automorphisms (the automorphism *group*) provides a great deal of information about symmetries in the graph. For example, the complete graph K_n has $n!$ automorphisms (any mapping will do), while an arbitrary random graph is likely to have few or perhaps only one, since G is always identical to itself.

Several variations of graph isomorphism arise in practice:

- *Is graph G contained in graph H?* – Instead of testing equality, we are often interested in knowing whether a small pattern graph G is a *subgraph* of H. Such problems as clique, independent set, and Hamiltonian cycle are important special cases of subgraph isomorphism.

  There are two distinct graph-theoretic notions of "contained in." *Subgraph isomorphism* asks whether there is a subset of edges and vertices of H that is isomorphic to a smaller graph G. *Induced subgraph isomorphism* asks whether there is a subset of vertices of H whose deletion leaves a subgraph isomorphic to a smaller graph G. For induced subgraph isomorphism, (1) all edges of G must be present in H, and (2) no *non-edges* of G can be present in H. Clique happens to be an instance of both subgraph isomorphism problems, while Hamiltonian cycle is only an example of vanilla subgraph isomorphism.

  Be aware of this distinction in your application. Subgraph isomorphism problems tend to be harder than graph isomorphism, while induced subgraph problems tend to be even harder than subgraph isomorphism. Some flavor of backtracking is your only viable approach.

- *Are your graphs labeled or unlabeled?* – In many applications, vertices or edges of the graphs are *labeled* with some attribute that must be respected in determining isomorphisms. For example, in comparing two bipartite graphs, each with "worker" vertices and "job" vertices, any isomorphism that equated a job with a worker would make no sense.

  Labels and related constraints can be factored into any backtracking algorithm. Further, such constraints significantly speed up the search, by creating many more opportunities for pruning whenever two vertex labels do not match up.

- *Are you testing whether two trees are isomorphic?* – Faster algorithms exist for certain special cases of graph isomorphism, such as trees and planar graphs. Perhaps the most important case is detecting isomorphisms among trees, a problem that arises in language pattern matching and parsing applications. A parse tree is often used to describe the structure of a text; two parse trees will be isomorphic if the underlying pair of texts have the same structure.

  Efficient algorithms for tree isomorphism begin with the leaves of both trees and work inward toward the center. Each vertex in one tree is assigned a label representing the set of vertices in the second tree that might possibly

be isomorphic to it, based on the constraints of labels and vertex degrees. For example, all the leaves in tree T_1 are initially potentially equivalent to all leaves of T_2. Now, working inward, we can partition the vertices adjacent to leaves in T_1 into classes based on how many leaves and non-leaves they are adjacent to. By carefully keeping track of the labels of the subtrees, we can make sure that we have the same distribution of labeled subtrees for T_1 and T_2. Any mismatch means $T_1 \neq T_2$, while completing the process partitions the vertices into equivalence classes defining all isomorphisms. See the references below for more details.

- *How many graphs do you have?* – Many data mining applications involve searching for all instances of a particular pattern graph in a big database of graphs. The chemical structure mapping application described above falls into this family. Such databases typically contain a large number of relatively small graphs. This puts an onus on indexing the graph database by small substructures (say five to ten vertex each), and doing expensive isomorphism tests only against those containing the same substructures as the query graph.

No polynomial-time algorithm is known for graph isomorphism, but neither is it known to be NP-complete. Along with integer factorization (see Section 13.8 (page 420)), it is one of the few important algorithmic problems whose rough computational complexity is still not known. The conventional wisdom is that isomorphism is a problem that lies between P and NP-complete if P $\neq$ NP.

Although no worst-case polynomial-time algorithm is known, testing isomorphism is *usually* not very hard in practice. The basic algorithm backtracks through all $n!$ possible relabelings of the vertices of graph h with the names of vertices of graph g, and then tests whether the graphs are identical. Of course, we can prune the search of all permutations with a given prefix as soon as we detect any mismatch between edges whose vertices are both in the prefix.

However, the real key to efficient isomorphism testing is to preprocess the vertices into "equivalence classes," partitioning them into sets of vertices so that two vertices in different sets cannot possibly be mistaken for each other. All vertices in each equivalence class must share the same value of some invariant that is independent of labeling. Possibilities include:

- *Vertex degree* – This simplest way to partition vertices is based on their degree—the number of edges incident on the vertex. Two vertices of different degrees cannot be identical. This simple partition can be a big win, but won't do much for regular (equal degree) graphs.

- *Shortest path matrix* – For each vertex v, the all-pairs shortest path matrix (see Section 15.4 (page 489)) defines a multiset of $n-1$ distances representing the distances between v and each of the other vertices. Any two identical vertices must define the exact same multiset of distances, so we can partition the vertices into equivalence classes defining identical distance multisets.

- *Counting length-k paths* – Taking the adjacency matrix of G and raising it to the kth power gives a matrix where $G^k[i,j]$ counts the number of (nonsimple) paths from i to j. For each vertex and each k, this matrix defines a multiset of path-counts, which can be used for partitioning as with distances above. You could try all $1 \leq k \leq n$ or beyond, and use any single deviation as an excuse to partition.

Using these invariants, you should be able to partition the vertices of most graphs into a large number of small equivalence classes. Finishing the job off with backtracking should then be short work. We assign each vertex the name of its equivalence class as a label, and treat it as a labeled matching problem. It is harder to detect isomorphisms between highly-symmetric graphs than it is with random graphs because of the reduced effectiveness of these equivalence-class partitioning heuristics.

Implementations: The best known isomorphism testing program is `nauty` (No AUTomorphisms, Yes?)—a set of very efficient C language procedures for determining the automorphism group of a vertex-colored graph. Nauty is also able to produce a canonically-labeled isomorph of the graph, to assist in isomorphism testing. It was the basis of the first program to generate all 11-vertex graphs without isomorphs, and can test most graphs with fewer than 100 vertices in well under a second. Nauty has been ported to a variety of operating systems and C compilers. It is available at *http://cs.anu.edu.au/∼bdm/nauty/*. The theory behind `nauty` is described in [McK81].

The `VFLib` graph-matching library contains implementations for several different algorithms for *both* graph and subgraph isomorphism testing. This library has been widely used and very carefully benchmarked [FSV01]. It is available at *http://amalfi.dis.unina.it/graph/*.

`GraphGrep` [GS02] (*http://www.cs.nyu.edu/shasha/papers/graphgrep/*) is a representative data mining tool for querying large databases of graphs.

Valiente [Val02] has made available the implementations of graph/subgraph isomorphism algorithms for both trees and graphs in his book [Val02]. These C++ implementations run on top of LEDA (see Section 19.1.1 (page 658)), and are available at *http://www.lsi.upc.edu/∼valiente/algorithm/*.

Kreher and Stinson [KS99] compute isomorphisms of graphs in addition to more general group-theoretic operations. These implementations in C are available at *http://www.math.mtu.edu/∼kreher/cages/Src.html*.

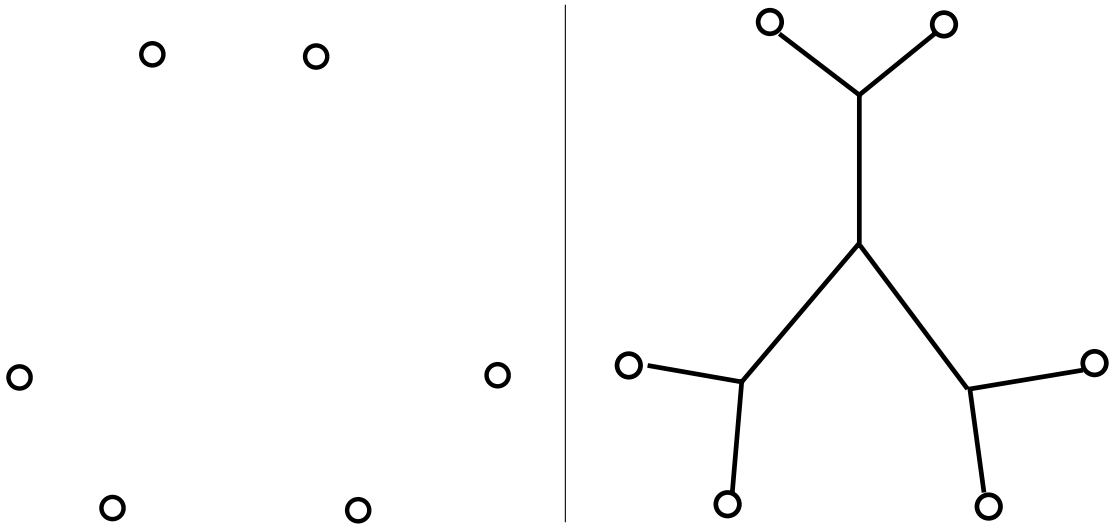
Notes: Graph isomorphism is an important problem in complexity theory. Monographs on isomorphism detection include [Hof82, KST93]. Valiente [Val02] focuses on algorithms for tree and subgraph isomorphism. Kreher and Stinson [KS99] take a more group-theoretic approach to isomorphism testing. Graph mining systems and algorithms are surveyed in [CH06]. See [FSV01] for performance comparisons between different graph and subgraph isomorphism algorithms.

Polynomial-time algorithms are known for planar graph isomorphism [HW74] and for graphs where the maximum vertex degree is bounded by a constant [Luk80]. The all-pairs

shortest path heuristic is due to [SD76], although there exist nonisomorphic graphs that realize the exact same set of distances [BH90]. A linear-time tree isomorphism algorithm for both labeled and unlabeled trees is presented in [AHU74].

A problem is said to be *isomorphism-complete* if it is provably as hard as isomorphism. Testing the isomorphism of bipartite graphs is isomorphism-complete, since any graph can be made bipartite by replacing each edge by two edges connected with a new vertex. Clearly, the original graphs are isomorphic if and only if the transformed graphs are.

Related Problems: Shortest path (see page 489), string matching (see page 628).



INPUT OUTPUT

16.10 Steiner Tree

Input description: A graph $G = (V, E)$. A subset of vertices $T \in V$.

Problem description: Find the smallest tree connecting all the vertices of T.

Discussion: Steiner trees arise often in network design problems, since the minimum Steiner tree describes how to connect a given set of sites using the smallest amount of wire. Analogous problems occur when designing networks of water pipes or heating ducts and in VLSI circuit design. Typical Steiner tree problems in VLSI are to connect a set of sites to (say) ground under constraints such as material cost, signal propagation time, or reducing capacitance.

The Steiner tree problem is distinguished from the minimum spanning tree (MST) problem (see Section 15.3 (page 484)) in that we are permitted to construct or select intermediate connection points to reduce the cost of the tree. Issues in Steiner tree construction include:

- *How many points do you have to connect?* – The Steiner tree of a pair of vertices is simply the shortest path between them (see Section 15.4 (page 489)). The Steiner tree of all the vertices, when $S = V$, simply defines the MST of G. The general minimum Steiner tree problem is NP-hard despite these special cases, and remains so under a broad range of restrictions.

- *Is the input a set of geometric points or a distance graph?* – Geometric versions of Steiner tree take a set of points as input, typically in the plane, and seek the smallest tree connecting the points. A complication is that the set of possible intermediate points is not given as part of the input but must be deduced from the set of points. These possible Steiner points must satisfy several geometric properties, which can be used to reduce the set of candidates down to a finite number. For example, every Steiner point will have a degree of exactly three in a minimum Steiner tree, and the angles formed between any two of these edges must be exactly 120 degrees.

- *Are there constraints on the edges we can use?* – Many wiring problems correspond to geometric versions of the problem, where all edges are restricted to being either horizontal or vertical. This is the so-called *rectilinear Steiner problem*. A different set of angular and degree conditions apply for rectilinear Steiner trees than for Euclidean trees. In particular, all angles must be multiples of 90 degrees, and each vertex is of a degree up to four.

- *Do I really need an optimal tree?* – Certain Steiner tree applications (e.g., circuit design and communications networks) justify investing large amounts of computation to find the best possible Steiner tree. This implies an exhaustive search technique such as backtracking or branch-and-bound. There are many opportunities for pruning search based on geometric and graph-theoretic constraints.

  Still, Steiner tree remains a hard problem. We recommend experimenting with the implementations described below before attempting your own.

- *How can I reconstruct Steiner vertices I never knew about?* – A very special type of Steiner tree arises in classification and evolution. A *phylogenic tree* illustrates the relative similarity between different objects or species. Each object represents (typically) a leaf/terminal vertex of the tree, with intermediate vertices representing branching points between classes of objects. For example, an evolutionary tree of animal species might have leaf nodes of *human, dog, snake* and internal nodes corresponding to taxa *(animal, mammal, reptile)*. A tree rooted at *animal* with *dog* and *human* classified under *mammal* implies that humans are closer to dogs than to snakes.

  Many different phylogenic tree construction algorithms have been developed that vary in (1) the data they attempt to model, and (2) the desired optimization criterion. Each combination of reconstruction algorithm and distance measure is likely to give a different answer, so identifying the "right" method for any given application is somewhat a question of faith. A reasonable procedure is to acquire a standard package of implementations, discussed below, and then see what happens to your data under all of them.

Fortunately, there is a good, efficient heuristic for finding Steiner trees that works well on all versions of the problem. Construct a graph modeling your input,

setting the weight of edge (i, j) equal to the distance from point i to point j. Find an MST of this graph. You are guaranteed a provably good approximation for both Euclidean and rectilinear Steiner trees.

