, represents the presence or absence of a single feature. This representation is useful for learning pattern classes defined by a critical set of features. For example, in spectroscopic applications, an important problem is selecting a small number of spectral frequencies that predict the concentration of some substance (e.g., concentration of glucose in human blood). The feature list approach to this problem assigns 1 bit to represent the presence or absence of each different observable frequency, and high fitness is assigned to those individuals whose feature settings correspond to good predictors for high (or low) glucose levels [Thomas 1993].

Genetic algorithms in various forms have been applied to many scientific and engineering problems, including optimization, automatic programming, machine and robot learning, modeling natural systems, and artificial life. They have been used in a wide variety of optimization tasks, including numerical optimization (see section on function optimization) and combinatorial optimization problems such as circuit design and job shop scheduling (see section on ordering problems). Genetic algorithms have also been used to evolve computer programs for specific tasks (see section on automatic programming) and to design other computational structures, e.g., cellular automata rules and sorting networks. In machine learning, they have been used to design neural networks, to evolve rules for rule-based systems, and to design and control robots. For an overview of genetic algorithms in machine learning, see DeJong [1990a, 1990b] and Schaffer et al. [1992].

Genetic algorithms have been used to model processes of innovation, the development of bidding strategies, the emergence of economic markets, the natural immune system, and ecological phenomena such as biological arms races, host–parasite coevolution, symbiosis, and resource flow. They have been used to study evolutionary aspects of social systems, such as the evolution of cooperation, the evolution of communication, and trail-following behavior in ants. They have been used to study questions in population genetics, such as "under what conditions will a gene for recombination be evolutionarily viable?" Finally, genetic algorithms are an important component in many artificial-life models, including systems that model interactions between species evolution and individual learning. See Further Information section and Mitchell and Forrest [1994] for details about genetic algorithms in modeling and artificial life.

The remainder of this section describes four illustrative examples of how genetic algorithms are used: numerical encodings for function optimization, permutation representations and special operators for sequencing problems, computer programs for automated programming, and endogenous fitness and other extensions for ecological modeling. The first two cover the most common classes of engineering applications. They are well understood and noncontroversial. The third example illustrates one of the most promising recent advances in genetic algorithms, but it was developed more recently and is less mature than the first two. The final example shows how genetic algorithms can be modified to more closely approximate natural evolutionary processes.

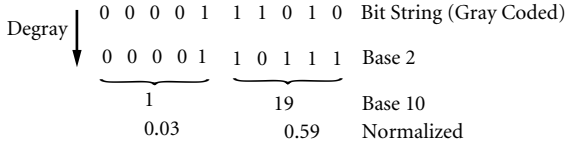
14.3.1 Function Optimization

Perhaps the most common application of genetic algorithms, pioneered by DeJong [1975], is multiparameter function optimization. Many problems can be formulated as a search for an optimal value, where the value is a complicated function of some input parameters. In some cases, the parameter settings that lead to the exact greatest (or least) value of the function are of interest. In other cases, the exact optimum is not required, just a near optimum, or even a value that represents a slight improvement over the previously best-known value. In these latter cases, genetic algorithms are often an appropriate method for finding good values.

As a simple example, consider the function $f(x, y) = yx^2 - x^4$. This function is solvable analytically, but if it were not, a genetic algorithm could be used to search for values of x and y that produce high values of $f(x, y)$ in a particular region of $\Re^2$. The most straightforward representation (Figure 14.3) is to assign regions of the bit string to represent each parameter (variable). Once the order in which the parameters are to appear is determined (in the figure x appears first and y appears second), the next step is to specify the domain for x and y (that is, the set of values for x and y that are candidate solutions). In our example, x and y will be real values in the interval $[0, 1)$. Because x and y are real valued in this example, and we are using a bit representation, the parameters need to be discretized. The precision of the solution is determined by how many bits are used to represent each parameter. In the example, 5 bits are assigned for x and 5 for y, although 10 is a more typical number. There are different ways of mapping between bits and decimal numbers, and so an encoding must also be chosen, and here we use gray coding.

Once a representation has been chosen, the genetic algorithm generates a random population of bit strings, decodes each bit string into the corresponding decimal values for x and y, applies the fitness function ($f(x, y) = yx^2 - x^4$) to the decoded values, selects the most fit individuals [those with the highest $f(x, y)$] for copying and variation, and then repeats the process. The population will tend to converge on a set of bit strings that represents an optimal or near optimal solution. However, there will always be some variation in the population due to mutation (Figure 14.2).

The standard binary encoding of decimal values has the drawback that in some cases all of the bits must be changed in order to increase a number by one. For example, the bit pattern 011 translates to 3 in decimal,



$$F(0000111010) = F(0.03, 0.59) = 0.59 \times (0.03)^2 - (0.03)^4 = 0.0005$$

FIGURE 14.3 Bit-string encoding of multiple real-valued parameters. An arbitrary string of 10 bits is interpreted in the following steps: (1) segment the string into two regions with the first 5 bits reserved for x and the second 5 bits for y; (2) interpret each 5-bit substring as a Gray code and map back to the corresponding binary code; (3) map each 5-bit substring to its decimal equivalent; (4) scale to the interval $[0, 1)$; (5) substitute the two scaled values for x and y in the fitness function F; (6) return $F(x, y)$ as the fitness of the original string.

but 4 is represented by 100. This can make it difficult for an individual that is close to an optimum to move even closer by mutation. Also, mutations in high-order bits (the leftmost bits) are more significant than mutations in low-order bits. This can violate the idea that bit strings in successive generations will have a better than average chance of having high fitness, because mutations may often be disruptive. Gray codes address the first of these problems. Gray codes have the property that incrementing or decrementing any number by one is always 1 bit change. In practice, Gray-coded representations are often more successful for multiparameter function optimization applications of genetic algorithms.

Many genetic algorithm practitioners encode real-valued parameters directly without converting to a bit-based representation. In this approach, each parameter can be thought of as a gene on the chromosome. Crossover is defined as before, except that crosses take place only between genes (between real numbers). Mutation is typically redefined so that it chooses a random value that is close to the current value. This representation strategy is often more effective in practice, but it requires some modification of the operators [Back and Schwefel 1993, Davis 1991]. There are a number of other representation tricks that are commonly employed for function optimization, including logarithmic scaling (interpreting bit strings as the logarithm of the true parameter value), dynamic encoding (a technique that allows the number and interpretation of bits allocated to a particular parameter to vary throughout a run), variable-length representations, delta coding (the bit strings express a distance away from some previous partial solution), and a multitude of nonbinary encodings.

This completes our description of a simple method for encoding parameters onto a bit string. Although a function of two variables was used as an example, the strength of the genetic algorithm lies in its ability to manipulate many parameters, and this method has been used for hundreds of applications, including aircraft design, tuning parameters for algorithms that detect and track multiple signals in an image, and locating regions of stability in systems of nonlinear difference equations. See Goldberg [1989], Davis [1991], and the Proceedings of the International Conference on Genetic Algorithms for more detail about these and other examples of successful function-optimization applications.

14.3.2 Ordering Problems

A common problem involves finding an optimal ordering for a sequence of N items. Examples include various NP-complete problems such as finding a tour of cities that minimizes the distance traveled (the traveling salesman problem), packing boxes into a bin to minimize wasted space (the bin packing problem), and graph coloring problems.

For example, in the traveling salesman problem, suppose there are four cities: 1, 2, 3, and 4 and that each city is labeled by a unique bit string.* A common fitness function for this problem is the length of the candidate tour. A natural way to represent a tour is as a permutation, so that 3 2 1 4 is one candidate tour and 4 1 2 3 is another. This representation is problematic for the genetic algorithm because mutation and crossover do not necessarily produce legal tours. For example, a crossover between positions two and three in the example produces the individuals 3 2 2 3 and 4 1 1 4, both of which are illegal tours — not all of the cities are visited and some are visited more than once.

Three general methods have been proposed to address this representation problem: (1) adopting a different representation, (2) designing specialized crossover operators that produce only legal tours, and (3) penalizing illegal solutions through the fitness function. Of these, the use of specialized operators has been the most successful method for applications of genetic algorithms to ordering problems such as the traveling salesman problem (for example, see Mühlenbein et al. [1988]), although a number of generic representations have been proposed and used successfully on other sequencing problems. Specialized crossover operators tend to be less general, and I will describe one such method, **edge recombination**, as an example of a special-purpose operator that can be used with the permutation representation already described.

*For simplicity, we will use integers in the following explanation rather than the bit strings to which they correspond.

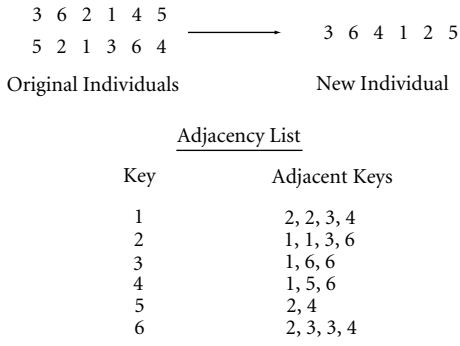


FIGURE 14.4 Example of edge-recombination operator. The adjacency list is constructed by examining each element in the parent permutations (labeled Key) and recording its adjacent elements. The new individual is constructed by selecting one parent arbitrarily (the top parent) and assigning its first element (3) to be the first element in the new permutation. The adjacencies of 3 are examined, and 6 is chosen to be the second element because it is a shared adjacency. The adjacencies of 6 are then examined, and of the unused ones, 4 is chosen randomly. Similarly, 1 is assigned to be the fourth element in the new permutation by random choice from $\{1, 5\}$. Then 2 is placed as the fifth element because it is a shared adjacency, and then the one remaining element, 5, is placed in the last position.

When designing special-purpose operators it is important to consider what information from the parents is being transmitted to the offspring, that is, what information is correlated with high-fitness individuals. In the case of traditional bitwise crossover, the answer is generally short, low-order schemas. (See Section 14.4.) But in the case of sequences, it is not immediately obvious what this means. Starkweather et al. [1991] identified three potential kinds of information that might be important for solving an ordering problem and therefore important to preserve through recombination: absolute position in the order, relative ordering (e.g., precedence relations might be important for a scheduling application), and adjacency information (as in the traveling salesman problem). They designed the edge-recombination operator to emphasize adjacency information. The operator is rather complicated, and there are many variants of the originally published operator. A simplified description follows (for details, see Starkweather et al. [1991]). For each pair of individuals to be crossed: (1) construct a table of adjacencies in the parents (see Figure 14.4) and (2) construct one new permutation (offspring) by combining information from the two parents:

- Select one parent at random and assign the first element in its permutation to be the first one in the child.
- Select the second element for the child, as follows: If there is an adjacency common to both parents, then choose that element to be the next one in the child's permutation; if there is an unused adjacency available from one parent, choose it; or if (1) and (2) fail, make a random selection.
- Select the remaining elements in order by repeating step 2.

An example of the edge-recombination operator is shown in Figure 14.4. Although this method has proved effective, it should be noted that it is more expensive to build the adjacency list for each parent and to perform edge recombination operation than it is to use a more standard crossover operator.

A final consideration in the choice of special-purpose operators is the amount of random information that is introduced when the operator is applied. This can be difficult to assess, but it can have a large effect (positive or negative) on the performance of the operator.

14.3.3 Automatic Programming

Genetic algorithms have been used to evolve a special kind of computer program [Koza 1992]. These programs are written in a subset of the programming language Lisp and more recently other languages.

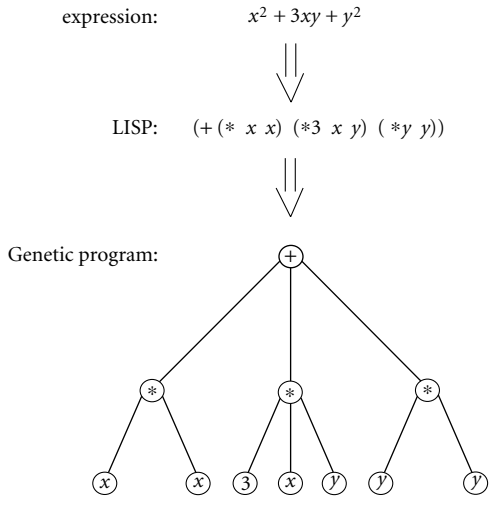


FIGURE 14.5 Tree representation of computer programs: The displayed tree corresponds to the expression $x^2 + 3xy + y^2$. Operators for each expression are displayed as a root, and the operands for each expression are displayed as children. (From Forrest, S. 1993a. *Science* 261:872–878. With permission.)

Lisp programs can naturally be represented as trees (Figure 14.5). Populations of random program trees are generated and evaluated as in the standard genetic algorithm. All other details are similar to those described for binary genetic algorithms with the exception of crossover. Instead of exchanging substrings, **genetic programs** exchange subtrees between individual program trees. This modified form of crossover appears to have many of the same advantages as traditional crossover (such as preserving partial solutions).

Genetic programming has the potential to be extremely powerful, because Lisp is a general-purpose programming language and genetic programming eliminates the need to devise an explicit chromosomal representation. In practice, however, genetic programs are built from subsets of Lisp tailored to particular problem domains, and at this point considerable skill is required to select just the right set of primitives for a particular problem. Although the method has been tested on a wide variety of problems, it has not yet been used extensively in real applications.

The genetic programming method is intriguing because its solutions are so different from human-designed programs for the same problem. Humans try to design elegant and general computer programs, whereas genetic programs are often needlessly complicated, not revealing the underlying algorithm. For example, a human-designed program for computing $\cos 2x$ might be $1 - 2\sin^2 x$, expressed in Lisp as $(-1(*2(*(\sin x)(\sin x))))$, whereas genetic programming discovered the following program (Koza 1992, p. 241):

$$(\sin(-(-2(*x2))(\sin(\sin(\sin(\sin(\sin(\sin(*(\sin(\sin 1))(\sin(\sin 1)))))))))))$$

For anyone who has studied computer programming this is apparently a major drawback because the evolved programs are inelegant, redundant, inefficient, difficult for a human to read, and do not reveal the underlying structure of the algorithm. However, genetic programs do resemble the kinds of ad hoc solutions that evolve in nature through gene duplication, mutation, and modifying structures from one purpose to another. There is some evidence that the junk components of a genetic program sometimes turn out to be useful components in other contexts. Thus, if the genetic programming endeavor is successful, it could revolutionize software design.

14.3.4 Genetic Algorithms for Making Models

The past three examples concentrated on understanding how genetic algorithms can be applied to solve problems. This subsection discusses how the genetic algorithm can be used to model other systems. Genetic algorithms have been employed as models of a wide variety of dynamical processes, including induction in psychology, natural evolution in ecosystems, evolution in immune systems, and imitation in social systems. Making computer models of evolution is somewhat different from many conventional models because the models are highly abstract. The data produced by these models are unlikely to make exact numerical predictions. Rather, they can reveal the conditions under which certain qualitative behaviors are likely to arise — diversity of phenotypes in resource-rich (or poor) environments, cooperation in competitive nonzero-sum games, and so forth. Thus, the models described here are being used to discover qualitative patterns of behavior and, in some cases, critical parameters in which small changes have drastic effects on the outcomes. Such modeling is common in nonlinear dynamics and in artificial intelligence, but it is much less accepted in other disciplines. Here we describe one of these examples: ecological modeling. This exploratory research project is still in an early stage of development. For examples of more mature modeling projects, see Holland et al. [1986] and Axelrod [1986].

The Echo system [Holland 1995] shows how genetic algorithms can be used to model ecosystems. The major differences between Echo and standard genetic algorithms are: (1) there is no explicit fitness function, (2) individuals have local storage (i.e., they consist of more than their genome), (3) the genetic representation is based on a larger alphabet than binary strings, and (4) individuals always have a spatial location. In Echo, fitness evaluation takes place implicitly. That is, individuals in the population (called *agents*) are allowed to make copies of themselves anytime they acquire enough *resources* to replicate their genome. Different resources are modeled by different letters of the alphabet (say, A, B, C, D), and genomes are constructed out of those same letters. These resources can exist independently of the agent's genome, either free in the environment or stored internally by the agent. Agents acquire resources by interacting with other agents through trading relationships and combat. Echo thus relaxes the constraint that an explicit fitness function must return a numerical evaluation of each agent. This **endogenous fitness function** is much closer to the way fitness is assessed in natural settings. In addition to trade and combat, a third form of interaction between agents is mating. Mating provides opportunities for agents to exchange genetic material through crossover, thus creating hybrids. Mating, together with mutation, provides the mechanism for new types of agents to evolve.

Populations in Echo exist on a two-dimensional grid of sites, although other connection topologies are possible. Many agents can cohabit one site, and agents can migrate between sites. Each site is the source of certain renewable resources. On each time step of the simulation, a fixed amount of resources at a site becomes available to the agents located at that site. Different sites can produce different amounts of different resources. For example, one site might produce 10 As and 5 Bs each time step, and its neighbor might produce 5 As, 0 Bs, and 5 Cs. The idea is that an agent will do well (reproduce often) if it is located at a site whose renewable resources match well with its genomic makeup or if it can acquire the relevant resources from other agents at its site.

In preliminary simulations, the Echo system has demonstrated surprisingly complex behaviors, including something resembling a biological arms race (in which two competing species develop progressively more complex offensive and defensive strategies), functional dependencies among different species, trophic cascades, and sensitivity (in terms of the number of different phenotypes) to differing levels of renewable resources. Although the Echo system is still largely untested, it illustrates how the fundamental ideas of genetic algorithms can be incorporated into a system that captures important features of natural ecological systems.

14.4 Mathematical Analysis of Genetic Algorithms

Although there are many problems for which the genetic algorithm can evolve a good solution in reasonable time, there are also problems for which it is inappropriate (such as problems in which it is important to find the exact global optimum). It would be useful to have a mathematical characterization of how

the genetic algorithm works that is predictive. Research on this aspect of genetic algorithms has not produced definitive answers. The domains for which one is likely to choose an adaptive method such as the genetic algorithm are precisely those about which we typically have little analytical knowledge — they are complex, noisy, or dynamic (changing over time). These characteristics make it virtually impossible to predict with certainty how well a particular algorithm will perform on a particular problem instance, especially if the algorithm is stochastic, as is the case with the genetic algorithm. In spite of this difficulty, there are fairly extensive theories about how and why genetic algorithms work in idealized settings.

Analysis of genetic algorithms begins with the concept of a search space. The genetic algorithm can be viewed as a procedure for searching the space of all possible binary strings of fixed length l. Under this interpretation, the algorithm is searching for points in the l-dimensional space $\{0, 1\}^l$ that have high fitness. The search space is identical for all problems of the same size (same l), but the locations of good points will generally differ. The surface defined by the fitness of each point, together with the neighborhood relation imposed by the operators, is sometimes referred to as the **fitness landscape**. The longer the bit strings, corresponding to higher values of l, the larger the search space is, growing exponentially with the length of l. For problems with a sufficiently large l, only a small fraction of this size search space can be examined, and thus it is unreasonable to expect an algorithm to locate the global optimum in the space. A more reasonable goal is to search for good regions of the search space corresponding to regularities in the problem domain. Holland [1975] introduced the notion of a *schema* to explain how genetic algorithms search for regions of high fitness. Schemas are theoretical constructs used to explain the behavior of genetic algorithms, and are not processed directly by the algorithm. The following description of schema processing is excerpted from Forrest and Mitchell [1993b].

A schema is a template, defined over the alphabet $\{0, 1, *\}$, which describes a pattern of bit strings in the search space $\{0, 1\}^l$ (the set of bit strings of length l). For each of the l bit positions, the template either specifies the value at that position (1 or 0), or indicates by the symbol $*$ (referred to as don't care) that either value is allowed.

For example, the two strings A and B have several bits in common. We can use schemas to describe the patterns these two strings share:

A = 100111
B = 010011
**0*11
****11
0*
01

A bit string x that matches a schema s's pattern is said to be an *instance* of s; for example, A and B are both instances of the schemas just shown. In schemas, 1s and 0s are referred to as *defined bits*; the *order* of a schema is the number of defined bits in that schema, and the *defining length* of a schema is the distance between the leftmost and rightmost defined bits in the string. For example, the defining length of **0**1 is 3.

Schemas define hyperplanes in the search space $\{0, 1\}^l$. Figure 14.6 shows four hyperplanes, corresponding to the schemas 0****, 1****, *0***, and *1***. Any point in the space is simultaneously an instance

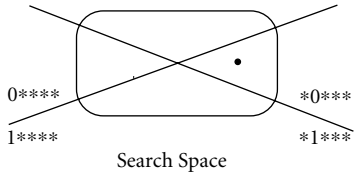


FIGURE 14.6 Schemas define hyperplanes in the search space. (From Forrest, S. and Mitchell, M. 1993b. *Machine Learning* 13:285–319. With permission.)

of two of these schemas. For example, the point shown in Figure 14.6 is an instance of both 1∗∗∗∗ and ∗0∗∗∗ (and also of 10∗∗∗).