The worst case for a MST approximation of the Euclidean Steiner tree is three points forming an equilateral triangle. The MST will contain two of the sides (for a length of 2), whereas the minimum Steiner tree will connect the three points using an interior point, for a total length of $\sqrt{3}$. This ratio of $\sqrt{3}/2 \approx 0.866$ is always achieved, and in practice the easily-computed MST is usually within a few percent of the optimal Steiner tree. The rectilinear Steiner tree / MST ratio is always $\geq 2/3 \approx 0.667$.

Such an MST can be refined by inserting a Steiner point whenever the edges of the minimum spanning tree incident on a vertex form an angle of less than 120 degrees between them. Inserting these points and locally readjusting the tree edges can move the solution a few more percent towards the optimum. Similar optimizations are possible for rectilinear spanning trees.

Note that we are only interested in the subtree connecting the terminal vertices. We may need to trim the MST if we add nonterminal vertices to the input of the problem. We retain only the tree edges which lie on the (unique) path between some pair of terminal nodes. The complete set of these can be found in $O(n)$ time by performing a BFS on the full tree starting from any single terminal node.

An alternative heuristic for graphs is based on shortest path. Start with a tree consisting of the shortest path between two terminals. For each remaining terminal t, find the shortest path to a vertex v in the tree and add this path to the tree. The time complexity and quality of this heuristic depend upon the insertion order of the terminals and how the shortest-path computations are performed, but something simple and fairly effective is likely to result.

Implementations: GeoSteiner is a package for solving both Euclidean and rectilinear Steiner tree problems in the plane by Warme, Winter, and Zachariasen [WWZ00]. It also solves the related problem of MSTs in hypergraphs, and claims to have solved problems as large as 10,000 points to optimality. It is available from *http://www.diku.dk/geosteiner/*. This is almost certainly the best code for geometric instances of Steiner tree.

FLUTE (*http://home.eng.iastate.edu/~cnchu/flute.html*) computes rectilinear Steiner trees, emphasizing speed. It contains a user-defined parameter to control the tradeoff between solution quality and run time.

GOBLIN (*http://www.math.uni-augsburg.de/~fremuth/goblin.html*) includes both heuristics and search methods for finding Steiner trees in graphs.

The programs PHYLIP (*http://evolution.genetics.washington.edu/phylip.html*) and PAUP (*http://paup.csit.fsu.edu/*) are widely-used packages for inferring phylogenic trees. Both contain over 20 different algorithms for constructing phylogenic trees from data. Although many of them are designed to work with molecular sequence data, several general methods accept arbitrary distance matrices as input.

Notes: Recent monographs on the Steiner tree problem include Hwang, Richards, and Winter [HRW92] and Prömel and Steger [PS02]. Du, et al. [DSR00] is a collection of recent surveys on all aspects of Steiner trees. Older surveys on the problem include [Kuh75]. Empirical results on Steiner tree heuristics include [SFG82, Vos92].

The Euclidean Steiner problem dates back to Fermat, who asked how to find a point p in the plane minimizing the sum of the distances to three given points. This was solved by Torricelli before 1640. Steiner was apparently one of several mathematicians who worked the general problem for n points, and was mistakenly credited with the problem. An interesting, more detailed history appears in [HRW92].

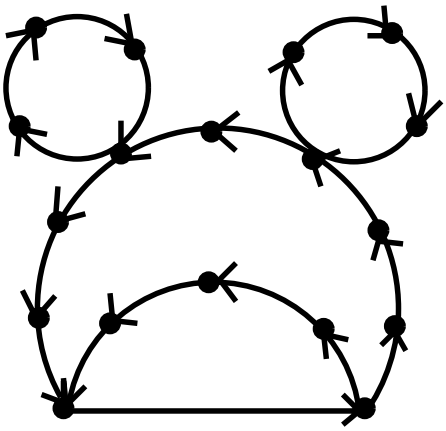
Gilbert and Pollak [GP68] first conjectured that the ratio of the length of the minimum Steiner tree over the MST is always $\geq \sqrt{3}/2 \approx 0.866$. After twenty years of active research, the Gilbert-Pollak ratio was finally proven by Du and Hwang [DH92]. The Euclidean MST for n points in the plane can be constructed in $O(n \lg n)$ time [PS85].

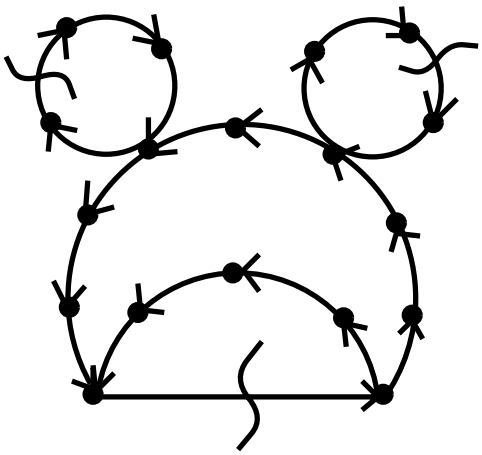
Arora [Aro98] gave a polynomial-time approximation scheme (PTAS) for Steiner trees in k-dimensional Euclidian space. A 1.55-factor approximation for Steiner trees on graphs is due to Robins and Zelikovsky [RZ05].

Expositions on the proof that the Steiner tree problem for graphs is hard [Kar72] include [Eve79a]. Expositions on exact algorithms for Steiner trees in graphs include [Law76]. The hardness of Steiner tree for Euclidean and rectilinear metrics was established in [GGJ77, GJ77]. Euclidean Steiner tree is not known to be in NP, because of numerical issues in representing distances.

Analogies can be drawn between minimum Steiner trees and minimum energy configurations in certain physical systems. The case that such analog systems—including the behavior of soap films over wire frames—"solve" the Steiner tree problem is discussed in [Mie58].

Related Problems: Minimum spanning tree (see page 484), shortest path (see page 489).





INPUT OUTPUT

16.11 Feedback Edge/Vertex Set

Input description: A (directed) graph $G = (V, E)$.

Problem description: What is the smallest set of edges E' or vertices V' whose deletion leaves an acyclic graph?

Discussion: Feedback set problems arise because many things are easier to do on directed acyclic graphs (DAGs) than general digraphs. Consider the problem of scheduling jobs with precedence constraints (i.e. , job A must come before job B). When the constraints are all consistent, the resulting graph is a DAG, and topological sort (see Section 15.2 (page 481)) can be used to order the vertices to respect them. But how can you design a schedule when there are cyclic constraints, such as A must be done before B, which must be done before C, which must be done before A?

By identifying a feedback set, we identify the smallest number of constraints that must be dropped to permit a valid schedule. In the *feedback edge* (or arc) set problem, we drop individual precedence constraints. In the *feedback vertex set* problem, we drop entire jobs and all constraints associated with them.

Similar considerations are involved in eliminating race conditions from electronic circuits. This explains why the problem is called "feedback" set. It is also known as the *maximum acyclic subgraph problem.*

One final application has to do with ranking tournaments. Suppose we want to rank order the skills of players at some two-player game, such as chess or tennis. We can construct a directed graph where there is an arc from x to y if x beats y in a game. The higher-ranked player *should* be at the lower-ranked player, although upsets often occur. A natural ranking is the topological sort resulting after deleting the minimum set of feedback edges (upsets) from the graph.

Issues in feedback set problems include:

- *Do* any *constraints have to be dropped?* – No changes are needed if the graph is already a DAG, which can be determined via topological sort. One way to find a feedback set modifies the topological sorting algorithm to delete whatever edge or vertex is causing the trouble whenever a contradiction is found. This feedback set might be much larger than needed, however, since feedback edge set and feedback vertex set are NP-complete on directed graphs.

- *How can I find a good feedback edge set?* – An effective linear-time heuristic constructs a vertex ordering and then deletes any arc going in the wrong direction. At least half the arcs must go either left-to-right or right-to-left for any vertex order, so take the smaller partition as your feedback set.

  But what is the right vertex order to start from? One natural order is to sort the vertices in terms of edge-imbalance, namely in-degree minus out-degree. Another approach starts by picking an arbitrary vertex v. Any vertex x defined by an in-going edge (x, v) will be placed to the left of v. Any x defined by out-going edge (v, x) will analogously be placed to the right of v. We can now recur on the left and right subsets to complete the vertex order.

- *How can I find a good feedback vertex set?* – The heuristics above yield vertex orders defining (hopefully) few back edges. We seek a small set of vertices that together cover these backedges. This is exactly a vertex cover problem, the heuristics for which are discussed in Section 16.3 (page 530).

- *What if I want to break all cycles in an undirected graph?* – The problem of finding feedback sets in undirected graphs is quite different from digraphs. Trees are undirected graphs without cycles, and every tree on n vertices contains exactly $n - 1$ edges. Thus the smallest feedback edge set of any undirected graph G is $|E| - (n - c)$, where c is the number of connected components of G. The back edges encountered during a depth-first search of G qualified as a minimum feedback edge set.

  The feedback vertex set problem remains NP-complete for undirected graphs, however. A reasonable heuristic uses breadth-first search to identify the shortest cycle in G. The vertices in this cycle are all deleted from G, and the shortest remaining cycle identified. This find-and-delete procedure is employed until the graph is acyclic. The optimal feedback vertex set must contain at least one vertex from each of these vertex-disjoint cycles, so the average deleted-cycle length determines just how good our approximation is.

It may pay to refine any of these heuristic solutions using randomization or simulated annealing. To move between states, we can modify the vertex permutation by swapping pairs in order or insert/delete vertices to/from the candidate feedback set.

Implementations: Greedy randomized adaptive search (GRASP) heuristics for both feedback vertex and feedback edge set problems have been implemented by Festa, et al. [FPR01] as Algorithm 815 of the *Collected Algorithms of the ACM* (see Section 19.1.6 (page 659)). These Fortran codes are also available from *http://www.research.att.com/∼mgcr/src/*.

GOBLIN (*http://www.math.uni-augsburg.de/∼fremuth/goblin.html*) includes an approximation heuristic for minimum feedback arc set.

The econ_order program of the Stanford GraphBase (see Section 19.1.8 (page 660)) permutes the rows and columns of a matrix so as to minimize the sum of the numbers below the main diagonal. Using an adjacency matrix as the input and deleting all edges below the main diagonal leaves an acyclic graph.

Notes: See [FPR99] for a survey on the feedback set problem. Expositions of the proofs that feedback minimization is hard [Kar72] include [AHU74, Eve79a]. Both feedback vertex and edge set remain hard even if no vertex has in-degree or out-degree greater than two [GJ79].

Bafna, et al. [BBF99] gives a 2-factor approximation for feedback vertex set in undirected graphs. Feedback edge sets in directed graphs can be approximated to within a factor of $O(\log n \log \log n)$ [ENSS98]. Heuristics for ranking tournaments are discussed in [CFR06]. Experiments with heuristics are reported in [Koe05].

An interesting application of feedback arc set to economics is presented in [Knu94]. For each pair A, B of sectors of the economy, we are given how much money flows from A to B. We seek to order the sectors to determine which sectors are primarily producers to other sectors, and which deliver primarily to consumers.

Related Problems: Bandwidth reduction (see page 398), topological sorting (see page 481), scheduling (see page 468).

17

Computational Geometry

Computational geometry is the algorithmic study of geometric problems. Its emergence coincided with application areas such as computer graphics, computer-aided design/manufacturing, and scientific computing, which together provide much of the motivation for geometric computing. The past twenty years have seen enormous maturity in computational geometry, resulting in a significant body of useful algorithms, software, textbooks, and research results.

Good books on computational geometry include:

- *de Berg, et al.* [dBvKOS00] – The "three Mark's" book is the best general introduction to the theory of computational geometry and its fundamental algorithms.

- *O'Rourke* [O'R01] – This is the best practical introduction to computational geometry. The emphasis is on careful and correct implementation of geometric algorithms. C and Java code are available from *http://maven.smith.edu/∼orourke/books/compgeom.html*.

- *Preparata and Shamos* [PS85] – Although somewhat out of date, this book remains a good general introduction to computational geometry, stressing algorithms for convex hulls, Voronoi diagrams, and intersection detection.

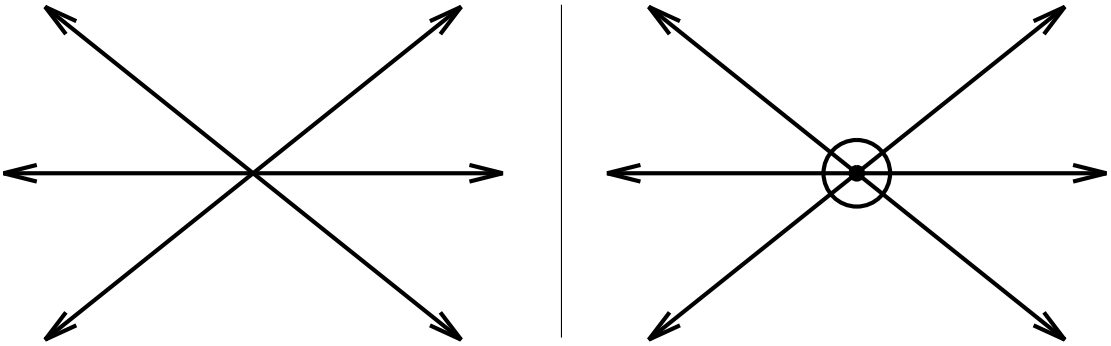
- *Goodman and O'Rourke* [GO04] – This recent collection of survey articles provides a detailed overview of what is known in most every subfield of discrete and computational geometry.