The fitness of any bit string in the population gives some information about the average fitness of the 2^l different schemas of which it is an instance, and so an explicit evaluation of a population of M individual strings is also an implicit evaluation of a much larger number of schemas. This is referred to as implicit parallelism. At the explicit level the genetic algorithm searches through populations of bit strings, but the genetic algorithm's search can also be interpreted as an implicit schema sampling process. Feedback from the fitness function, combined with selection and recombination, biases the sampling procedure over time away from those schemas that give negative feedback (low average fitness) and toward those that give positive feedback (high average fitness). Ultimately, the search procedure should identify regularities, or patterns, in the environment that lead to high fitness. Because the space of possible patterns is larger than the space of possible individuals (3^l vs. 2^l), implicit parallelism is potentially advantageous.

An important theoretical result about genetic algorithms is the Schema Theorem [Holland 1975, Goldberg 1989], which states that the observed best schemas will on average be allocated an exponentially increasing number of samples in the next generation. Figure 14.7 illustrates the rapid convergence on fit schemas by the genetic algorithm. This strong convergence property of the genetic algorithm is a two-edged

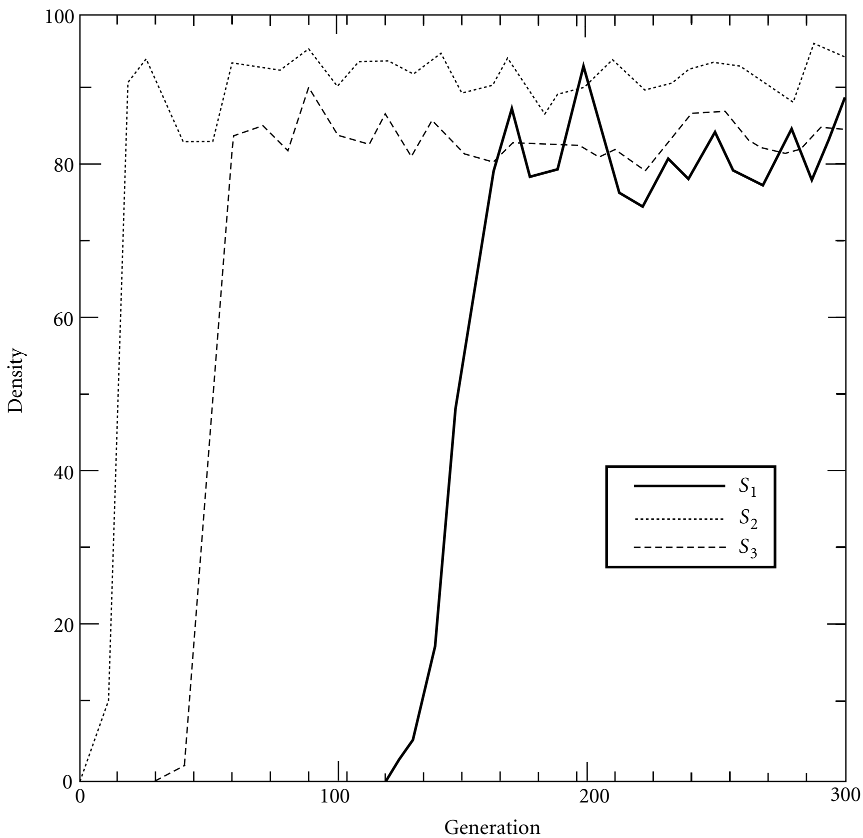


FIGURE 14.7 Schema frequencies over time. The graph plots schema frequencies in the population over time for three schemas:

s_1 = 1111111111111111∗∗∗;

s_2 = ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗1111111111111111∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗;

s_3 = ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗111111111111111111111111111111111.

The function plotted was a royal road function [Forrest and Mitchell 1993a] in which the optimum value is the string of all 1s. (From Forrest, S. 1993a. *Science* 261:872–878. With permission.)

sword. On the one hand, the fact that the genetic algorithm can close in on a fit part of the space very quickly is a powerful property; on the other hand, because the genetic algorithm always operates on finite-size populations, there is inherently some sampling error in the search, and in some cases the genetic algorithm can magnify a small sampling error, causing premature convergence on local optima.

According to the **building blocks hypothesis** [Holland 1975, Goldberg 1989], the genetic algorithm initially detects biases toward higher fitness in some low-order schemas (those with a small number of defined bits), and converges on this part of the search space. Over time, it detects biases in higher-order schemas by combining information from low-order schemas via crossover, and eventually it converges on a small region of the search space that has high fitness. The building blocks hypothesis states that this process is the source of the genetic algorithm's power as a search and optimization method. If this hypothesis about how genetic algorithms work is correct, then crossover is of primary importance, and it distinguishes genetic algorithms from other similar methods, such as simulated annealing and greedy algorithms. A number of authors have questioned the adequacy of the building blocks hypothesis as an explanation for how genetic algorithms work and there are several active research efforts studying schema processing in genetic algorithms. Nevertheless, the explanation of schemas and recombination that I have just described stands as the most common account of why genetic algorithms perform as they do.

There are several other approaches to analyzing mathematically the behavior of genetic algorithms: models developed for population genetics, algebraic models, signal-to-noise analysis, landscape analysis, statistical mechanics, Markov chains, and methods based on probably approximately correct (PAC) learning. This work extends and refines the schema analysis just given and in some cases challenges the claim that recombination through crossover is an important component of genetic algorithm performance. See Further Information section for additional reading.

14.5 Research Issues and Summary

The idea of using evolution to solve difficult problems and to model natural phenomena is promising. The genetic algorithms that I have described in this chapter are one of the first steps in this direction. Necessarily, they have abstracted out much of the richness of biology, and in the future we can expect a wide variety of evolutionary systems based on the principles of genetic algorithms but less closely tied to these specific mechanisms. For example, more elaborate representation techniques, including those that use complex genotype-to-phenotype mappings and increasing use of nonbinary alphabets can be expected. Endogenous fitness functions, similar to the one described for Echo, may become more common, as well as dynamic and coevolutionary fitness functions. More generally, biological mechanisms of all kinds will be incorporated into computational systems, including nervous systems, embryology, parasites, viruses, and immune systems.

From an algorithmic perspective, genetic algorithms join a broader class of stochastic methods for solving problems. An important area of future research is to understand carefully how these algorithms relate to one another and which algorithms are best for which problems. This is a difficult area in which to make progress. Controlled studies on idealized problems may have little relevance for practical problems, and benchmarks on specific problem instances may not apply to other instances. In spite of these impediments, this is an important direction for future research.

Acknowledgments

The author gratefully acknowledges support from the National Science Foundation (Grant IRI-9157644), the Office of Naval Research (Grant N00014-95-1-0364), ATR Human Information Processing Research Laboratories, and the Santa Fe Institute. Ron Hightower prepared Figure 14.2.

Significant portions of this chapter are excerpted with permission from Forrest, S. 1993. Genetic algorithms: principles of adaption applied to computation. *Science* 261 (Aug. 13):872–878. © 1993 American Association for the Advancement of Science.

Defining Terms

Building blocks hypothesis: The hypothesis that the genetic algorithm searches by first detecting biases toward higher fitness in some low-order schemas (those with a small number of defined bits) and converging on this part of the search space. Over time, it then detects biases in higher-order schemas by combining information from low-order schemas via crossover and eventually converges on a small region of the search space that has high fitness. The building blocks hypothesis states that this process is the source of the genetic algorithm's power as a search and optimization method [Holland 1975, Goldberg 1989].

Chromosome: A string of symbols (usually in bits) that contains the genetic information about an individual. The chromosome is interpreted by the fitness function to produce an evaluation of the individual's fitness.

Crossover: An operator for producing new individuals from two parent individuals. The operator works by exchanging substrings between the two individuals to obtain new offspring. In some cases, both offspring are passed to the new generation; in others, one is arbitrarily chosen to be passed on; the number of crossover points can be restricted to one per pair, two per pair, or N per pair.

Edge recombination: A special-purpose crossover operator designed to be used with permutation representations for sequencing problems. The edge-recombination operator attempts to preserve adjacencies between neighboring elements in the parent permutations [Starkweather et al. 1991].

Endogenous fitness function: Fitness is not assessed explicitly using a fitness function. Some other criterion for reproduction is adopted. For example, individuals might be required to accumulate enough internal resources to copy themselves before they can reproduce. Individuals who can gather resources efficiently would then reproduce frequently and their traits would become more prevalent in the population.

Fitness function: Each individual is tested empirically in an environment, receiving a numerical evaluation of its merit, assigned by a fitness function F. The environment can be almost anything — another computer simulation, interactions with other individuals in the population, actions in the physical world (by a robot for example), or a human's subjective judgment.

Fitness landscape: The surface defined by the fitness of each point in the search space, together with the neighborhood relation imposed by the operators.

Generation: One iteration, or time step, of the genetic algorithm. New generations can be produced either synchronously, so that the old generation is completely replaced (the time step model), or asynchronously, so that generations overlap. In the asynchronous case, generations are defined in terms of some fixed number of fitness-function evaluations.

Genetic programs: A form of genetic algorithm that uses a tree-based representation. The tree represents a program that can be evaluated, for example, an S-expression.

Genotype: The string of symbols, usually bits, used to represent an individual. Each bit position (set to 1 or 0) represents one gene. The term bit string in this context refers both to genotypes and to the individuals that they define.

Individuals: The structures that are evolved by the genetic algorithm. They can represent solutions to problems, strategies for playing games, visual images, or computer programs. Typically, each individual consists only of its genetic material, which is organized into one (haploid) chromosome.

Mutation: An operator for varying an individual. In mutation, individual bits are flipped probabilistically in individuals selected for reproduction. In representations other than bit strings, mutation is redefined to an appropriate smallest unit of change. For example, in permutation representations, mutation is often defined to be the swap of two neighboring elements in the permutation; in real-valued representations, mutation can be a creep operator that perturbs the real number up or down some small increment.

Schema: A theoretical construct used to explain the behavior of genetic algorithms. Schemas are not processed directly by the algorithm. Schemas are coordinate hyperplanes in the search space of strings.

Selection: Some individuals are more fit than others (better suited to their environment). These individuals have more offspring, that is, they are selected for reproduction. Selection is implemented by eliminating low-fitness individuals from the population, and inheritance is implemented by making multiple copies of high-fitness individuals.

References

Axelrod, R. 1986. An evolutionary approach to norms. *Am. Political Sci. Rev*. 80 (Dec).

Back, T. and Schwefel, H. P. 1993. An overview of evolutionary algorithms. *Evolutionary Comput*. 1:1–23.

Belew, R. K. and Booker, L. B., eds. 1991. *Proc. 4th Int. Conf. Genet. Algorithms*. July. Morgan Kaufmann, San Mateo, CA.

Booker, L. B., Riolo, R. L., and Holland, J. H. 1989. Learning and representation in classifier systems. *Art. Intelligence* 40:235–282.

Box, G. E. P. 1957. Evolutionary operation: a method for increasing industrial productivity. *J. R. Stat. Soc*. 6(2):81–101.

Davis, L., ed. 1991. *The Genetic Algorithms Handbook*. Van Nostrand Reinhold, New York.

DeJong, K. A. 1975. *An analysis of the behavior of a class of genetic adaptive systems*. Ph.D. thesis, University of Michigan, Ann Arbor.

DeJong, K. A. 1990a. Genetic-algorithm-based learning. *Machine Learning* 3:611–638.

DeJong, K. A. 1990b. Introduction to second special issue on genetic algorithms. *Machine Learning*. 5(4):351–353.

Eshelman, L. J., ed. 1995. *Proc. 6th Int. Conf. Genet. Algorithms*. Morgan Kaufmann, San Francisco.

Filho, J. L. R., Treleaven, P. C., and Alippi, C. 1994. Genetic-algorithm programming environments. *Computer* 27(6):28–45.

Fogel, L. J., Owens, A. J., and Walsh, M. J. 1966. *Artificial Intelligence Through Simulated Evolution*. Wiley, New York.

Fonseca, C. M. and Fleming, P. J. 1995. An overview of evolutionary algorithms in multiobjective optimization. *Evolutionary Comput*. 3(1):1–16.

Forrest, S. 1993a. Genetic algorithms: principles of adaptation applied to computation. *Science* 261:872–878.

Forrest, S., ed. 1993b. *Proc. Fifth Int. Conf. Genet. Algorithms*. Morgan Kaufmann, San Mateo, CA.

Forrest, S. and Mitchell, M. 1993a. Towards a stronger building-blocks hypothesis: effects of relative building-block fitness on ga performance. In *Foundations of Genetic Algorithms*, Vol. 2, L. D. Whitley, ed., pp. 109–126. Morgan Kaufmann, San Mateo, CA.

Forrest, S. and Mitchell, M. 1993b. What makes a problem hard for a genetic algorithm? Some anomalous results and their explanation. *Machine Learning* 13(2/3).

Goldberg, D. E. 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison Wesley, Reading, MA.

Grefenstette, J. J. 1985. *Proc. Int. Conf. Genet. Algorithms Appl*. NCARAI and Texas Instruments.

Grefenstette, J. J. 1987. *Proc. 2nd Int. Conf. Genet. Algorithms*. Lawrence Erlbaum, Hillsdale, NJ.

Hillis, W. D. 1990. Co-evolving parasites improve simulated evolution as an optimization procedure. *Physica D* 42:228–234.

Holland, J. H. 1962. Outline for a logical theory of adaptive systems. *J. ACM* 3:297–314.

Holland, J. H. 1975. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI; 1992. 2nd ed. MIT Press, Cambridge, MA.

Holland, J. H. 1992. Genetic algorithms. *Sci. Am*., pp. 114–116.

Holland, J. H. 1995. *Hidden Order: How Adaptation Builds Complexity*. Addison–Wesley, Reading, MA.

Holland, J. H., Holyoak, K. J., Nisbett, R. E., and Thagard, P. 1986. *Induction: Processes of Inference, Learning, and Discovery*. MIT Press, Cambridge, MA.

Koza, J. R. 1992. *Genetic Programming*. MIT Press, Cambridge, MA.

Männer, R. and Manderick, B., eds. 1992. *Parallel Problem Solving From Nature 2*. North Holland, Amsterdam.
Mitchell, M. 1996. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA.
Mitchell, M. and Forrest, S. 1994. Genetic algorithms and artificial life. *Artif. Life* 1(3):267–289; reprinted 1995. In *Artificial Life: An Overview*, C. G. Langton, ed. MIT Press, Cambridge, MA.
Mühlenbein, H., Gorges-Schleuter, M., and Kramer, O. 1988. *Parallel Comput*. 6:65–88.
Rawlins, G., ed. 1991. *Foundations of Genetic Algorithms*. Morgan Kaufmann, San Mateo, CA.
Schaffer, J. D., ed. 1989. *Proc. 3rd Int. Conf. Genet. Algorithms*. Morgan Kaufmann, San Mateo, CA.
Schaffer, J. D., Whitley, D., and Eshelman, L. J. 1992. Combinations of genetic algorithms and neural networks: a survey of the state of the art. In *Int. Workshop Combinations Genet. Algorithms Neural Networks*, L. D. Whitley and J. D. Schaffer, eds., pp. 1–37. IEEE Computer Society Press, Los Alamitos, CA.
Schwefel, H. P. and Männer, R., eds. 1990. Parallel problem solving from nature. *Lecture Notes in Computer Science*. Springer–Verlag, Berlin.
Srinivas, M. and Patnaik, L. M. 1994. Genetic algorithms: a survey. *Computer* 27(6):17–27.
Starkweather, T., McDaniel, S., Mathias, K., Whitley, D., and Whitley, C. 1991. A comparison of genetic sequencing operators. In *4th Int. Conf. Genet. Algorithms*, R. K. Belew and L. B. Booker, eds., pp. 69–76. Morgan Kaufmann, Los Altos, CA.
Thomas, E. V. 1993. Frequency Selection Using Genetic Algorithms. *Sandia National Lab. Tech. Rep*. SAND93-0010, Albuquerque, NM.
Whitley, L. D., ed. 1993. *Foundations of Genetic Algorithms 2*. Morgan Kaufmann, San Mateo, CA.
Whitley, L. D. and Vose, M., eds. 1995. *Foundations of Genetic Algorithms 3*. Morgan Kaufmann, San Francisco.

Further Information

Review articles on genetic algorithms include Booker et al. [1989], Holland [1992], Forrest [1993a], Mitchell and Forrest [1994], Srinivas and Patnaik [1994] and Filho et al. [1994]. Books that describe the theory and practice of genetic algorithms in greater detail include Holland [1975], Goldberg [1989], Davis [1991], Koza [1992], Holland et al. [1986], and Mitchell [1996]. Holland [1975] was the first book-length description of genetic algorithms, and it contains much of the original insight about the power and breadth of adaptive algorithms. The 1992 reprinting contains interesting updates by Holland. However, Goldberg [1989], Davis [1991], and Mitchell [1996] are more accessible introductions to the basic concepts and implementation issues. Koza [1992] describes genetic programming and Holland et al. [1986] discuss the relevance of genetic algorithms to cognitive modeling.

Current research on genetic algorithms is reported many places, including the Proceedings of the International Conference on Genetic Algorithms [Grefenstette 1985, 1987, Schaffer 1989, Belew and Booker 1991, Forrest 1993b, Eshelman 1995], the proceedings of conferences on Parallel Problem Solving from Nature [Schwefel and Männer 1990, Männer and Manderick 1992], and the workshops on Foundations of Genetic Algorithms [Rawlins 1991, Whitley 1993, Whitley and Vose 1995]. Finally, the artificial-life literature contains many interesting papers about genetic algorithms.

There are several archival journals that publish articles about genetic algorithms. These include *Evolutionary Computation* (a journal devoted to GAs), *Complex Systems*, *Machine Learning*, *Adaptive Behavior*, and *Artificial Life*.

Information about genetic algorithms activities, public domain packages, etc., is maintained through the WWW at URL http://www.aic.nrl.navy.mil/galist/ or through anonymous ftp at ftp.aic.nrl.navy.mil [192.26.18.68] in/pub/galist.

15

Combinatorial Optimization

Vijay Chandru
Indian Institute of Science

M. R. Rao
Indian Institute of Management

15.1 Introduction

Bin packing, routing, scheduling, layout, and network design are generic examples of combinatorial optimization problems that often arise in computer engineering and decision support. Unfortunately, almost all interesting generic classes of combinatorial optimization problems are $\mathcal{NP}$-hard. The scale at which these problems arise in applications and the explosive exponential complexity of the search spaces preclude the use of simplistic enumeration and search techniques. Despite the worst-case intractability of combinatorial optimization, in practice we are able to solve many large problems and often with off-the-shelf software. Effective software for combinatorial optimization is usually problem specific and based on sophisticated algorithms that combine approximation methods with search schemes and that exploit mathematical (and not just syntactic) structure in the problem at hand.

Multidisciplinary interests in combinatorial optimization have led to several fairly distinct paradigms in the development of this subject. Each paradigm may be thought of as a particular combination of a *representation scheme* and a *methodology* (see Table 15.1). The most established of these, the **integer programming** paradigm, uses implicit algebraic forms (linear constraints) to represent combinatorial

TABLE 15.1 Paradigms in Combinatorial Optimization

Paradigm	Representation	Methodology
Integer programming	Linear constraints, Linear objective, Integer variables	Linear programming and extensions
Search	State space, Discrete control	Dynamic programming, $\mathcal{A}^*$
Local improvement	Neighborhoods Fitness functions	Hill climbing, Simulated annealing, Tabu search, Genetic algorithms
Constraint logic programming	Horn rules	Resolution, constraint solvers

optimization and **linear programming** and its extensions as the workhorses in the design of the solution algorithms. It is this paradigm that forms the central theme of this chapter.

Other well known paradigms in combinatorial optimization are **search, local improvement,** and **constraint logic programming.** Search uses state-space representations and partial enumeration techniques such as $\mathcal{A}^*$ and dynamic programming. Local improvement requires only a representation of neighborhood in the solution space, and methodologies vary from simple hill climbing to the more sophisticated techniques of simulated annealing, tabu search, and genetic algorithms. Constraint logic programming uses the syntax of Horn rules to represent combinatorial optimization problems and uses resolution to orchestrate the solution of these problems with the use of domain-specific constraint solvers. Whereas integer programming was developed and nurtured by the mathematical programming community, these other paradigms have been popularized by the artificial intelligence community.