The leading conference in computational geometry is the *ACM Symposium on Computational Geometry*, held annually in late May or early June. Although the primary results presented at the conference are theoretical, there has been a

S.S. Skiena, *The Algorithm Design Manual*, 2nd ed., DOI: 10.1007/978-1-84800-070-4_17,

concerted effort on the part of the research community to increase the presence of applied, experimental work through video reviews and poster sessions.

There is a growing body of implementations of geometric algorithms. We point out specific implementations where applicable in the catalog, but the reader should be particularly aware of `CGAL` (Computational Geometry Algorithms Library)—a comprehensive library of geometric algorithms in C++ produced as a result of a joint European project. Anyone with a serious interest in geometric computing should check it out at *http://www.cgal.org/*.



INPUT OUTPUT

17.1 Robust Geometric Primitives

Input description: A point p and line segment l, or two line segments l_1, l_2.

Problem description: Does p lie over, under, or on l? Does l_1 intersect l_2?

Discussion: Implementing basic geometric primitives is a task fraught with peril, even for such simple tasks as returning the intersection point of two lines. It is more complicated than you may think. What should you return if the two lines are parallel, meaning they don't intersect at all? What if the lines are identical, so the intersection is not a point but the entire line? What if one of the lines is horizontal, so that in the course of solving the equations for the intersection point you are likely to divide by zero? What if the two lines are almost parallel, so that the intersection point is so far from the origin as to cause arithmetic overflows? These issues become even more complicated for intersecting line segments, since there are many other special cases that must be watched for and treated specially.

If you are new to implementing geometric algorithms, I suggest that you study O'Rourke's *Computational Geometry in C* [O'R01] for practical advice and complete implementations of basic geometric algorithms and data structures. You are likely to avoid many headaches by following in his footsteps.

There are two different issues at work here: geometric degeneracy and numerical stability. *Degeneracy* refers to annoying special cases that must be treated in substantially different ways, such as when two lines intersect in more or less than a single point. There are three primary approaches to dealing with degeneracy:

- *Ignore it* – Make as an operating assumption that your program will work correctly only if no three points are collinear, no three lines meet at a point, no intersections happen at the endpoints of line segments, etc. This is probably the most common approach, and what I might recommend for short-term

projects if you can live with frequent crashes. The drawback is that interesting data often comes from points sampled on a grid, and so is inherently very degenerate.

- *Fake it* – Randomly or symbolically perturb your data so that it becomes nondegenerate. By moving each of your points a small amount in a random direction, you can break many of the existing degeneracies in the data, hopefully without creating too many new problems. This probably should be the first thing to try once you decide that your program is crashing too often. A problem with random perturbations is that they can change the shape of your data in subtle ways, which may be intolerable for your application. There also exist techniques to "symbolically" perturb your data to remove degeneracies in a consistent manner, but these require serious study to apply correctly.
- *Deal with it* – Geometric applications can be made more robust by writing special code to handle each of the special cases that arise. This can work well if done with care at the beginning, but not so well if kludges are added whenever the system crashes. Expect to expend significant effort if you are determined to do it right.

Geometric computations often involve floating-point arithmetic, which leads to problems with overflows and numerical precision. There are three basic approaches to the issue of numerical stability:

- *Integer arithmetic* – By forcing all points of interest to lie on a fixed-size integer grid, you can perform exact comparisons to test whether any two points are equal or two line segments intersect. The cost is that the intersection point of two lines may not be exactly representable as a grid point. This is likely to be the simplest and best method, if you can get away with it.
- *Double precision reals* – By using double-precision floating point numbers, you may get lucky and avoid numerical errors. Your best bet might be to keep all the data as single-precision reals, and use double-precision for intermediate computations.
- *Arbitrary precision arithmetic* – This is certain to be correct, but also to be slow. This approach seems to be gaining favor in the research community. Careful analysis can minimize the need for high-precision arithmetic and thus the performance penalty. Still, you should expect high-precision arithmetic to be several orders of magnitude slower than standard floating-point arithmetic.

The difficulties associated with producing robust geometric software are still under attack by researchers. The best practical technique is to base your applications on a small set of geometric primitives that handle as much of the low-level geometry as possible. These primitives include:

- *Area of a triangle* – Although it is well known that the area $A(t)$ of a triangle $t = (a, b, c)$ is half the base times the height, computing the length of the base and altitude is messy work with trigonometric functions. It is better to use the determinant formula for *twice* the area:

$$2 \cdot A(t) = \begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{vmatrix} = a_x b_y - a_y b_x + a_y c_x - a_x c_y + b_x c_y - c_x b_y$$

  This formula generalizes to compute $d!$ times the volume of a simplex in d dimensions. Thus, $3! = 6$ times the volume of a tetrahedron $t = (a, b, c, d)$ in three dimensions is

$$6 \cdot A(t) = \begin{vmatrix} a_x & a_y & a_z & 1 \\ b_x & b_y & b_z & 1 \\ c_x & c_y & c_z & 1 \\ d_x & d_y & d_z & 1 \end{vmatrix}$$

  Note that these are signed volumes and can be negative, so take the absolute value first. Section 13.4 (page 404) explains how to compute determinants.

  The conceptually simplest way to compute the area of a polygon (or polyhedron) is to triangulate it and then sum up the area of each triangle. Implementations of a slicker algorithm that avoids triangulation are discussed in [O'R01, SR03].

- *Above-below-on test* – Does a given point c lie above, below, or on a given line l? A clean way to deal with this is to represent l as a directed line that passes through point a before point b, and ask whether c lies to the left or right of the directed line l. It is up to you to decide whether left means above or below.

  This primitive can be implemented using the sign of the triangle area as computed above. If the area of $t(a, b, c) > 0$, then c lies to the left of $\overline{ab}$. If the area of $t(a, b, c) = 0$, then c lies on $\overline{ab}$. Finally, if the area of $t(a, b, c) < 0$, then c lies to the right of $\overline{ab}$. This generalizes naturally to three dimensions, where the sign of the area denotes whether d lies above or below the oriented plane (a, b, c).

- *Line segment intersection* – The above-below primitive can be used to test whether a line intersects a line segment. It does iff one endpoint of the segment is to the left of the line and the other is to the right. Segment intersection is similar but more complicated. We refer you to implementations described below. The decision as to whether two segments intersect if they share an endpoint depends upon your application and is representative of the problems with degeneracy.

- *In-circle test* – Does point d lie inside or outside the circle defined by points a, b, and c in the plane? This primitive occurs in all Delaunay triangulation algorithms, and can be used as a robust way to do distance comparisons. Assuming that a, b, c are labeled in counterclockwise order around the circle, compute the determinant

$$\text{incircle}(a,b,c,d) = \begin{vmatrix} a_x & a_y & a_x^2+a_y^2 & 1 \\ b_x & b_y & b_x^2+b_y^2 & 1 \\ c_x & c_y & c_x^2+c_y^2 & 1 \\ d_x & d_y & d_x^2+d_y^2 & 1 \end{vmatrix}$$

  In-circle will return 0 if all four points are cocircular—a positive value if d is inside the circle, and negative if d is outside.

Check out the Implementations section before you build your own.

Implementations: CGAL (*www.cgal.org*) and LEDA (see Section 19.1.1 (page 658)) both provide very complete sets of geometric primitives for planar geometry written in C++. LEDA is easier to learn and to work with, but CGAL is more comprehensive and freely available. If you are starting a significant geometric application, you should at least check them out before you try to write your own.

O'Rourke [O'R01] provides implementations in C of most of the primitives discussed in this section. See Section 19.1.10 (page 662). These primitives were implemented primarily for exposition rather than production use, but they should be reliable and appropriate for small applications.

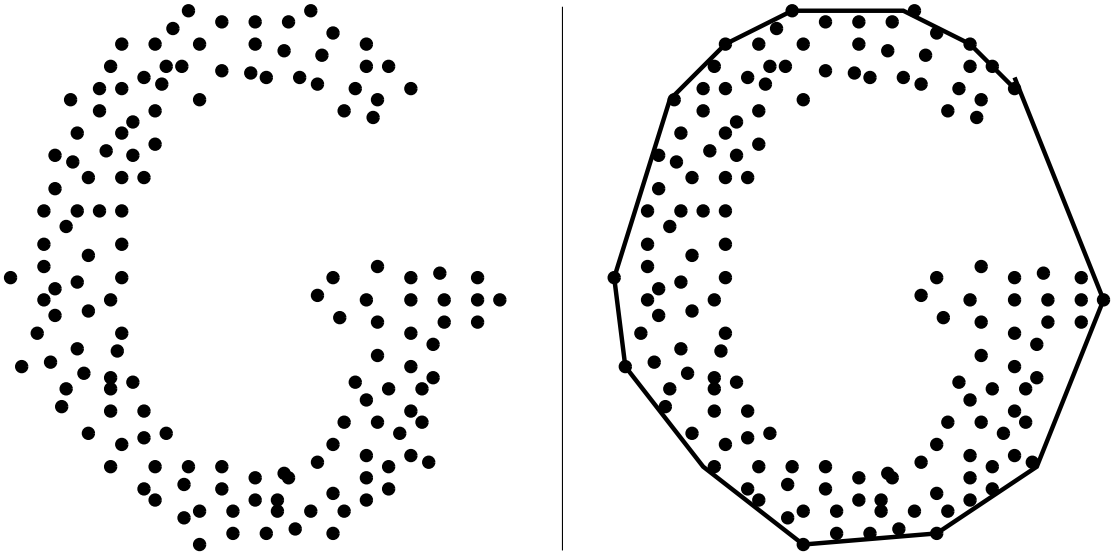
The *Core Library* (see *http://cs.nyu.edu/exact/*) provides an API, which supports the Exact Geometric Computation (EGC) approach to numerically robust algorithms. With small changes, any C/C++ program can use it to readily support three levels of accuracy: machine-precision, arbitrary-precision, and guaranteed.

Shewchuk's [She97] robust implementation of basic geometric primitives in C++ is available at *http://www.cs.cmu.edu/∼quake/robust.html*.

Notes: O'Rourke [O'R01] provides an implementation-oriented introduction to computational geometry which stresses robust geometric primitives. It is recommended reading. LEDA [MN99] provides another excellent role model.

Yap [Yap04] gives a excellent survey on techniques for achieving robust geometric computation, with a book [MY07] in preparation. Kettner, et al. [KMP+04] provides graphic evidence of the troubles that arise when employing real arithmetic in geometric algorithms such as convex hull. Controlled perturbation [MOS06] is a new approach for robust computation that is receiving considerable attention. Shewchuk [She97] and Fortune and van Wyk [FvW93] present careful studies on the costs of using arbitrary-precision arithmetic for geometric computation. By being careful about when to use it, reasonable efficiency can be maintained while achieving complete robustness.

Related Problems: Intersection detection (see page 591), maintaining arrangements (see page 614).



INPUT OUTPUT

17.2 Convex Hull

Input description: A set S of n points in d-dimensional space.

Problem description: Find the smallest convex polygon (or polyhedron) containing all the points of S.

Discussion: Finding the convex hull of a set of points is *the* most important elementary problem in computational geometry, just as sorting is the most important elementary problem in combinatorial algorithms. It arises because constructing the hull captures a rough idea of the shape or extent of a data set.

Convex hull serves as a first preprocessing step to many, if not most, geometric algorithms. For example, consider the problem of finding the diameter of a set of points, which is the pair of points that lie a maximum distance apart. The diameter must be between two points on the convex hull. The $O(n \lg n)$ algorithm for computing diameter proceeds by first constructing the convex hull, then for each hull vertex finding which other hull vertex lies farthest from it. The so-called "rotating-calipers" method can be used to move efficiently from one-diametrically opposed hull vertex to another by always proceeding in a clockwise fashion around the hull.

There are almost as many convex hull algorithms as sorting algorithms. Answer the following questions to help choose between them:

- *How many dimensions are you working with?* – Convex hulls in two and even three dimensions are fairly easy to work with. However, certain valid assumptions in lower dimensions break down as the dimensionality increases. For example, any n-vertex polygon in two dimensions has exactly n edges. However, the relationship between the numbers of faces and vertices becomes more complicated even in three dimensions. A cube has eight vertices and six faces, while an octahedron has eight faces and six vertices. This has implications for data structures that represent hulls—are you just looking for the hull points or do you need the defining polyhedron? Be aware of these complications of high-dimensional spaces if your problem takes you there.

  Simple $O(n \log n)$ convex-hull algorithms are available for the important special cases of two and three dimensions. In higher dimensions, things get more complicated. *Gift-wrapping* is the basic algorithm for constructing higher-dimensional convex hulls. Observe that a three-dimensional convex polyhedron is composed of two-dimensional faces, or *facets*, that are connected by one-dimensional lines or *edges*. Each edge joins exactly two facets together. Gift-wrapping starts by finding an initial facet associated with the lowest vertex and then conducting a breadth-first search from this facet to discover new, additional facets. Each edge e defining the boundary of a facet must be shared with one other facet. By running through each of the n points, we can identify which point defines the next facet with e. Essentially, we "wrap" the points one facet at a time by bending the wrapping paper around an edge until it hits the first point.