An abstract formulation of combinatorial optimization is

$$\text{(CO)} \quad \min\{f(I) : I \in \mathcal{I}\}$$

where $\mathcal{I}$ is a collection of subsets of a finite ground set $E = \{e_1, e_2, \ldots, e_n\}$ and f is a criterion (objective) function that maps 2^E (the power set of E) to the reals. A **mixed integer linear program** (MILP) is of the form

$$\text{(MILP)} \quad \min_{x \in \Re^n}\{\mathbf{cx} : A\mathbf{x} \geq \mathbf{b},\ \mathbf{x}_j \text{ integer } \forall\ j \in J\}$$

which seeks to minimize a linear function of the decision vector $\mathbf{x}$ subject to linear inequality constraints and the requirement that a subset of the decision variables is integer valued. This model captures many variants. If $J = \{1, 2, \ldots, n\}$, we say that the integer program is *pure*, and *mixed* otherwise. Linear equations and bounds on the variables can be easily accommodated in the inequality constraints. Notice that by adding in inequalities of the form $0 \leq \mathbf{x}_j \leq 1$ for a $j \in J$ we have forced $\mathbf{x}_j$ to take value 0 or 1. It is such Boolean variables that help capture combinatorial optimization problems as special cases of MILP.

Pure integer programming with variables that take arbitrary integer values is a class which has strong connections to number theory and particularly the geometry of numbers and Presburgher arithmetic. Although this is a fascinating subject with important applications in cryptography, in the interests of brevity we shall largely restrict our attention to MILP where the integer variables are Boolean.

The fact that mixed integer linear programs subsume combinatorial optimization problems follows from two simple observations. The first is that a collection $\mathcal{I}$ of subsets of a finite ground set E can always be represented by a corresponding collection of incidence vectors, which are $\{0, 1\}$-vectors in $\Re^E$. Further, arbitrary nonlinear functions can be represented via piecewise linear approximations by using linear constraints and mixed variables (continuous and Boolean).

The next section contains a primer on linear inequalities, polyhedra, and linear programming. These are the tools we will need to analyze and solve integer programs. Section 15.4, is a testimony to the earlier

cryptic comments on how integer programs model combinatorial optimization problems. In addition to working a number of examples of such integer programming formulations, we shall also review a formal representation theory of (Boolean) mixed integer linear programs.

With any mixed integer program we associate a **linear programming relaxation** obtained by simply ignoring the integrality restrictions on the variables. The point being, of course, that we have polynomial-time (and practical) algorithms for solving linear programs. Thus, the linear programming relaxation of (MILP) is given by

$$\text{(LP)} \quad \min_{x \in \Re^n} \{\mathbf{cx} : A\mathbf{x} \geq \mathbf{b}\}$$

The thesis underlying the integer linear programming approach to combinatorial optimization is that this linear programming relaxation retains enough of the structure of the combinatorial optimization problem to be a useful weak representation. In Section 15.5 we shall take a closer look at this thesis in that we shall encounter special structures for which this relaxation is *tight*. For general integer programs, there are several alternative schemes for generating linear programming relaxations with varying qualities of approximation. A general principle is that we often need to disaggregate integer formulations to obtain higher quality linear programming relaxations. To solve such huge linear programs we need specialized techniques of large-scale linear programming. These aspects will be the content of Section 15.3.

The reader should note that the focus in this chapter is on solving hard combinatorial optimization problems. We catalog the special structures in integer programs that lead to tight linear programming relaxations (Section 15.5) and hence to polynomial-time algorithms. These include structures such as network flows, matching, and matroid optimization problems. Many hard problems actually have pieces of these nice structures embedded in them. Practitioners of combinatorial optimization have always used insights from special structures to devise strategies for hard problems.

The computational art of integer programming rests on useful interplays between search methodologies and linear programming relaxations. The paradigms of branch and bound and branch and cut are the two enormously effective partial enumeration schemes that have evolved at this interface. These will be discussed in Section 15.6. It may be noted that all general purpose integer programming software available today uses one or both of these paradigms.

The inherent complexity of integer linear programming has led to a long-standing research program in approximation methods for these problems. Linear programming relaxation and Lagrangian relaxation are two general approximation schemes that have been the real workhorses of computational practice. Primal–dual strategies and semidefinite relaxations are two recent entrants that appear to be very promising. Section 15.7 of this chapter reviews these developments in the approximation of combinatorial optimization problems.

We conclude the chapter with brief comments on future prospects in combinatorial optimization from the algebraic modeling perspective.

15.2 A Primer on Linear Programming

Polyhedral combinatorics is the study of embeddings of combinatorial structures in Euclidean space and their algebraic representations. We will make extensive use of some standard terminology from polyhedral theory. Definitions of terms not given in the brief review below can be found in Nemhauser and Wolsey [1988].

A (convex) **polyhedron** in $\Re^n$ can be algebraically defined in two ways. The first and more straightforward definition is the *implicit* representation of a polyhedron in $\Re^n$ as the solution set to a finite system of linear inequalities in n variables. A single linear inequality $\mathbf{ax} \leq a_0$; $\mathbf{a} \neq \mathbf{0}$ defines a *half-space* of $\Re^n$. Therefore, geometrically a polyhedron is the intersection set of a finite number of half-spaces.

A *polytope* is a bounded polyhedron. Every polytope is the convex closure of a finite set of points. Given a set of points whose convex combinations generate a polytope, we have an explicit or *parametric* algebraic representation of it. A *polyhedral cone* is the solution set of a system of homogeneous linear inequalities.

Every (polyhedral) cone is the conical or positive closure of a finite set of vectors. These generators of the cone provide a parametric representation of the cone. And finally, a polyhedron can be alternatively defined as the Minkowski sum of a polytope and a cone. Moving from one representation of any of these polyhedral objects to another defines the essence of the computational burden of polyhedral combinatorics. This is particularly true if we are interested in *minimal* representations.

A set of points $\mathbf{x}^1, \ldots, \mathbf{x}^m$ is *affinely independent* if the unique solution of $\sum_{i=1}^m \lambda_i \mathbf{x}^i = 0$, $\sum_{i=1}^m \lambda_i = 0$ is $\lambda_i = 0$ for $i = 1, \ldots, m$. Note that the maximum number of affinely independent points in $\Re^n$ is $n+1$. A polyhedron P is of *dimension* k, $\dim P = k$, if the maximum number of affinely independent points in P is $k+1$. A polyhedron $P \subseteq \Re^n$ of dimension n is called *full dimensional*. An inequality $\mathbf{ax} \leq a_0$ is called *valid* for a polyhedron P if it is satisfied by all $\mathbf{x}$ in P. It is called *supporting* if in addition there is an $\bar{\mathbf{x}}$ in P that satisfies $\mathbf{a}\bar{\mathbf{x}} = a_0$. A *face* of the polyhedron is the set of all $\mathbf{x}$ in P that also satisfies a valid inequality as an equality. In general, many valid inequalities might represent the same face. Faces other than P itself are called *proper*. A *facet* of P is a maximal nonempty and proper face. A facet is then a face of P with a dimension of $\dim P - 1$. A face of dimension zero, i.e., a point v in P that is a face by itself, is called an **extreme point** of P. The extreme points are the elements of P that cannot be expressed as a strict convex combination of two distinct points in P. For a full-dimensional polyhedron, the valid inequality representing a facet is unique up to multiplication by a positive scalar, and facet-inducing inequalities give a minimal implicit representation of the polyhedron. Extreme points, on the other hand, give rise to minimal parametric representations of polytopes.

The two fundamental problems of linear programming (which are polynomially equivalent) follow:

- *Solvability.* This is the problem of checking if a system of linear constraints on real (rational) variables is solvable or not. Geometrically, we have to check if a polyhedron, defined by such constraints, is nonempty.
- *Optimization.* This is the problem (LP) of optimizing a linear objective function over a polyhedron described by a system of linear constraints.

Building on polarity in cones and polyhedra, duality in linear programming is a fundamental concept which is related to both the complexity of linear programming and to the design of algorithms for solvability and optimization. We will encounter the solvability version of duality (called Farkas Lemma) while discussing the Fourier elimination technique subsequently. Here we will state the main duality results for optimization. If we take the *primal* linear program to be

$$(P) \quad \min_{x \in \Re^n} \{\mathbf{cx} : A\mathbf{x} \geq \mathbf{b}\}$$

there is an associated *dual* linear program

$$(D) \quad \max_{y \in \Re^m} \{\mathbf{b}^T\mathbf{y} : A^T\mathbf{y} = \mathbf{c}^T, \mathbf{y} \geq \mathbf{0}\}$$

and the two problems satisfy the following:

1. For any $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$ feasible in (P) and (D) (i.e., they satisfy the respective constraints), we have $\mathbf{c}\hat{\mathbf{x}} \geq \mathbf{b}^T\hat{\mathbf{y}}$ (weak duality). Consequently, (P) has a finite optimal solution if and only if (D) does.
2. The pair $\mathbf{x}^*$ and $\mathbf{y}^*$ are optimal solutions for (P) and (D), respectively, if and only if $\mathbf{x}^*$ and $\mathbf{y}^*$ are feasible in (P) and (D) (i.e., they satisfy the respective constraints) and $\mathbf{cx}^* = \mathbf{b}^T\mathbf{y}^*$ (strong duality).
3. The pair $\mathbf{x}^*$ and $\mathbf{y}^*$ are optimal solutions for (P) and (D), respectively, if and only if $\mathbf{x}^*$ and $\mathbf{y}^*$ are feasible in (P) and (D) (i.e., they satisfy the respective constraints) and $(A\mathbf{x}^* - \mathbf{b})^T\mathbf{y}^* = 0$ (complementary slackness).

The strong duality condition gives us a good stopping criterion for optimization algorithms. The complementary slackness condition, on the other hand, gives us a constructive tool for moving from dual

to primal solutions and vice versa. The weak duality condition gives us a technique for obtaining lower bounds for minimization problems and upper bounds for maximization problems.

Note that the properties just given have been stated for linear programs in a particular form. The reader should be able to check that if, for example, the primal is of the form

$$(P') \quad \min_{\mathbf{x}\in\Re^n}\{\mathbf{cx} : A\mathbf{x} = \mathbf{b},\ \mathbf{x} \geq \mathbf{0}\}$$

then the corresponding dual will have the form

$$(D') \quad \max_{\mathbf{y}\in\Re^m}\{\mathbf{b}^T\mathbf{y} : A^T\mathbf{y} \leq \mathbf{c}^T\}$$

The tricks needed for seeing this are that any equation can be written as two inequalities, an unrestricted variable can be substituted by the difference of two nonnegatively constrained variables, and an inequality can be treated as an equality by adding a nonnegatively constrained variable to the lesser side. Using these tricks, the reader could also check that duality in linear programming is involutory (i.e., the dual of the dual is the primal).

15.2.1 Algorithms for Linear Programming

We will now take a quick tour of some algorithms for linear programming. We start with the classical technique of Fourier, which is interesting because of its really simple syntactic specification. It leads to simple proofs of the duality principle of linear programming (solvability) that has been alluded to. We will then review the simplex method of linear programming, a method that has been finely honed over almost five decades. We will spend some time with the ellipsoid method and, in particular, with the polynomial equivalence of solvability (optimization) and separation problems, for this aspect of the ellipsoid method has had a major impact on the identification of many tractable classes of combinatorial optimization problems. We conclude the primer with a description of Karmarkar's [1984] breakthrough, which was an important landmark in the brief history of linear programming. A noteworthy role of interior point methods has been to make practical the theoretical demonstrations of tractability of various aspects of linear programming, including solvability and optimization, that were provided via the ellipsoid method.

15.2.1.1 Fourier's Scheme for Linear Inequalities

Constraint systems of linear *inequalities* of the form $A\mathbf{x} \leq \mathbf{b}$, where A is an $m \times n$ matrix of real numbers, are widely used in mathematical models. Testing the solvability of such a system is equivalent to linear programming.

Suppose we wish to eliminate the first variable $\mathbf{x}_1$ from the system $A\mathbf{x} \leq \mathbf{b}$. Let us denote

$$I^+ = \{i : A_{i1} > 0\} \qquad I^- = \{i : A_{i1} < 0\} \qquad I^0 = \{i : A_{i1} = 0\}$$

Our goal is to create an equivalent system of linear inequalities $\tilde{A}\tilde{\mathbf{x}} \leq \tilde{\mathbf{b}}$ defined on the variables $\tilde{\mathbf{x}} = (\mathbf{x}_2, \mathbf{x}_3, \ldots, \mathbf{x}_n)$:

- If I^+ is empty then we can simply delete all the inequalities with indices in I^- since they can be trivially satisfied by choosing a large enough value for $\mathbf{x}_1$. Similarly, if I^- is empty we can discard all inequalities in I^+.
- For each $k \in I^+$, $l \in I^-$ we add $-A_{l1}$ times the inequality $A_k\mathbf{x} \leq \mathbf{b}_k$ to A_{k1} times $A_l\mathbf{x} \leq \mathbf{b}_l$. In these new inequalities the coefficient of $\mathbf{x}_1$ is wiped out, that is, $\mathbf{x}_1$ is eliminated. Add these new inequalities to those already in I^0.
- The inequalities $\{\tilde{A}_{i1}\tilde{\mathbf{x}} \leq \tilde{\mathbf{b}}_i\}$ for all $i \in I^0$ represent the equivalent system on the variables $\tilde{\mathbf{x}} = (\mathbf{x}_2, \mathbf{x}_3, \ldots, \mathbf{x}_n)$.

Repeat this construction with $\tilde{A}\tilde{\mathbf{x}} \leq \tilde{\mathbf{b}}$ to eliminate $\mathbf{x}_2$ and so on until all variables are eliminated. If the resulting $\tilde{\mathbf{b}}$ (after eliminating $\mathbf{x}_n$) is nonnegative, we declare the original (and intermediate) inequality systems as being consistent. Otherwise,* $\tilde{\mathbf{b}} \not\geq 0$ and we declare the system inconsistent.

As an illustration of the power of elimination as a tool for theorem proving, we show now that Farkas Lemma is a simple consequence of the correctness of Fourier elimination. The lemma gives a direct proof that solvability of linear inequalities is in $\mathcal{NP} \bigcap co\mathcal{NP}$.

FARKAS LEMMA 15.1 (Duality in Linear Programming: Solvability). *Exactly one of the alternatives*

$$I. \quad \exists\, \mathbf{x} \in \Re^n \,:\, A\mathbf{x} \leq \mathbf{b}$$

$$II. \quad \exists\, \mathbf{y} \in \Re^m_+ \,:\, \mathbf{y}^t A = \mathbf{0},\, \mathbf{y}^t \mathbf{b} < 0$$

is true for any given real matrices $A, \mathbf{b}$.

***Proof* 15.1** Let us analyze the case when Fourier elimination provides a proof of the inconsistency of a given linear inequality system $A\mathbf{x} \leq \mathbf{b}$. The method clearly converts the given system into $RA\mathbf{x} \leq R\mathbf{b}$ where RA is zero and $R\mathbf{b}$ has at least one negative component. Therefore, there is some row of R, say, $\mathbf{r}$, such that $\mathbf{r}A = \mathbf{0}$ and $\mathbf{r}\mathbf{b} < 0$. Thus $\neg I$ implies II. It is easy to see that I and II cannot both be true for fixed $A, \mathbf{b}$. □

In general, the Fourier elimination method is quite inefficient. Let k be any positive integer and n the number of variables be $2^k + k + 2$. If the input inequalities have left-hand sides of the form $\pm\mathbf{x}_r \pm \mathbf{x}_s \pm \mathbf{x}_t$ for all possible $1 \leq r < s < t \leq n$, it is easy to prove by induction that after k variables are eliminated, by Fourier's method, we would have at least $2^{n/2}$ inequalities. The method is therefore exponential in the worst case, and the explosion in the number of inequalities has been noted, in practice as well, on a wide variety of problems. We will discuss the central idea of minimal generators of the projection cone that results in a much improved elimination method.

First, let us identify the set of variables to be eliminated. Let the input system be of the form

$$P = \{(\mathbf{x}, \mathbf{u}) \in \Re^{n_1+n_2} \mid A\mathbf{x} + B\mathbf{u} \leq \mathbf{b}\}$$

where $\mathbf{u}$ is the set to be eliminated. The projection of P onto $\mathbf{x}$ or equivalently the effect of eliminating the $\mathbf{u}$ variables is

$$P_{\mathbf{x}} = \{\mathbf{x} \in \Re^{n_1} \mid \exists \mathbf{u} \in \Re^{n_2} \text{ such that } A\mathbf{x} + B\mathbf{u} \leq \mathbf{b}\}$$

Now W, the *projection cone* of P, is given by

$$W = \{\mathbf{w} \in \Re^m \mid \mathbf{w}B = \mathbf{0},\, \mathbf{w} \geq \mathbf{0}\}$$

A simple application of Farkas Lemma yields a description of $P_{\mathbf{x}}$ in terms of W.

PROJECTION LEMMA 15.2 *Let G be any set of generators (e.g., the set of extreme rays) of the cone W. Then $P_{\mathbf{x}} = \{\mathbf{x} \in \Re^{n_1} | (\mathbf{g}A)x \leq \mathbf{g}b\, \forall\, \mathbf{g} \in G\}$.*

The lemma, sometimes attributed to Černikov [1961], reduces the computation of $P_{\mathbf{x}}$ to enumerating the extreme rays of the cone W or equivalently the extreme points of the polytope $W \cap \{\mathbf{w} \in \Re^m | \sum_{i=1}^m \mathbf{w}_i = 1\}$.

*Note that the final $\tilde{\mathbf{b}}$ may not be defined if all of the inequalities are deleted by the monotone sign condition of the first step of the construction described. In such a situation, we declare the system $A\mathbf{x} \leq \mathbf{b}$ *strongly consistent* since it is consistent for any choice of $\mathbf{b}$ in $\Re^m$. To avoid making repeated references to this exceptional situation, let us simply assume that it does not occur. The reader is urged to verify that this assumption is indeed benign.

15.2.1.2 Simplex Method

Consider a polyhedron $\mathcal{K} = \{\mathbf{x} \in \Re^n : A\mathbf{x} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}\}$. Now $\mathcal{K}$ cannot contain an infinite (in both directions) line since it is lying within the nonnegative orthant of $\Re^n$. Such a polyhedron is called a *pointed* polyhedron. Given a pointed polyhedron $\mathcal{K}$ we observe the following:

- If $\mathcal{K} \neq \emptyset$, then $\mathcal{K}$ has at least one extreme point.
- If $\min\{\mathbf{cx} : A\mathbf{x} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}\}$ has an optimal solution, then it has an optimal extreme point solution.

These observations together are sometimes called the fundamental theorem of linear programming since they suggest simple finite tests for both solvability and optimization. To generate all extreme points of $\mathcal{K}$, in order to find an optimal solution, is an impractical idea. However, we may try to run a partial search of the space of extreme points for an optimal solution. A simple local improvement search strategy of moving from extreme point to adjacent extreme point until we get to a local optimum is nothing but the simplex method of linear programming. The local optimum also turns out to be a global optimum because of the convexity of the polyhedron $\mathcal{K}$ and the linearity of the objective function $\mathbf{cx}$.

The simplex method walks along edge paths on the combinatorial graph structure defined by the boundary of convex polyhedra. Since these graphs are quite dense (Balinski's theorem states that the graph of d-dimensional polyhedron must be d-connected [Ziegler 1995]) and possibly large (the Lower Bound Theorem states that the number of vertices can be exponential in the dimension [Ziegler 1995]), it is indeed somewhat of a miracle that it manages to get to an optimal extreme point as quickly as it does. Empirical and probabilistic analyses indicate that the number of iterations of the simplex method is just slightly more than linear in the dimension of the primal polyhedron. However, there is no known variant of the simplex method with a worst-case polynomial guarantee on the number of iterations. Even a polynomial bound on the diameter of polyhedral graphs is not known.