  The key to efficiency is making sure that each edge is explored only once. Implemented properly in d dimensions, gift-wrapping takes $O(n\phi_{d-1} + \phi_{d-2} \lg \phi_{d-2})$, where ϕ_{d-1} is the number of facets and ϕ_{d-2} is the number of edges in the convex hull. This can be as bad as $O(n^{\lfloor d/2 \rfloor + 1})$ when the convex hull is very complex. I recommend that you use one of the codes described below rather than roll your own.

- *Is your data given as vertices or half-spaces?* – The problem of finding the intersection of a set of n half-spaces in d dimensions (each containing the origin) is dual to that of computing convex hulls of n points in d dimensions. Thus, the same basic algorithm suffices for both problems. The necessary duality transformation is discussed in Section 17.15 (page 614). The problem of half-plane intersection differs from convex hull when no interior point is given, since the instance may be infeasible (i.e., the intersection of the half-planes empty).

- *How many points are likely to be on the hull?* – If your point set was generated "randomly," it is likely that most points lie within the interior of the hull. Planar convex-hull programs can be made more efficient in practice using the observation that the leftmost, rightmost, topmost, and bottommost points all must be on the convex hull. This usually gives a set of either three or

four distinct points, defining a triangle or quadrilateral. Any point inside this region *cannot* be on the convex hull and so can be discarded in a linear sweep through the points. Ideally, only a few points will then remain to run through the full convex-hull algorithm.

This trick can also be applied beyond two dimensions, although it loses effectiveness as the dimension increases.

- *How do I find the shape of my point set?* – Although convex hulls provide a gross measure of shape, any details associated with concavities are lost. The convex hull of the G from the example input would be indistinguishable from the convex hull of O. *Alpha-shapes* are a more general structure that can be parameterized so as to retain arbitrarily large concavities. Implementations and references on alpha-shapes are included below.

The primary convex-hull algorithm in the plane is the *Graham scan*. Graham scan starts with one point p known to be on the convex hull (say the point with the lowest x-coordinate) and sorts the rest of the points in angular order around p. Starting with a hull consisting of p and the point with the smallest angle, we proceed counterclockwise around v adding points. If the angle formed by the new point and the last hull edge is less than 180 degrees, we add this new point to the hull. If the angle formed by the new point and the last "hull" edge is greater than 180 degrees, then a chain of vertices starting from the last hull edge must be deleted to maintain convexity. The total time is $O(n \lg n)$, since the bottleneck is the cost of sorting the points around v.

The basic Graham scan procedure can also be used to construct a nonself-intersecting (or *simple*) polygon passing through all the points. Sort the points around v, but instead of testing angles, simply connect the points in angular order. Connecting this to v gives a polygon without self-intersection, although it typically has many ugly skinny protrusions.

The gift-wrapping algorithm becomes especially simple in two dimensions, since each "facet" becomes an edge, each "edge" becomes a vertex of the polygon, and the "breadth-first search" simply walks around the hull in a clockwise or counterclockwise order. The 2D gift-wrapping (or *Jarvis march*) algorithm runs in $O(nh)$ time, where h is the number of vertices on the convex hull. I recommend sticking with Graham scan unless you *really* know in advance that there cannot be too many vertices on the hull.

Implementations: The CGAL library (*www.cgal.org*) offers C++ implementations of an extensive variety of convex-hull algorithms for two, three, and arbitrary numbers of dimensions. Alternate C++ implementations of planar convex hulls includes LEDA (see Section 19.1.1 (page 658)).

Qhull [BDH97] is a popular low-dimensional, convex-hull code, optimized for from two to about eight dimensions. It is written in C and can also construct Delaunay triangulations, Voronoi vertices, furthest-site Voronoi vertices, and

half-space intersections. Qhull has been widely used in scientific applications and has a well-maintained homepage at *http://www.qhull.org/*.

O'Rourke [O'R01] provides a robust implementation of the Graham scan in two dimensions and an $O(n^2)$ implementation of an incremental algorithm for convex hulls in three dimensions. C and Java implementations are both available. See Section 19.1.10 (page 662).

For excellent alpha-shapes code, originating from the work of Edelsbrunner and Mucke [EM94], check out *http://biogeometry.duke.edu/software/alphashapes/*. Ken Clarkson's higher-dimensional convex-hull code *Hull* also does alpha-shapes, and is available at *http://www.netlib.org/voronoi/hull.html*.

Different codes are needed for enumerating the vertices of intersecting half-spaces in higher dimensions. Avis's *lhs* (*http://cgm.cs.mcgill.ca/~avis/C/lrs.html*) is an arithmetically-robust ANSI C implementation of the Avis-Fukuda reverse search algorithm for vertex enumeration/convex-hull problems. Since the polyhedra is implicitly traversed but not explicitly stored in memory, even problems with very large output sizes can sometimes be solved.

Notes: Planar convex hulls play a similar role in computational geometry as sorting does in algorithm theory. Like sorting, convex hull is a fundamental problem for which many different algorithmic approaches lead to interesting or optimal algorithms. Quickhull and mergehull are examples of hull algorithms inspired by sorting algorithms [PS85]. A simple construction involving points on a parabola presented in Section 9.2.4 (page 322) reduces sorting to convex hull, so the information-theoretic lower bound for sorting implies that planar convex hull requires $\Omega(n \lg n)$ time to compute. A stronger lower bound is established in [Yao81].

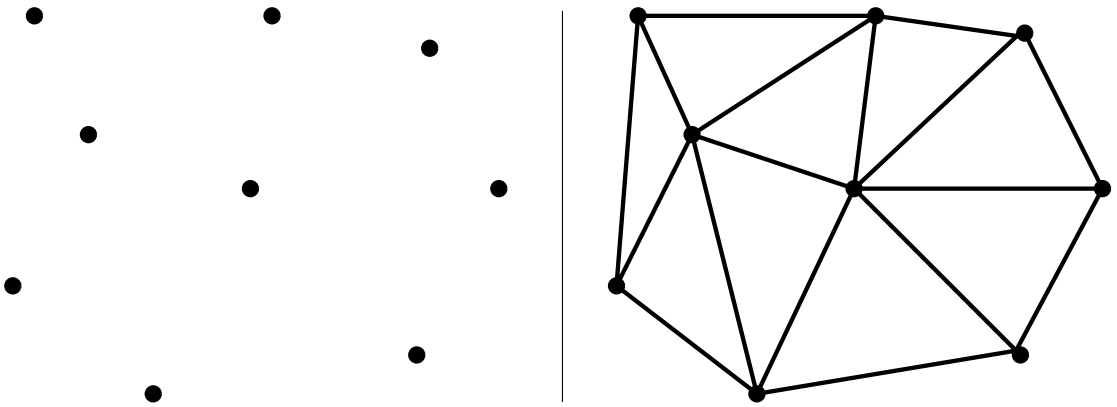
Good expositions of the Graham scan algorithm [Gra72] and the Jarvis march [Jar73] include [dBvKOS00, CLRS01, O'R01, PS85]. The optimal planar convex-hull algorithm [KS86] takes $O(n \lg h)$ time, where h is the number of hull vertices that captures the best performance of both Graham scan and gift wrapping and is (theoretically) better in between. Planar convex hull can be efficiently computed *in-place*, meaning without requiring additional memory in [BIK+04]. Seidel [Sei04] gives an up-to-date survey of convex hull algorithms and variants, particularly for higher dimensions.

Alpha-hulls, introduced in [EKS83], provide a useful notion of the shape of a point set. A generalization to three dimensions, with an implementation, is presented in [EM94].

Reverse-search algorithms for constructing convex hulls are effective in higher dimensions [AF96]. Through a clever lifting-map construction [ES86], the problem of building Voronoi diagrams in d-dimensions can be reduced to constructing convex hulls in $(d+1)$-dimensions. See Section 17.4 (page 576) for more details.

Dynamic algorithms for convex-hull maintenance are data structures that permit inserting and deleting arbitrary points while always representing the current convex hull. The first such dynamic data structure [OvL81] supported insertions and deletions in $O(\lg^2 n)$ time. More recently, Chan [Cha01] reduced the cost of such operation of near-logarithmic amortized time.

Related Problems: Sorting (see page 436), Voronoi diagrams (see page 576).



INPUT OUTPUT

17.3 Triangulation

Input description: A set of points or a polyhedron.

Problem description: Partition the interior of the point set or polyhedron into triangles.

Discussion: The first step in working with complicated geometric objects is often to break them into simple geometric objects. This makes triangulation a fundamental problem in computational geometry. The simplest geometric objects are triangles in two dimensions, and tetrahedra in three. Classical applications of triangulation include finite element analysis and computer graphics.

A particularly interesting application of triangulation is surface or function interpolation. Suppose that we have sampled the height of a mountain at a certain number of points. How can we estimate the height at any point q in the plane? We can project the sampled points on the plane, and then triangulate them. This triangulation partitions the plane into triangles, so we can estimate height by interpolating between the heights of the three points of the triangle that contain q. Furthermore, this triangulation and the associated height values define a mountain surface suitable for graphics rendering.

A triangulation in the plane is constructed by adding nonintersecting chords between the vertices until no more such chords can be added. Specific issues arising in triangulation include:

- *Are you triangulating a point set or a polygon?* – Often we are given a set of points to triangulate, as in the surface interpolation problem discussed

earlier. This involves first constructing the convex hull of the point set and then carving up the interior into triangles.

The simplest such $O(n \lg n)$ algorithm first sorts the points by x-coordinate. It then inserts them from left to right as per the convex-hull algorithm of page 105, building the triangulation by adding a chord to each point newly cut off from the hull.

- *Does the shape of the triangles in your triangulation matter?* – There are usually many different ways to partition your input into triangles. Consider a set of n points in convex position in the plane. The simplest way to triangulate them would be to add to the convex-hull diagonals from the first point to all of the others. However, this has the tendency to create skinny triangles.

  Many applications seek to to avoid skinny triangles, or equivalently, minimize small angles in the triangulation. The *Delaunay triangulation* of a point set minimizes the maximum angle over all possible triangulations. This isn't exactly what we are looking for, but it is pretty close, and the Delaunay triangulation has enough other interesting properties (including that it is dual to the Voronoi diagram) to make it the quality triangulation of choice. Further, it can be constructed in $O(n \lg n)$ time using implementations described below.

- *How can I improve the shape of a given triangulation?* – Each internal edge of any triangulation is shared between two triangles. The four vertices defining these two triangles form either (1) a convex quadrilateral, or (2) a triangle with a triangular bite taken out of it. The beauty of the convex case is that exchanging the internal edge with a chord linking the other two vertices yields a different triangulation.

  This gives us a local "edge-flip" operation for changing and possibly improving a given triangulation. Indeed, a Delaunay triangulation can be constructed from any initial triangulation by removing skinny triangles until no locally-improving exchange remains.

- *What dimension are we working in?* – Three-dimensional problems are usually harder than two-dimensional problems. The three-dimensional generalization of triangulation involves partitioning the space into four-vertex tetrahedra by adding nonintersecting faces. One important difficulty is that there is no way to tetrahedralize the interior of certain polyhedra without adding extra vertices. Furthermore, it is NP-complete to decide whether such a tetrahedralization exists, so we should not feel afraid to add extra vertices to simplify our problem.

- *What constraints does the input have?* – When we are triangulating a polygon or polyhedra, we are restricted to adding chords that do not intersect any of the boundary facets. In general, we may have a set of obstacles or

constraints that cannot be intersected by inserted chords. The best such triangulation is likely to be the so-called *constrained Delaunay triangulation*. Implementations are described next.

- *Are you allowed to add extra points, or move input vertices?* – When the shape of the triangles does matter, it might pay to strategically add a small number of extra "Steiner" points to the data set to facilitate the construction of a triangulation (say) with no small angles. As discussed above, *no* triangulation may exist for certain polyhedra without adding Steiner points.

To construct a triangulation of a convex polygon in linear time, just pick an arbitrary starting vertex v and insert chords from v to each other vertex in the polygon. Because the polygon is convex, we can be confident that none of the boundary edges of the polygon will be intersected by these chords, and that all of them lie within the polygon. The simplest algorithm for constructing general polygon triangulations tries each of the $O(n^2)$ possible chords and inserts them if they do not intersect a boundary edge or previously inserted chord. There are practical algorithms that run in $O(n \lg n)$ time and theoretically interesting algorithms that run in linear time. See the Implementations and Notes section for details.

Implementations: Triangle, by Jonathan Shewchuk, is an award-winning C language code that generates Delaunay triangulations, constrained Delaunay triangulations (forced to have certain edges), and quality-conforming Delaunay triangulations (which avoid small angles by inserting extra points). It has been widely used for finite element analysis and is fast and robust. Triangle is the first thing I would try if I needed a two-dimensional triangulation code. It is available at *http://www.cs.cmu.edu/~quake/triangle.html*.

Fortune's Sweep2 is a widely used 2D code for Voronoi diagrams and Delaunay triangulations, written in C. This code may be simpler to work with if all you need is the Delaunay triangulation of points in the plane. It is based on his sweepline algorithm [For87] for Voronoi diagrams and is available from Netlib (see Section 19.1.5 (page 659)) at *http://www.netlib.org/voronoi/*.