Procedure 15.1 Primal Simplex ($\mathcal{K}$, $\boldsymbol{c}$):

0. **Initialize:**

   $\mathbf{x}_0$:= an extreme point of $\mathcal{K}$
   $k := 0$

1. **Iterative step:**

   do
   If for all edge directions $\mathcal{D}_k$ at $\mathbf{x}_k$, the objective function is nondecreasing, i.e.,

   $$\mathbf{cd} \geq 0 \quad \forall\, \mathbf{d} \in \mathcal{D}_k$$

   then exit and return optimal $\mathbf{x}_k$.
   Else pick some $\mathbf{d}_k$ in $\mathcal{D}_k$ such that $\mathbf{cd}_k < 0$.
   If $\mathbf{d}_k \geq 0$ **then** declare the linear program unbounded in objective value and exit.
   Else $\mathbf{x}_{k+1} := \mathbf{x}_k + \theta_k * \mathbf{d}_k$, where

   $$\theta_k = \max\{\theta : \mathbf{x}_k + \theta * \mathbf{d}_k \geq 0\}$$

   $k := k + 1$
   od

2. **End**

Remark 15.1 In the initialization step, we assumed that an extreme point $\mathbf{x}_0$ of the polyhedron $\mathcal{K}$ is available. This also assumes that the solvability of the constraints defining $\mathcal{K}$ has been established. These

assumptions are reasonable since we can formulate the solvability problem as an optimization problem, with a self-evident extreme point, whose optimal solution either establishes unsolvability of $A\mathbf{x} = \mathbf{b}$, $\mathbf{x} \geq \mathbf{0}$ or provides an extreme point of $\mathcal{K}$. Such an optimization problem is usually called a phase I model. The point being, of course, that the simplex method, as just described, can be invoked on the phase I model and, if successful, can be invoked once again to carry out the intended minimization of $\mathbf{cx}$. There are several different formulations of the phase I model that have been advocated. Here is one:

$$\min\{\nu_0 \ : \ A\mathbf{x} + \mathbf{b}\nu_0 = \mathbf{b},\ \mathbf{x} \geq \mathbf{0},\ \nu_0 \geq 0\}$$

The solution $(\mathbf{x},\ \nu_0)^T = (0, \ldots, 0, 1)$ is a self-evident extreme point and $\nu_0 = 0$ at an optimal solution of this model is a necessary and sufficient condition for the solvability of $A\mathbf{x} = \mathbf{b}$, $\mathbf{x} \geq \mathbf{0}$.

Remark 15.2 The scheme for generating improving edge directions uses an algebraic representation of the extreme points as certain bases, called feasible bases, of the vector space generated by the columns of the matrix A. It is possible to have linear programs for which an extreme point is geometrically overdetermined (degenerate), i.e., there are more than d facets of $\mathcal{K}$ that contain the extreme point, where d is the dimension of $\mathcal{K}$. In such a situation, there would be several feasible bases corresponding to the same extreme point. When this happens, the linear program is said to be *primal degenerate*.

Remark 15.3 There are two sources of nondeterminism in the primal simplex procedure. The first involves the choice of edge direction $\mathbf{d}_k$ made in step 1. At a typical iteration there may be many edge directions that are improving in the sense that $\mathbf{cd}_k < 0$. Dantzig's rule, the maximum improvement rule, and steepest descent rule are some of the many rules that have been used to make the choice of edge direction in the simplex method. There is, unfortunately, no clearly dominant rule and successful codes exploit the empirical and analytic insights that have been gained over the years to resolve the edge selection nondeterminism in simplex methods. The second source of nondeterminism arises from degeneracy. When there are multiple feasible bases corresponding to an extreme point, the simplex method has to pivot from basis to adjacent basis by picking an entering basic variable (a pseudoEdge direction) and by dropping one of the old ones. A wrong choice of the leaving variables may lead to cycling in the sequence of feasible bases generated at this extreme point. Cycling is a serious problem when linear programs are highly degenerate as in the case of linear relaxations of many combinatorial optimization problems. The lexicographic rule (perturbation rule) for the choice of leaving variables in the simplex method is a provably finite method (i.e., all cycles are broken). A clever method proposed by Bland (cf. Schrijver [1986]) preorders the rows and columns of the matrix A. In the case of nondeterminism in either entering or leaving variable choices, Bland's rule just picks the lowest index candidate. All cycles are avoided by this rule also.

The simplex method has been the veritable workhorse of linear programming for four decades now. However, as already noted, we do not know of a simplex method that has worst-case bounds that are polynomial. In fact, Klee and Minty exploited the sensitivity of the original simplex method of Dantzig, to projective scaling of the data, and constructed exponential examples for it. Recently, Spielman and Tang [2001] introduced the concept of smoothed analysis and smoothed complexity of algorithms, which is a hybrid of worst-case and average-case analysis of algorithms. Essentially, this involves the study of performance of algorithms under small random Gaussian perturbations of the coefficients of the constraint matrix. The authors show that a variant of the simplex algorithm, known as the *shadow vertex simplex algorithm* (Gass and Saaty [1955]) has polynomial smoothed complexity.

The ellipsoid method of Shor [1970] was devised to overcome poor scaling in convex programming problems and, therefore, turned out to be the natural choice of an algorithm to first establish polynomial-time solvability of linear programming. Later Karmarkar [1984] took care of both projection and scaling simultaneously and arrived at a superior algorithm.

15.2.1.3 The Ellipsoid Algorithm

The ellipsoid algorithm of Shor [1970] gained prominence in the late 1970s when Haĉijan [1979] (pronounced Khachiyan) showed that this convex programming method specializes to a polynomial-time algorithm for linear programming problems. This theoretical breakthrough naturally led to intense study of this method and its properties. The survey paper by Bland et al. [1981] and the monograph by Akgül [1984] attest to this fact. The direct theoretical consequences for combinatorial optimization problems was independently documented by Padberg and Rao [1981], Karp and Papadimitriou [1982], and Grötschel et al. [1988]. The ability of this method to implicitly handle linear programs with an exponential list of constraints and maintain polynomial-time convergence is a characteristic that is the key to its applications in combinatorial optimization. For an elegant treatment of the many deep theoretical consequences of the ellipsoid algorithm, the reader is directed to the monograph by Lovász [1986] and the book by Grötschel et al. [1988].

Computational experience with the ellipsoid algorithm, however, showed a disappointing gap between the theoretical promise and practical efficiency of this method in the solution of linear programming problems. Dense matrix computations as well as the slow average-case convergence properties are the reasons most often cited for this behavior of the ellipsoid algorithm. On the positive side though, it has been noted (cf. Ecker and Kupferschmid [1983]) that the ellipsoid method is competitive with the best known algorithms for (nonlinear) convex programming problems.

Let us consider the problem of testing if a polyhedron $\mathcal{Q} \in \Re^d$, defined by linear inequalities, is nonempty. For technical reasons let us assume that $\mathcal{Q}$ is rational, i.e., all extreme points and rays of $\mathcal{Q}$ are rational vectors or, equivalently, that all inequalities in some description of $\mathcal{Q}$ involve only rational coefficients. The ellipsoid method does not require the linear inequalities describing $\mathcal{Q}$ to be explicitly specified. It suffices to have an oracle representation of $\mathcal{Q}$. Several different types of oracles can be used in conjunction with the ellipsoid method (Karp and Papadimitriou [1982], Padberg and Rao [1981], Grötschel et al. [1988]). We will use the *strong separation oracle:*

Oracle: **Strong Separation**($\mathcal{Q}$, $\mathbf{y}$)

> Given a vector $\mathbf{y} \in \Re^d$, decide whether $\mathbf{y} \in \mathcal{Q}$, and if not find a hyperplane that separates $\mathbf{y}$ from $\mathcal{Q}$; more precisely, find a vector $\mathbf{c} \in \Re^d$ such that $\mathbf{c}^T\mathbf{y} < \min\{\mathbf{c}^T\mathbf{x} \mid \mathbf{x} \in \mathcal{Q}\}$.

The ellipsoid algorithm initially chooses an ellipsoid large enough to contain a part of the polyhedron $\mathcal{Q}$ if it is nonempty. This is easily accomplished because we know that if $\mathcal{Q}$ is nonempty then it has a rational solution whose (binary encoding) length is bounded by a polynomial function of the length of the largest coefficient in the linear program and the dimension of the space.

The center of the ellipsoid is a feasible point if the separation oracle tells us so. In this case, the algorithm terminates with the coordinates of the center as a solution. Otherwise, the separation oracle outputs an inequality that separates the center point of the ellipsoid from the polyhedron $\mathcal{Q}$. We translate the hyperplane defined by this inequality to the center point. The hyperplane slices the ellipsoid into two halves, one of which can be discarded. The algorithm now creates a new ellipsoid that is the minimum volume ellipsoid containing the remaining half of the old one. The algorithm questions if the new center is feasible and so on. The key is that the new ellipsoid has substantially smaller volume than the previous one. When the volume of the current ellipsoid shrinks to a sufficiently small value, we are able to conclude that $\mathcal{Q}$ is empty. This fact is used to show the polynomial-time convergence of the algorithm.

The crux of the complexity analysis of the algorithm is on the a priori determination of the iteration bound. This in turn depends on three factors. The volume of the initial ellipsoid E_0, the rate of volume shrinkage ($vol(E_{k+1})/vol(E_k) < e^{-\frac{1}{(2d)}}$), and the volume threshold at which we can safely conclude that $\mathcal{Q}$ must be empty. The assumption of $\mathcal{Q}$ being a rational polyhedron is used to argue that $\mathcal{Q}$ can be

modified into a full-dimensional polytope without affecting the decision question: "Is $\mathcal{Q}$ non-empty?" After careful accounting for all of these technical details and some others (e.g., compensating for the roundoff errors caused by the square root computation in the algorithm), it is possible to establish the following fundamental result.

Theorem 15.1 *There exists a polynomial $g(d, \phi)$ such that the* **ellipsoid method** *runs in time bounded by $T\, g(d, \phi)$ where ϕ is an upper bound on the size of linear inequalities in some description of $\mathcal{Q}$ and T is the maximum time required by the oracle* **Strong Separation**$(\mathcal{Q}, \mathbf{y})$ *on inputs* $\mathbf{y}$ *of size at most $g(d, \phi)$.*

The size of a linear inequality is just the length of the encoding of all of the coefficients needed to describe the inequality. A direct implication of the theorem is that solvability of linear inequalities can be checked in polynomial time if strong separation can be solved in polynomial time. This implies that the standard linear programming solvability question has a polynomial-time algorithm (since separation can be effected by simply checking all of the constraints). Happily, this approach provides polynomial-time algorithms for much more than just the standard case of linear programming solvability. The theorem can be extended to show that the optimization of a linear objective function over $\mathcal{Q}$ also reduces to a polynomial number of calls to the strong separation oracle on $\mathcal{Q}$. A converse to this theorem also holds, namely, separation can be solved by a polynomial number of calls to a solvability/optimization oracle (Grötschel et al. [1982]). Thus, optimization and separation are polynomially equivalent. This provides a very powerful technique for identifying tractable classes of optimization problems. Semidefinite programming and submodular function minimization are two important classes of optimization problems that can be solved in polynomial time using this property of the ellipsoid method.