Mesh generation for finite element methods is an enormous field. Steve Owen's Meshing Research Corner (*http://www.andrew.cmu.edu/user/sowen/mesh.html*) provides a comprehensive overview of the literature, with links to an enormous variety of software. QMG (*http://www.cs.cornell.edu/Info/People/vavasis/qmg-home.html*) is a particularly recommended code.

Both the CGAL (*www.cgal.org*) and LEDA (see Section 19.1.1 (page 658)) libraries offer C++ implementations of an extensive variety of triangulation algorithms for two and three dimensions, including constrained and furthest site Delaunay triangulations.

Higher-dimensional Delaunay triangulations are a special case of higher-dimensional convex hulls. Qhull [BDH97] is a popular low-dimensional convex hull code, say for from two to about eight dimensions. It is written in C and can also construct Delaunay triangulations, Voronoi vertices, furthest-site Voronoi

vertices, and half-space intersections. Qhull has been widely used in scientific applications and has a well-maintained homepage at *http://www.qhull.org/*. Another choice is Ken Clarkson's higher-dimensional convex-hull code, *Hull*, available at *http://www.netlib.org/voronoi/hull.html*.

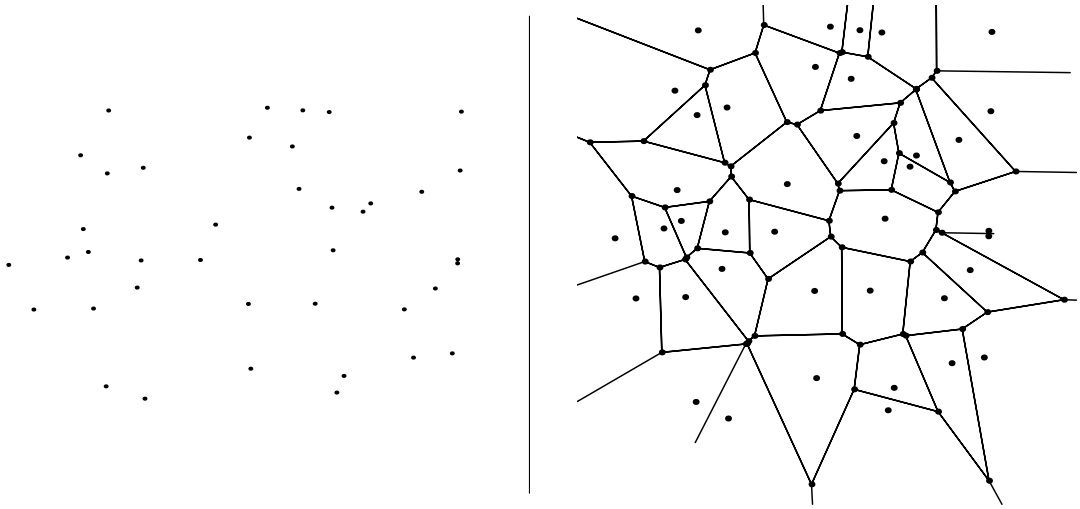
Notes: After a long search, Chazelle [Cha91] discovered a linear-time algorithm for triangulating a simple polygon. This algorithm is sufficiently hopeless to implement that it qualifies more as an existence proof. The first $O(n \lg n)$ algorithm for polygon triangulation was given by [GJPT78]. An $O(n \lg \lg n)$ algorithm by Tarjan and Van Wyk [TW88] followed before Chazelle's result. Bern [Ber04a] gives a recent survey on polygon and point-set triangulation.

The *International Meshing Roundtable* is an annual conference for people interested in mesh and grid generation. Excellent surveys on mesh generation include [Ber02, Ede06].

Linear-time algorithms for triangulating monotone polygons have been long known [GJPT78], and are the basis of algorithms for triangulating simple polygons. A polygon is monotone when there exists a direction d such that any line with slope d intersects the polygon in at most two points.

A heavily studied class of optimal triangulations seeks to minimize the total length of the chords used. The computational complexity of constructing this *minimum weight triangulation* was resolved when Rote [MR06] proved it NP-complete. Thus interest has shifted to provably good approximation algorithms. The minimum weight triangulation of a convex polygon can be found in $O(n^3)$ time using dynamic programming, as discussed in Section 8.6.1 (page 300).

Related Problems: Voronoi diagrams (see page 576), polygon partitioning (see page 601).



INPUT OUTPUT

17.4 Voronoi Diagrams

Input description: A set S of points $p_1, \ldots, p_n$.

Problem description: Decompose space into regions around each point such that all points in the region around p_i are closer to p_i than any other point in S.

Discussion: Voronoi diagrams represent the region of influence around each of a given set of sites. If these sites represent the locations of McDonald's restaurants, the Voronoi diagram partitions space into cells around each restaurant. For each person living in a particular cell, the defining McDonald's represents the closest place to get a Big Mac.

Voronoi diagrams have a surprising variety of applications:

- *Nearest neighbor search* – Finding the nearest neighbor of query point q from among a fixed set of points S is simply a matter of determining the cell in the Voronoi diagram of S that contains q. See Section 17.5 (page 580) for more details.

- *Facility location* – Suppose McDonald's wants to open another restaurant. To minimize interference with existing McDonald's, it should be located as far away from the closest restaurant as possible. This location is always at a vertex of the Voronoi diagram, and can be found in a linear-time search through all the Voronoi vertices.

- *Largest empty circle* – Suppose you needed to obtain a large, contiguous, undeveloped piece of land on which to build a factory. The same condition used to select McDonald's locations is appropriate for other undesirable facilities, namely that they be as far as possible from any relevant sites of interest. A Voronoi vertex defines the center of the largest empty circle among the points.

- *Path planning* – If the sites of S are the centers of obstacles we seek to avoid, the edges of the Voronoi diagram define the possible channels that maximize the distance to the obstacles. Thus the "safest" path among the obstacles will stick to the edges of the Voronoi diagram.

- *Quality triangulations* – In triangulating a set of points, we often desire nice, fat triangles that avoid small angles and skinny triangles. The *Delaunay triangulation* maximizes the minimum angle over all triangulations. Furthermore, it is easily constructed as the dual of the Voronoi diagram. See Section 17.3 (page 572) for details.

Each edge of a Voronoi diagram is a segment of the perpendicular bisector of two points in S, since this is the line that partitions the plane between the points. The conceptually simplest method to construct Voronoi diagrams is randomized incremental construction. To add another site to the diagram, we locate the cell that contains it and add the perpendicular bisectors separating this new site from all sites defining impacted regions. If the sites are inserted in random order, only a small number of regions are likely to be impacted with each insertion.

However, the method of choice is Fortune's sweepline algorithm, especially since robust implementations of it are readily available. The algorithm works by projecting the set of sites in the plane into a set of cones in three dimensions such that the Voronoi diagram is defined by projecting the cones back onto the plane. Advantages of Fortune's algorithm include that (1) it runs in optimal $\Theta(n \log n)$ time, (2) it is reasonable to implement, and (3) we need not store the entire diagram if we can use it as we sweep over it.

There is an interesting relationship between convex hulls in $d+1$ dimensions and Delaunay triangulations (or equivalently Voronoi diagrams) in d-dimensions. By projecting each site in E^d $(x_1, x_2, \ldots, x_d)$ into the point $(x_1, x_2, \ldots, x_d, \sum_{i=1}^{d} x_i^2)$, taking the convex hull of this $(d+1)$-dimensional point set and then projecting back into d dimensions, we obtain the Delaunay triangulation. Details are given in the Notes section, but this provides the best way to construct Voronoi diagrams in higher dimensions. Programs that compute higher-dimensional convex hulls are discussed in Section 17.2 (page 568).

Several important variations of standard Voronoi diagrams arise in practice. See the references below for details:

- *Non-Euclidean distance metrics* – Recall that Voronoi diagrams decompose space into regions of influence around each of the given sites. We have assumed that Euclidean distance measures influence, but this is inappropriate

for certain applications. If people drive to McDonald's, the time it takes to get there depends upon where the major roads are. Efficient algorithms are known for constructing Voronoi diagrams under a variety of different metrics, and for curved or constrained objects.

- *Power diagrams* – These structures decompose space into regions of influence around the sites, where the sites are no longer constrained to have all the same power. Imagine a map of radio stations operating at a given frequency. The region of influence around a station depends both on the power of its transmitter and the position/power of neighboring transmitters.

- *Kth-order and furthest-site diagrams* – The idea of decomposing space into regions sharing some property can be taken beyond closest-point Voronoi diagrams. Any point within a single cell of the kth-order Voronoi diagram shares the same set of k's closest points in S. In furthest-site diagrams, any point within a particular region shares the same furthest point in S. Point location (see Section 17.7 (page 587)) on these structures permits fast retrieval of the appropriate points.

Implementations: Fortune's Sweep2 is a widely used 2D code for Voronoi diagrams and Delaunay triangulations, written in C. This code is simple to work with if all you need is the Voronoi diagram. It is based on his sweepline algorithm [For87] for Voronoi diagrams and is available from Netlib (see Section 19.1.5 (page 659)) at *http://www.netlib.org/voronoi/*.

Both the CGAL (*www.cgal.org*) and LEDA (see Section 19.1.1 (page 658)) libraries offer C++ implementations of a variety of Voronoi diagram and Delaunay triangulation algorithms in two and three dimensions.

Higher-dimensional and furthest-site Voronoi diagrams can be constructed as a special case of higher-dimensional convex hulls. Qhull [BDH97] is a popular low-dimensional convex-hull code, useful for from two to about eight dimensions. It is written in C and can also construct Delaunay triangulations, Voronoi vertices, furthest-site Voronoi vertices, and half-space intersections. Qhull has been widely used in scientific applications and has a well-maintained homepage at *http://www.qhull.org/*. Another choice is Ken Clarkson's higher-dimensional convex-hull code, *Hull*, available at *http://www.netlib.org/voronoi/hull.html*.

Notes: Voronoi diagrams were studied by Dirichlet in 1850 and are occasionally referred to as *Dirichlet tessellations*. They are named after G. Voronoi, who discussed them in a 1908 paper. In mathematics, concepts get named after the last person to discover them.

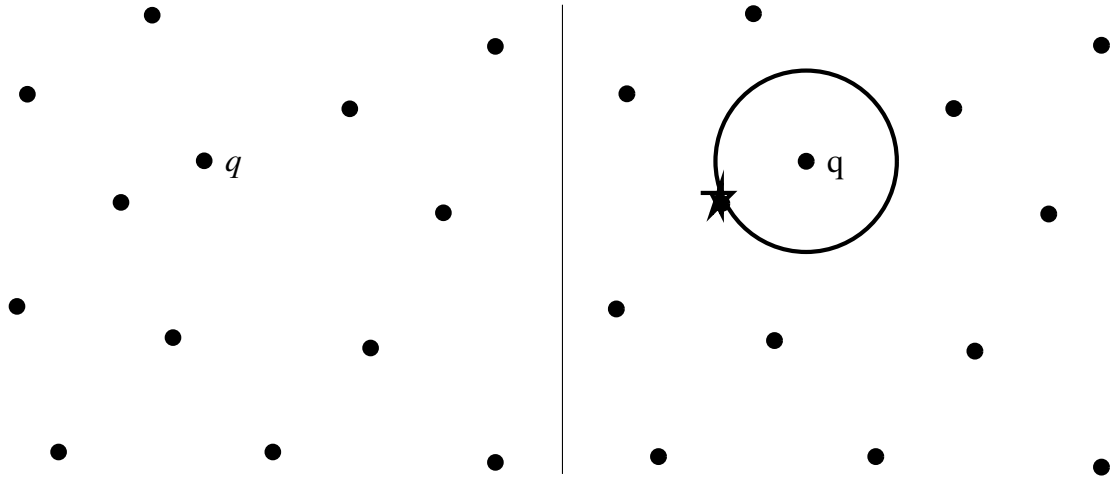
The book by Okabi, et al. [OBSC00] is the most complete treatment of Voronoi diagrams and their applications. Aurenhammer [Aur91] and Fortune [For04] provide excellent surveys on Voronoi diagrams and associated variants such as power diagrams. The first $O(n \lg n)$ algorithm for constructing Voronoi diagrams was based on divide-and-conquer and is due to Shamos and Hoey [SH75]. Good expositions of both Fortune's sweepline algorithm [For87] for constructing Voronoi diagrams in $O(n \lg n)$ and the relationship

between Delaunay triangulations and $(d+1)$-dimensional convex hulls [ES86] include [dBvKOS00, O'R01].

In a kth-order Voronoi diagram, we partition the plane such that each point in a region is closest to the same set of k sites. Using the algorithm of [ES86], the complete set of kth-order Voronoi diagrams can be constructed in $O(n^3)$ time. By doing point location on this structure, the k nearest neighbors to a query point can be found in $O(k + \lg n)$. Expositions on kth-order Voronoi diagrams include [O'R01, PS85].

The smallest enclosing circle problem can be solved in $O(n \lg n)$ time using $(n-1)$st order Voronoi diagrams [PS85]. In fact, there exists a linear-time algorithm based on low-dimensional linear programming [Meg83]. A linear algorithm for computing the Voronoi diagram of a convex polygon is given by [AGSS89].

Related Problems: Nearest neighbor search (see page 580), point location (see page 587), triangulation (see page 572).



INPUT OUTPUT

17.5 Nearest Neighbor Search

Input description: A set S of n points in d dimensions; a query point q.

Problem description: Which point in S is closest to q?

Discussion: The need to quickly find the nearest neighbor of a query point arises in a variety of geometric applications. The classic example involves designing a system to dispatch emergency vehicles to the scene of a fire. Once the dispatcher learns the location of the fire, she uses a map to find the firehouse closest to this point to minimize transportation delays. This situation occurs in any application mapping customers to service providers.

A nearest-neighbor search is also important in classification. Suppose we are given a collection of numerical data about people (say age, height, weight, years of education, and income level) each of whom has been labeled as Democrat or Republican. We seek a classifier to decide which way a given person is likely to vote. Each of the people in our data set is represented by a party-labeled point in d-dimensional space. A simple classifier can be built by assigning the new point to the party affiliation of its nearest neighbor.

Such nearest-neighbor classifiers are widely used, often in high-dimensional spaces. The vector-quantization method of image compression partitions an image into 8×8 pixel regions. This method uses a predetermined library of several thousand 8×8 pixel tiles and replaces each image region by the most similar library tile. The most similar tile is the point in 64-dimensional space that is closest to

the image region in question. Compression is achieved by reporting the identifier of the closest library tile instead of the 64 pixels, at some loss of image fidelity.

Issues arising in nearest-neighbor search include:

- *How many points are you searching?* – When your data set contains only a small number of points (say $n \leq 100$), or if only a few queries are ever destined to be performed, the simple approach is best. Compare the query point q against each of the n data points. Only when fast queries are needed for large numbers of points does it pay to consider more sophisticated methods.

- *How many dimensions are you working in?* – A nearest neighbor search gets progressively harder as the dimensionality increases. The *kd*-tree data structure, presented in Section 12.6 (page 389), does a very good job in moderate-dimensional spaces—even the plane. Still, above 20 dimensions or so, kd-tree search degenerates pretty much to a linear search through the data points. Searches in high-dimensional spaces become hard because a sphere of radius r, representing all the points with distance $\leq r$ from the center, progressively fills up less volume relative to a cube as the dimensionality increases. Thus, any data structure based on partitioning points into enclosing volumes will become progressively less effective.

  In two dimensions, Voronoi diagrams (see Section 17.4 (page 576)) provide an efficient data structure for nearest-neighbor queries. The Voronoi diagram of a point set in the plane decomposes the plane into regions such that the cell containing data point p consists of all points that are nearer to p than any other point in S. Finding the nearest neighbor of query point q reduces to finding which Voronoi diagram cell contains q and reporting the data point associated with it. Although Voronoi diagrams can be built in higher dimensions, their size rapidly grows to the point of unusability.

- *Do you really need the exact nearest neighbor?* – Finding the exact nearest neighbor in a very high dimensional space is hard work; indeed, you probably won't do better than a linear (brute force) search. However, there are algorithms/heuristics that can give you a reasonably close neighbor of your query point fairly quickly.

  One important technique is *dimension reduction*. Projections exist that map any set of n points in d-dimensions into a $d' = O(\lg n/\epsilon^2)$-dimensional space such that distance to the nearest neighbor in the low-dimensional space is within $(1+\epsilon)$ times that of the actual nearest neighbor. Projecting the points onto a random hyperplane of dimension d' in E^d will likely do the trick.

  Another idea is adding some randomness when you search your data structure. A *kd*-tree can be efficiently searched for the cell containing the query point q—a cell whose boundary points are good candidates to be close neighbors. Now suppose we search for a point q', which is a small random perturbations of q. It should land in a different but nearby cell, one of whose

boundary points might prove to be an even closer neighbor of q. Repeating such random queries gives us a way to productively use exactly as much computing time as we are willing to spend to improve the answer.

- *Is your data set static or dynamic?* – Will there be occasional insertions or deletions of new data points in your application? If these are very rare events, it might pay to rebuild your data structure from scratch each time. If they are frequent, select a version of the kd-tree that supports insertions and deletions.

The nearest neighbor graph on a set S of n points links each vertex to its nearest neighbor. This graph is a subgraph of the Delaunay triangulation, and so can be computed in $O(n \log n)$. This is quite a bargain, since it takes $\Theta(n \log n)$ time just to discover the closest pair of points in S.

As a lower bound, the closest-pair problem reduces to sorting in one dimension. In a sorted set of numbers, the closest pair corresponds to two numbers that lie next to each other, so we need only check that which is the minimum gap between the $n - 1$ adjacent pairs. The limiting case occurs when the closest pair are zero distance apart, meaning that the elements are not unique.

Implementations: *ANN* is a C++ library for both exact and approximate nearest neighbor searching in arbitrarily high dimensions. It performs well for searches over hundreds of thousands of points in up to about 20 dimensions. It supports all l_p distance norms, including Euclidean and Manhattan distance, and is available at *http://www.cs.umd.edu/~mount/ANN/*. It is the first code I would turn to for nearest neighbor search.

Samet's spatial index demos (*http://donar.umiacs.umd.edu/quadtree/*) provide a series of Java applets illustrating many variants of *kd*-trees, in association with [Sam06]. *KDTREE 2* contains C++ and Fortran 95 implementations of *kd*-trees for efficient nearest nearest neighbor search in many dimensions. See *http://arxiv.org/abs/physics/0408067*.

Ranger [MS93] is a tool for visualizing and experimenting with nearest-neighbor and orthogonal-range queries in high-dimensional data sets, using multidimensional search trees. Four different search data structures are supported by *Ranger*: naive kd-trees, median kd-trees, nonorthogonal kd-trees, and the vantage point tree. It is available in the algorithm repository *http://www.cs.sunysb.edu/~algorith*.

Nearpt3 is a special-purpose code for nearest-neighbor search of extremely large point sets in three dimensions. See *http://wrfranklin.org/Research/nearpt3*.

See Section 17.4 (page 576) for a complete collection of Voronoi diagram implementations. In particular, CGAL (*www.cgal.org*) and LEDA (see Section 19.1.1 (page 658)) provide implementations of Voronoi diagrams in C++, as well as planar point location to make effective use of them for nearest-neighbor search.

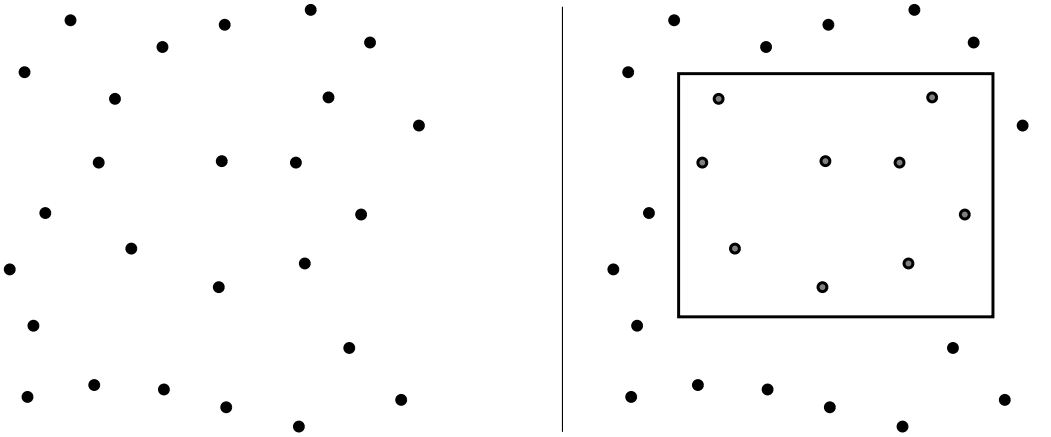
Notes: Indyk [Ind04] ably surveys recent results in approximate nearest neighbor search in high dimensions based on random projection methods. Both theoretical [IM04] and empirical [BM01] results indicate that the method preserves distances quite nicely.

The theoretical guarantees underlying Ayra and Mount's approximate nearest neighbor code *ANN* [AM93, AMN+98] are somewhat different. A sparse weighted graph structure is built from the data set, and the nearest neighbor is found by starting at a random point and greedily walking towards the query point in the graph. The closest point found over several random trials is declared the winner. Similar data structures hold promise for other problems in high-dimensional spaces. Nearest-neighbor search was a subject of the Fifth DIMACS challenge, as reported in [GJM02].

Samet [Sam06] is the best reference on kd-trees and other spatial data structures. All major (and many minor) variants are developed in substantial detail. A shorter survey [Sam05] is also available. The technique of using random perturbations of the query point is due to [Pan06].

Good expositions on finding the closest pair of points in the plane [BS76] include [CLRS01, Man89]. These algorithms use a divide-and-conquer approach instead of just selecting from the Delaunay triangulation.

Related Problems: Kd-trees (see page 389), Voronoi diagrams (see page 576), range search (see page 584).



INPUT OUTPUT

17.6 Range Search

Input description: A set S of n points in E^d and a query region Q.

Problem description: What points in S lie within Q?

Discussion: Range-search problems arise in database and geographic information system (GIS) applications. Any data object with d numerical fields, such as person with height, weight, and income, can be modeled as a point in d-dimensional space. A *range query* describes a region in space and asks for all points or the number of points in the region. For example, asking for all people with income between \$0 and \$10,000, with height between 6'0" and 7'0", and weight between 50 and 140 lbs., defines a box containing people whose wallet and body are both thin.

The difficulty of range search depends on several factors:

- *How many range queries are you going to perform?* – The simplest approach to range search tests each of the n points one by one against the query polygon Q. This works just fine when the number of queries will be small. Algorithms to test whether a point is within a given polygon are presented in Section 17.7 (page 587).

- *What shape is your query polygon?* – The easiest regions to query against are *axis-parallel rectangles*, because the inside/outside test reduces to verifying whether each coordinate lies within a prescribed range. The input-output figure illustrates such an *orthogonal range query*.

When querying against a nonconvex polygon, it will pay to partition your polygon into convex pieces or (even better) triangles and then query the point set against each one of the pieces. This works because it is quick and easy to test whether a point lies inside a convex polygon. Algorithms for such convex decompositions are discussed in Section 17.11 (page 601).

- *How many dimensions?* – A general approach to range queries builds a kd-tree on the point set, as discussed in Section 12.6 (page 389). A depth-first traversal of the kd-tree is performed for the query, with each tree node expanded only if the associated rectangle intersects the query region. The entire tree might have to be traversed for sufficiently large or misaligned query regions, but in general kd-trees lead to an efficient solution. Algorithms with more efficient worst-case performance are known in two dimensions, but kd-trees are likely to work just fine in the plane. In higher dimensions, they provide the only viable solution to the problem.

- *Is your point set static, or might there be insertions/deletions?* – A clever practical approach to range search and many other geometric searching problems is based on Delaunay triangulations. Delaunay triangulation edges connect each point p to nearby points, including its nearest neighbor. To perform a range query, we start by using planar point location (see Section 17.7 (page 587)) to quickly identify a triangle within the region of interest. We then do a depth-first search around a vertex of this triangle, pruning the search whenever it visits a point too distant to have interesting undiscovered neighbors. This should be efficient, because the total number of points visited should be roughly proportional to the number within the query region.

  One nice thing about this approach is that it is relatively easy to employ "edge-flip" operations to fix up a Delaunay triangulation following a point insertion or deletion. See Section 17.3 (page 572) for more details.

- *Can I just count the number of points in a region, or do I have to identify them?* – For many applications, it suffices to count the number of points in a region instead of returning them. Harkening back to the introductory example, we may want to know whether there are more thin/poor people or rich/fat ones. The need to find the densest or emptiest region in space often arises, and this problem can be solved by counting range queries.

  A nice data structure for efficiently answering such aggregate range queries is based on the dominance ordering of the point set. A point x is said to *dominate* point y if y lies both below and to the left of x. Let $DOM(p)$ be a function that counts the number of points in S that are dominated by p. The number of points m in the orthogonal rectangle defined by $x_{\min} \leq x \leq x_{\max}$ and $y_{\min} \leq y \leq y_{\max}$ is given by

$$m = DOM(x_{\max}, y_{\max}) - DOM(x_{\max}, y_{\min}) - DOM(x_{\min}, y_{\max}) + DOM(x_{\min}, y_{\min})$$

The second additive term corrects for the points for the lower left-hand corner that have been subtracted away twice.

To answer arbitrary dominance queries efficiently, partition the space into n^2 rectangles by drawing a horizontal and vertical line through each of the n points. The set of dominated points is identical for each point in any rectangle, so the dominance count of the lower left-hand corner of each rectangle can be precomputed, stored, and reported for any query point within it. Range queries reduce to binary search and thus take $O(\lg n)$ time. Unfortunately, this data structure takes quadratic space. However, the same idea can be adapted to kd-trees to create a more space-efficient search structure.

Implementations: Both CGAL (*www.cgal.org*) and LEDA (see Section 19.1.1 (page 658)) use a dynamic Delaunay triangulation data structure to support circular, triangular, and orthogonal range queries. Both libraries also provide implementations of range tree data structures, which support orthogonal range queries in $O(k + \lg^2 n)$ time where n is the complexity of the subdivision and k is the number of points in the rectangular region.