15.2.1.4 Semidefinite Programming

The following optimization problem defined on symmetric $(n \times n)$ real matrices

$$\text{(SDP)} \quad \min_{X \in \Re^{n \times n}} \left\{ \sum_{ij} C \bullet X : A \bullet X = B,\ X \succeq 0 \right\}$$

is called a semidefinite program. Note that $X \succeq 0$ denotes the requirement that X is a positive semidefinite matrix, and $F \bullet G$ for $n \times n$ matrices F and G denotes the product matrix $(F_{ij} * G_{ij})$. From the definition of positive semidefinite matrices, $X \succeq 0$ is equivalent to

$$\mathbf{q}^T X \mathbf{q} \geq 0 \quad \text{for every } \mathbf{q} \in \Re^n$$

Thus semidefinite programming (SDP) is really a linear program on $O(n^2)$ variables with an (uncountably) infinite number of linear inequality constraints. Fortunately, the strong separation oracle is easily realized for these constraints. For a given symmetric X we use Cholesky factorization to identify the minimum eigenvalue $\lambda_{\min}$. If $\lambda_{\min}$ is nonnegative then $X \succeq 0$ and if, on the other hand, $\lambda_{\min}$ is negative we have a separating inequality

$$\boldsymbol{\gamma}_{\min}^T X \boldsymbol{\gamma}_{\min} \geq 0$$

where $\boldsymbol{\gamma}_{\min}$ is the eigenvector corresponding to $\lambda_{\min}$. Since the Cholesky factorization can be computed by an $O(n^3)$ algorithm, we have a polynomial-time separation oracle and an efficient algorithm for SDP via the ellipsoid method. Alizadeh [1995] has shown that interior point methods can also be adapted to solving SDP to within an additive error ϵ in time polynomial in the size of the input and $\log 1/\epsilon$.

This result has been used to construct efficient approximation algorithms for maximum stable sets and cuts of graphs, Shannon capacity of graphs, and minimum colorings of graphs. It has been used to define hierarchies of relaxations for integer linear programs that strictly improve on known exponential-size linear programming relaxations. We shall encounter the use of SDP in the approximation of a maximum weight cut of a given vertex-weighted graph in Section 15.7.

15.2.1.5 Minimizing Submodular Set Functions

The minimization of submodular set functions is another important class of optimization problems for which ellipsoidal and projective scaling algorithms provide polynomial-time solution methods.

Definition 15.1 Let N be a finite set. A real valued set function f defined on the subsets of N is submodular if $f(X \cup Y) + f(X \cap Y) \leq f(X) + f(Y)$ for $X, Y \subseteq N$.

Example 15.1

Let $G = (V, E)$ be an undirected graph with V as the node set and E as the edge set. Let $c_{ij} \geq 0$ be the weight or capacity associated with edge $(ij) \in E$. For $S \subseteq V$, define the cut function $c(S) = \sum_{i \in S,\, j \in V \setminus S} c_{ij}$. The cut function defined on the subsets of V is submodular since $c(X) + c(Y) - c(X \cup Y) - c(X \cap Y) = \sum_{i \in X \setminus Y,\, j \in Y \setminus X} 2c_{ij} \geq 0$.

The optimization problem of interest is

$$\min\{f(X) : X \subseteq N\}$$

The following remarkable construction that connects submodular function minimization with convex function minimization is due to Lovász (see Grötschel et al. [1988]).

Definition 15.2 The Lovász extension $\hat{f}(.)$ of a submodular function $f(.)$ satisfies

- $\hat{f} : [0,1]^N \rightarrow \Re$.
- $\hat{f}(\mathbf{x}) = \sum_{I \in \mathcal{I}} \lambda_I f(\mathbf{x}_I)$ where $\mathbf{x} = \sum_{I \in \mathcal{I}} \lambda_I \mathbf{x}_I$, $\mathbf{x} \in [0,1]^N$, $\mathbf{x}_I$ is the incidence vector of I for each $I \in \mathcal{I}$, $\lambda_I > 0$ for each I in $\mathcal{I}$, and $\mathcal{I} = \{I_1, I_2, \ldots, I_k\}$ with $\emptyset \neq I_1 \subset I_2 \subset \cdots \subset I_k \subseteq N$. Note that the representation $\mathbf{x} = \sum_{I \in \mathcal{I}} \lambda_I \mathbf{x}_I$ is unique given that the $\lambda_I > 0$ and that the sets in $\mathcal{I}$ are nested.

It is easy to check that $\hat{f}(.)$ is a convex function. Lovász also showed that the minimization of the submodular function $f(.)$ is a special case of convex programming by proving

$$\min\{f(X) : X \subseteq N\} = \min\{\hat{f}(\mathbf{x}) : \mathbf{x} \in [0,1]^N\}$$

Further, if $\mathbf{x}^*$ is an optimal solution to the convex program and

$$\mathbf{x}^* = \sum_{I \in \mathcal{I}} \lambda_I \mathbf{x}_I$$

then for each $\lambda_I > 0$, it can be shown that $I \in \mathcal{I}$ minimizes f. The ellipsoid method can be used to solve this convex program (and hence submodular minimization) using a polynomial number of calls to an oracle for f [this oracle returns the value of $f(X)$ when input X].

15.2.1.6 Interior Point Methods

The announcement of the polynomial solvability of linear programming followed by the probabilistic analyses of the simplex method in the early 1980s left researchers in linear programming with a dilemma. We had one method that was good in a theoretical sense but poor in practice and another that was good in practice (and on average) but poor in a theoretical worst-case sense. This left the door wide open for a method that was good in both senses. Narendra Karmarkar closed this gap with a breathtaking new projective scaling algorithm. In retrospect, the new algorithm has been identified with a class of nonlinear programming methods known as logarithmic barrier methods. Implementations of a primal–dual variant of the logarithmic barrier method have proven to be the best approach at present. It is this variant that we describe.

It is well known that moving through the interior of the feasible region of a linear program using the negative of the gradient of the objective function, as the movement direction, runs into trouble because of getting *jammed* into corners (in high dimensions, corners make up most of the interior of a polyhedron). This jamming can be overcome if the negative gradient is balanced with a *centering* direction. The centering

direction in Karmarkar's algorithm is based on the *analytic center* $\mathbf{y}_c$ of a full-dimensional polyhedron $\mathcal{D} = \{\mathbf{y} : A^T\mathbf{y} \leq c\}$ which is the unique optimal solution to

$$\max \left\{ \sum_{j=1}^{n} \ell n\,(\mathbf{z}_j) \;:\; A^T\mathbf{y} + \mathbf{z} = \mathbf{c} \right\}$$

Recall the primal and dual forms of a linear program may be taken as

$$\begin{aligned} (P) \quad & \min\{\mathbf{c}\mathbf{x} : A\mathbf{x} = \mathbf{b},\ \mathbf{x} \geq \mathbf{0}\} \\ (D) \quad & \max\{\mathbf{b}^T\mathbf{y} : A^T\mathbf{y} \leq \mathbf{c}\} \end{aligned}$$

The logarithmic barrier formulation of the dual (D) is

$$(D_\mu) \quad \max \left\{ \mathbf{b}^T\mathbf{y} + \mu \sum_{j=1}^{n} \ell n\,(\mathbf{z}_j) :\; A^T\mathbf{y} \;+\; \mathbf{z} \;=\; \mathbf{c} \right\}$$

Notice that (D_μ) is equivalent to (D) as $\mu \to 0^+$. The optimality (Karush–Kuhn–Tucker) conditions for (D_μ) are given by

$$\begin{aligned} D_{\mathbf{x}} D_{\mathbf{z}} \mathbf{e} &= \mu \mathbf{e} \\ A\mathbf{x} &= \mathbf{b} \\ A^T\mathbf{y} + \mathbf{z} &= \mathbf{c} \end{aligned} \tag{15.1}$$

where $D_{\mathbf{x}}$ and $D_{\mathbf{z}}$ denote $n \times n$ diagonal matrices whose diagonals are $\mathbf{x}$ and $\mathbf{z}$, respectively. Notice that if we set μ to 0, the above conditions are precisely the primal–dual optimality conditions: complementary slackness, primal and dual feasibility of a pair of optimal (P) and (D) solutions. The problem has been reduced to solving the equations in $\mathbf{x}, \mathbf{y}, \mathbf{z}$. The classical technique for solving equations is Newton's method, which prescribes the directions,

$$\begin{aligned} \Delta\mathbf{y} &= -\left(AD_{\mathbf{x}}D_{\mathbf{z}}^{-1}A^T\right)^{-1} AD_{\mathbf{z}}^{-1}(\mu\mathbf{e} - D_{\mathbf{x}}D_{\mathbf{z}}\mathbf{e})\Delta\mathbf{z} = -A^T\Delta\mathbf{y}\Delta\mathbf{x} \\ &= D_{\mathbf{z}}^{-1}(\mu\mathbf{e} - D_{\mathbf{x}}D_{\mathbf{z}}\mathbf{e}) - D_{\mathbf{x}}D_{\mathbf{z}}^{-1}\Delta\mathbf{z} \end{aligned} \tag{15.2}$$

The strategy is to take one Newton step, reduce μ, and iterate until the optimization is complete. The criterion for stopping can be determined by checking for feasibility $(\mathbf{x}, \mathbf{z} \geq \mathbf{0})$ and if the duality gap $(\mathbf{x}^t\mathbf{z})$ is close enough to 0. We are now ready to describe the algorithm.

Procedure 15.2 Primal-Dual Interior:

0. **Initialize:**

   $\mathbf{x}_0 > 0, \mathbf{y}_0 \in \Re^m, \mathbf{z}_0 > 0, \mu_0 > 0, \epsilon > 0, \rho > 0$
   $k := 0$

1. **Iterative step:**

   do
   Stop if $A\mathbf{x}_k = \mathbf{b}$, $A^T\mathbf{y}_k + \mathbf{z}_k = \mathbf{c}$ and $\mathbf{x}_k^T\mathbf{z}_k \leq \epsilon$.
   $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k^P \Delta\mathbf{x}_k$
   $\mathbf{y}_{k+1} \leftarrow \mathbf{y}_k + \alpha_k^D \Delta\mathbf{y}_k$
   $\mathbf{z}_{k+1} \leftarrow \mathbf{z}_k + \alpha_k^D \Delta\mathbf{z}_k$
   /* $\Delta\mathbf{x}_k$, $\Delta\mathbf{y}_k$, $\Delta\mathbf{z}_k$ **are the Newton directions from (1)** */
   $\mu_{k+1} \leftarrow \rho\mu_k$
   $k := k + 1$
   od

2. **End**