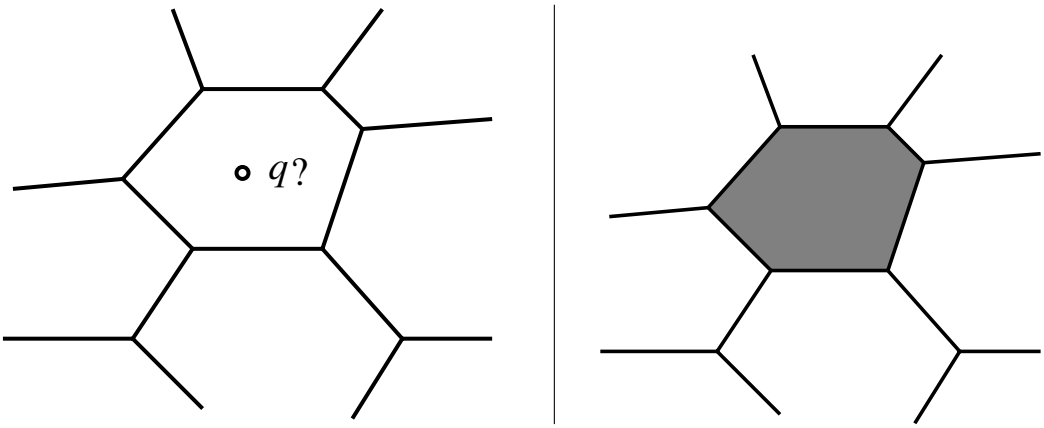
ANN is a C++ library for both exact and approximate nearest neighbor searching in arbitrarily high dimensions. It performs well for searches over hundreds of thousands of points in up to about 20 dimensions. It supports fixed-radius, nearest-neighbor queries over all l_p distance norms, which can be used to approximate circular and orthogonal range queries under the l_2 and l_1 norms, respectively. *ANN* is available at *http://www.cs.umd.edu/∼mount/ANN/*.

Ranger is a tool for visualizing and experimenting with nearest-neighbor and orthogonal-range queries in high-dimensional data sets, using multidimensional search trees. Four different search data structures are supported by *Ranger*: naive kd-trees, median kd-trees, nonorthogonal kd-trees, and the vantage point tree. For each of these, *Ranger* supports queries in up to 25 dimensions under any Minkowski metric. It is available in the algorithm repository.

Notes: Good expositions on data structures with worst-case $O(\lg n + k)$ performance for orthogonal-range searching [Wil85] include [dBvKOS00, PS85]. Exposition on *kd*-trees for orthogonal range queries in two dimensions appear in [dBvKOS00, PS85]. Their worst-case performance can be very bad; [LW77] describes an instance in two dimensions requiring $O(\sqrt{n})$ time to report that a rectangle is empty.

The problem becomes considerably more difficult for nonorthogonal range queries, where the query region is not an axis-aligned rectangle. For half-plane intersection queries, $O(\lg n)$ time and linear space suffice [CGL85]; for range searching with simplex query regions (such as a triangle in the plane), lower bounds preclude efficient worst-case data structures. See Agrawal [Aga04] for a survey and discussion.

Related Problems: Kd-trees (see page 389), point location (see page 587).



INPUT OUTPUT

17.7 Point Location

Input description: A decomposition of the plane into polygonal regions and a query point q.

Problem description: Which region contains the query point q?

Discussion: Point location is a fundamental subproblem in computational geometry, usually needed as an ingredient to solve larger geometric problems. In a typical police dispatch system, the city will be partitioned into different precincts or districts. Given a map of regions and a query point (the crime scene), the system must identify which region contains the point. This is exactly the problem of planar point location. Variations include:

- *Is a given point inside or outside of polygon P?* – The simplest version of point location involves only two regions, inside-P and outside-P, and asks which contains a given query point. For polygons with lots of narrow spirals, this can be surprisingly difficult to tell by inspection. The secret to doing it both by eye or machine is to draw a ray starting from the query point and ending beyond the furthest extent of the polygon. Count the number of times the polygon crosses through an edge. The query point will lie within the polygon iff this number is odd. The case of the line passing through a vertex instead of an edge is evident from context, since we are counting the number of times we pass through the boundary of the polygon. Testing each of the n edges for intersection against the query ray takes $O(n)$ time. Faster

algorithms for convex polygons are based on binary search, and take $O(\lg n)$ time.

- *How many queries must be performed?* – It is always possible to perform this inside-polygon test separately on each region in a given planar subdivision. However, it would be wasteful to perform many such point location queries on the same subdivision. It would be much better to construct a grid-like or tree-like data structure on top of our subdivision to get us near the correct region quickly. Such search structures are discussed in more detail below.

- *How complicated are the regions of your subdivision?* – More sophisticated inside-outside tests are required when the regions of your subdivision are arbitrary polygons. By triangulating all polygonal regions first, each inside-outside test reduces to testing whether a point is in a triangle. Such tests can be made particularly fast and simple, at the minor cost of recording the full polygon name for each triangle. An added benefit is that the smaller your regions are, the better grid-like or tree-like superstructures are likely to perform. Some care should be taken when you triangulate to avoid long skinny triangles, as discussed in Section 17.3 (page 572).

- *How regularly sized and spaced are your regions?* – If all resulting triangles are about the same size and shape, the simplest point location method imposes a regularly-spaced $k \times k$ grid of horizontal and vertical lines over the entire subdivision. For each of the k^2 rectangular regions, we maintain a list of all the regions that are at least partially contained within the rectangle. Performing a point location query in such a *grid file* involves a binary search or hash table lookup to identify which rectangle contains query point q, and then searching each region in the resulting list to identify the right one.

  Such grid files can perform very well, provided that each triangular region overlaps only a few rectangles (thus minimizing storage space) and each rectangle overlaps only a few triangles (thus minimizing search time). Whether it performs well depends on the regularity of your subdivision. Some flexibility can be achieved by spacing the horizontal lines irregularly, depending upon where the regions actually lie. The *slab method*, discussed below, is a variation on this idea that guarantees worst-case efficient point location at the cost of quadratic space.

- *How many dimensions will you be working in?* – In three or more dimensions, some flavor of kd-tree will almost certainly be the point-location method of choice. They may also be the right answer for planar subdivisions that are too irregular for grid files.

  Kd-trees, described in Section 12.6 (page 389), decompose the space into a hierarchy of rectangular boxes. At each node in the tree, the current box is split into a small number (typically 2, 4, or 2^d for dimension d) of smaller boxes. Each leaf box is labeled with the (small) set regions that are at least

partially contained in the box. The point location search starts at the root of the tree and keeps traversing down the child whose box contains the query point q. When the search hits a leaf, we test each of the relevant regions to see which one of them contains q. As with grid files, we hope that each leaf contains a small number of regions and that each region does not cut across too many leaf cells.

- *Am I close to the right cell?* – Walking is a simple point-location technique that might even work well beyond two dimensions. Start from an arbitrary point p in an arbitrary cell, hopefully close to the query point q. Construct the line (or ray) from p to q and identify which face of the cell this hits (a so-called *ray shooting query*). Such queries take constant time in triangulated arrangements.

  Proceeding to the neighboring cell through this face gets us one step closer to the target. The expected path length will be $O(n^{1/d})$ for sufficiently regular d-dimensional arrangements, although linear in the worst case.

The simplest algorithm to guarantee $O(\lg n)$ worst-case access is the *slab* method, which draws horizontal lines through each vertex, thus creating $n + 1$ "slabs" between the lines. Since the slabs are defined by horizontal lines, finding the slab containing a particular query point can be done using a binary search on the y-coordinate of q. Since there can be no vertices within any slab, the region containing a point within a slab can be identified by a second binary search on the edges that cross the slab. The catch is that a binary search tree must be maintained for each slab, for a worst-case of $O(n^2)$ space. A more space-efficient approach based on building a hierarchy of triangulations over the regions also achieves $O(\lg n)$ for search and is discussed in the notes below.

Worst-case efficient computational geometry methods either require a lot of storage or are fairly complicated to implement. We identify implementations of worst-case methods below, which are worth at least experimenting with. However, we recommend kd-trees for most general point-location applications.

Implementations: Both CGAL (*www.cgal.org*) and LEDA (see Section 19.1.1 (page 658)) provide excellent support for maintaining planar subdivisions in C++. CGAL favors a jump-and-walk strategy, although a worst-case logarithmic search is also provided. LEDA implements expected $O(\lg n)$ point location using partially persistent search trees.

ANN is a C++ library for both exact and approximate nearest-neighbor searching in arbitrarily high dimensions. It can be used to quickly identify a nearby cell boundary point to begin walking from. Check it out at *http://www.cs.umd.edu/~mount/ANN/*.

Arrange is a package for maintaining arrangements of polygons in either the plane or on the sphere. Polygons may be degenerate, and hence represent arrangements of lines. A randomized incremental construction algorithm is used, and efficient point location on the arrangement is supported.

Arrange is written in C by Michael Goldwasser and is available from *http://euler.slu.edu/∼goldwasser/publications/*.

Routines (in C) to test whether a point lies in a simple polygon have been provided by [O'R01, SR03].

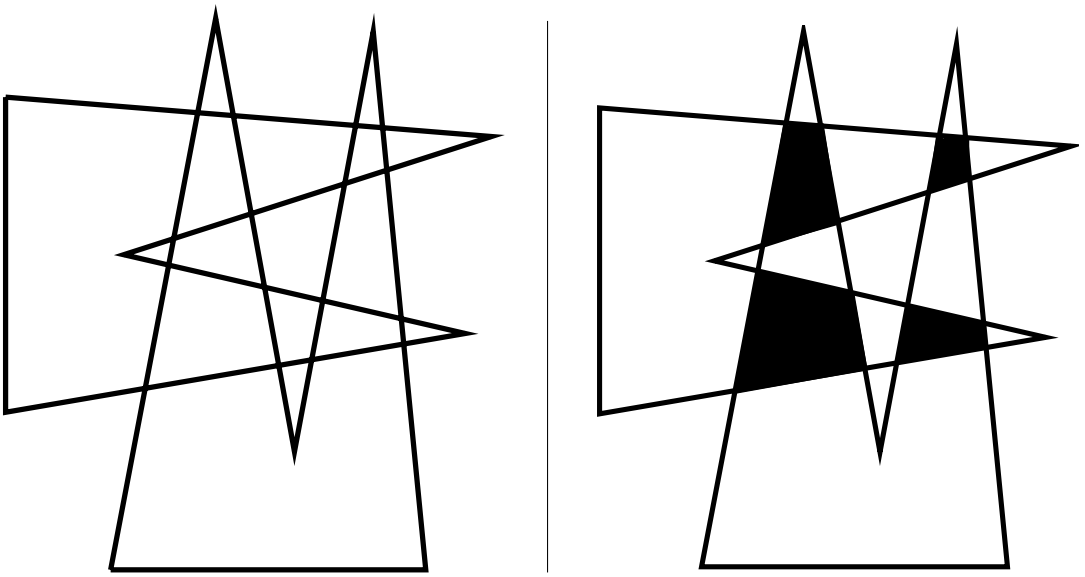
Notes: Snoeyink [Sno04] gives an excellent survey of the state-of-the-art in point location, both theoretical and practical. Very thorough treatments of deterministic planar-point location data structures are provided by [dBvKOS00, PS85].

Tamassia and Vismara [TV01] use planar point location as a case study of geometric algorithm engineering, in Java. An experimental study of algorithms for planar point location is described in [EKA84]. The winner was a bucketing technique akin to the grid file.

The elegant triangle refinement method of Kirkpatrick [Kir83] builds a hierarchy of triangulations above the actual planar subdivision such that each triangle on a given level intersects only a constant number of triangles on the following level. Since each triangulation is a fraction of the size of the subsequent one, the total space is obtained by summing up a geometric series and hence is linear. Furthermore, the height of the hierarchy is $O(\lg n)$, ensuring fast query times. An alternative algorithm realizing the same time bounds is [EGS86]. The slab method described above is due to [DL76] and is presented in [PS85]. Expositions on the inside-outside test for simple polygons include [Hai94, O'R01, PS85, SR03].

More recently, there has been interest in dynamic data structures for point location, which support fast incremental updates of the planar subdivision (such as insertions and deletions of edges and vertices) as well as fast point location. Chiang and Tamassia's [CT92] survey is an appropriate place to begin, with updated references in [Sno04].

Related Problems: Kd-trees (see page 389), Voronoi diagrams (see page 576), nearest neighbor search (see page 580).



INPUT OUTPUT

17.8 Intersection Detection

Input description: A set S of lines and line segments $l_1, \ldots, l_n$ or a pair of polygons or polyhedra P_1 and P_2.

Problem description: Which pairs of line segments intersect each other? What is the intersection of P_1 and P_2?

Discussion: Intersection detection is a fundamental geometric primitive with many applications. Picture a virtual-reality simulation of an architectural model for a building. The illusion of reality vanishes the instant the virtual person walks through a virtual wall. To enforce such physical constraints, any such intersection between polyhedral models must be immediately detected and the operator notified or constrained.

Another application of intersection detection is design rule checking for VLSI layouts. A minor design defect resulting in two crossing metal strips could short out the chip, but such errors can be detected before fabrication, using programs to find all intersections between line segments.

Issues arising in intersection detection include:

- *Do you want to compute the intersection or just report it?* – We distinguish between intersection detection and computing the actual intersection.

Detecting that an intersection exists can be substantially easier, and often suffices. For the virtual-reality application, it might not be important to know exactly where we hit the wall—just that we hit it.

- *Are you intersecting lines or line segments?* – The big difference here is that any two lines with different slopes intersect in at exactly one point. All the points of intersections can be found in $O(n^2)$ time by comparing each pair of lines. Constructing the arrangement of the lines provides more information than just the intersection points, and is discussed in Section 17.15 (page 614).

  Finding all the intersections between n line segments is considerably more challenging. Even the basic primitive of testing whether two line segments intersect is not as trivial, as discussed in Section 17.1 (page 564). Of course, we could explicitly test each line segment pair and thus find all intersections in $O(n^2)$ time, but faster algorithms exist when there are few intersection points.

- *How many intersection points do you expect?* – In VLSI design-rule checking, we expect the set of line segments to have few if any intersections. What we seek is an algorithm whose time is *output sensitive*, taking time proportional to the number of intersection points.

  Such output-sensitive algorithms exist for line-segment intersection. The fastest algorithm takes $O(n \lg n + k)$ time, where k is the number of intersections. These algorithms are based on the planar sweepline approach.

- *Can you see point x from point y?* – Visibility queries ask whether vertex x can see vertex y unobstructed in a room full of obstacles. This can be phrased as the following line-segment intersection problem: does the line segment from x to y intersect any obstacle? Such visibility problems arise in robot motion planning (see Section 17.14) and in hidden-surface elimination for computer graphics.

- *Are the intersecting objects convex?* – Better intersection algorithms exist when the line segments form the boundaries of polygons. The critical issue here becomes whether the polygons are convex. Intersecting a convex n-gon with a convex m-gon can be done in $O(n + m)$ time, using the sweepline algorithm discussed next. This is possible because the intersection of two convex polygons must form another convex polygon with at most $n + m$ vertices.

  However, the intersection of two nonconvex polygons is not so well behaved. Consider the intersection of two "combs" generalizing the Picasso-like frontspiece to this section. As illustrated, the intersection of nonconvex polygons may be disconnected and have quadratic size in the worst case.

Intersecting polyhedra is more complicated than polygons, because two polyhedra can intersect even when no edges do. Consider the example of a needle piercing the interior of a face. In general, however, the same issues arise for both polygons and polyhedra.

- *Are you searching for intersections repeatedly with the same basic objects?* – In the walk-through application just described, the room and the objects in it don't change between one scene and the next. Only the person moves, and intersections are rare.

  One common technique is to approximate the objects in the scene by simpler objects that enclose them, such as boxes. Whenever two enclosing boxes intersect, then the underlying objects *might* intersect, and so further work is necessary to decide the issue. However, it is much more efficient to test whether simple boxes intersect than more complicated objects, so we win if collisions are rare. Many variations on this theme are possible, but this idea leads to large performance improvements for complicated environments.

Planar sweep algorithms can be used to efficiently compute the intersections among a set of line segments, or the intersection/union of two polygons. These algorithms keep track of interesting changes as we sweep a vertical line from left to right over the data. At its leftmost position, the line intersects nothing, but as it moves to the right, it encounters a series of events:

- *Insertion* – The leftmost point of a line segment may be encountered, and it is now available to intersect some other line segment.

- *Deletion* – The rightmost point of a line segment is encountered. This means that we have completely swept over the segment, and so it can be deleted from further consideration.

- *Intersection* – If the active line segments that intersect the sweep line are maintained as sorted from top to bottom, the next intersection must occur between neighboring line segments. After this intersection, these two line segments swap their relative order.

Keeping track of what is going on requires two data structures. The future is maintained by an *event queue*, or a priority queue ordered by the x-coordinate of all possible future events of interest: insertion, deletion, and intersection. See Section 12.2 (page 373) for priority queue implementations. The present is represented by the *horizon*—an ordered list of line segments intersecting the current position of the sweepline. The horizon can be maintained using any dictionary data structure, such as a balanced tree.

To adapt this approach to compute the intersection or union of polygons, we modify the processing of the three basic event types. This algorithm can be considerably simplified for pairs of convex polygons, since (1) at most four polygon

edges intersect the sweepline, so no horizon data structure is needed, and (2) no event-queue sorting is needed, since we can start from the leftmost vertex of each polygon and proceed to the right by following the polygonal ordering.

Implementations: Both LEDA (see Section 19.1.1 (page 658)) and CGAL (*www.cgal.org*) offers extensive support for line segment and polygonal intersection. In particular, they provide a C++ implementation of the Bentley-Ottmann sweepline algorithm [BO79], finding all k intersection points between n line segments in the plane in $O((n+k)\lg n)$ time.

O'Rourke [O'R01] provides a robust program in C to compute the intersection of two convex polygons. See Section 19.1.10 (page 662).

The University of North Carolina GAMMA group has produced several efficient collision detection libraries, of which *SWIFT++* [EL01] is the most recent member. It can detect intersection, compute approximate/exact distances between objects, and determine object-pair contacts in scenes composed of rigid polyhedral models. See *http://www.cs.unc.edu/∼geom/collide/* for pointers to all of these libraries.

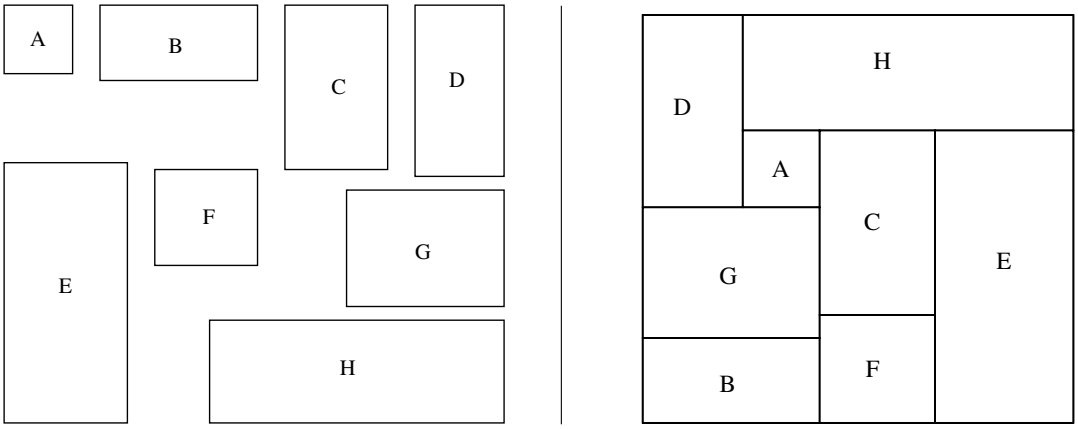
Finding the mutual intersection of a collection of half-spaces is a special case of the convex-hulls, and Qhull [BDH97] is convex hull code of choice for general dimensions. Qhull has been widely used in scientific applications and has a well-maintained homepage at *http://www.qhull.org/*.

Notes: Mount [Mou04] is an excellent survey of algorithms for computing intersections of geometric objects such as line segments, polygons, and polyhedra. Books with chapters discussing such problems include [dBvKOS00, CLRS01, PS85]. Preparata and Shamos [PS85] provide a good exposition on the special case of finding intersections and unions of axis-oriented rectangles—a problem that arises often in VLSI design.

An optimal $O(n \lg n + k)$ algorithm for computing line segment intersections is due to Chazelle and Edelsbrunner [CE92]. Simpler, randomized algorithms achieving the same time bound are thoroughly presented by Mulmuley [Mul94].

Lin and Manocha [LM04] survey techniques and software for collision detection.

Related Problems: Maintaining arrangements (see page 614), motion planning (see page 610).



17.9 Bin Packing

Input description: A set of n items with sizes $d_1, \ldots, d_n$. A set of m bins with capacity $c_1, \ldots, c_m$.

Problem description: Store all the items using the smallest number of bins.

Discussion: Bin packing arises in a variety of packaging and manufacturing problems. Suppose that you are manufacturing widgets cut from sheet metal or pants cut from cloth. To minimize cost and waste, we seek to lay out the parts so as to use as few fixed-size metal sheets or bolts of cloth as possible. Identifying which part goes on which sheet in which location is a bin-packing variant called the *cutting stock* problem. Once our widgets have been successfully manufactured, we are faced with another bin-packing problem—namely how best to fit the boxes into trucks to minimize the number of trucks needed to ship everything.

Even the most elementary-sounding bin-packing problems are NP-complete; see the discussion of integer partition in Section 13.10 (page 427). Thus, we are doomed to think in terms of heuristics instead of worst-case optimal algorithms. Fortunately, relatively simple heuristics tend to work well on most bin-packing problems. Further, many applications have peculiar, problem-specific constraints that tend to frustrate sophisticated algorithms for bin packing. The following factors will affect the choice of heuristic:

- *What are the shapes and sizes of the objects?* – The character of a bin-packing problem depends greatly on the shapes of the objects to be packed. Solving a standard jigsaw puzzle is a much different problem than packing squares

into a rectangular box. In *one-dimensional bin packing*, each object's size is given simply as an integer. This is equivalent to packing boxes of equal width into a chimney of that width, and makes it a special case of the knapsack problem of Section 13.10 (page 427).

When all the boxes are of identical size and shape, repeatedly filling each row gives a reasonable, but not necessarily optimal, packing. Consider trying to fill a 3×3 square with 2×1 bricks. You can only pack three bricks using one orientation, while four bricks suffice with two.

- *Are there constraints on the orientation and placement of objects?* – Many boxes are labeled "this side up" (imposing an orientation on the box) or "do not stack" (requiring them sit on top of any box pile). Respecting these constraints restricts our flexibility in packing and hence will increase in the number of trucks needed to send out certain shipments. Most shippers solve the problem by ignoring the labels. Indeed, your task will be simpler if you don't have to worry about the consequences of them.

- *Is the problem on-line or off-line?* – Do we know the complete set of objects to pack at the beginning of the job (an *off-line* problem)? Or will we get them one at a time and deal with them as they arrive (an *on-line* problem)? The difference is important, because we can do a better job packing when we can take a global view and plan ahead. For example, we can arrange the objects in an order that will facilitate efficient packing, perhaps by sorting them from biggest to smallest.

The standard off-line heuristics for bin packing order the objects by size or shape and then insert them into bins. Typical insertion rules are (1) select the first or leftmost bin the object fits in, (2) select the bin with the most room, (3) select the bin that provides the tightest fit, or (4) select a random bin.

Analytical and empirical results suggest that *first-fit decreasing* is the best heuristic. Sort the objects in decreasing order of size, so that the biggest object is first and the smallest last. Insert each object one by one into the first bin that has room for it. If no bin has room, we must start another bin. In the case of one-dimensional bin packing, this can never require more than 22% more bins than necessary and usually does much better. First-fit decreasing has an intuitive appeal to it, for we pack the bulky objects first and hope that little objects can fill up the cracks.

First-fit decreasing is easily implemented in $O(n \lg n + bn)$ time, where $b \leq \min(n, m)$ is the number of bins actually used. Simply do a linear sweep through the bins on each insertion. A faster $O(n \lg n)$ implementation is possible by using a binary tree to keep track of the space remaining in each bin.

We can fiddle with the insertion order in such a scheme to deal with problem-specific constraints. For example, it is reasonable to take "do not stack" boxes last (perhaps after artificially lowering the height of the bins to leave some room up

top to work with) and to place fixed-orientation boxes at the beginning (so we can use the extra flexibility later to stick boxes on top).

Packing boxes is much easier than packing arbitrary geometric shapes, enough so that a general approach packs each part into its own box and then packs the boxes. Finding an enclosing rectangle for a polygonal part is easy; just find the upper, lower, left, and right tangents in a given orientation. Finding the orientation and minimizing the area (or volume) of such a box is more difficult, but can be done in both two and three dimensions [O'R85]. See the Implementations section for a fast approximation to minimum enclosing box.

In the case of nonconvex parts, considerable useful space can be wasted in the holes created by placing the part in a box. One solution is to find the *maximum empty rectangle* within each boxed part and use this to contain other parts if it is sufficiently large. More advanced solutions are discussed in the references.

Implementations: Martello and Toth's collection of Fortran implementations of algorithms for a variety of knapsack problem variants are available at *http://www.or.deis.unibo.it/kp.html*. An electronic copy of [MT90a] has also been generously made available.

David Pisinger maintains a well-organized collection of C-language codes for knapsack problems and related variants like bin packing and container loading. These are available at *http://www.diku.dk/∼pisinger/codes.html*.

A first step towards packing arbitrary shapes packs each in its own minimum volume box. For a code to find an approximation to the optimal packing, see *http://valis.cs.uiuc.edu/∼sariel/research/papers/00/diameter/diam_prog.html*. This algorithm runs in near-linear time [BH01].

Notes: See [CFC94, CGJ96, LMM02] for surveys of the extensive literature on bin packing and the cutting stock problem. Keller, Pferschy, and Psinger [KPP04] is the most current reference on the knapsack problem and variants. Experimental results on bin-packing heuristics include [BJLM83, MT87].

Efficient algorithms are known for finding the largest empty rectangle in a polygon [DMR97] and point set [CDL86].

Sphere packing is an important and well-studied special case of bin packing, with applications to error-correcting codes. Particularly notorious was the "Kepler conjecture"—the problem of establishing the densest packing of unit spheres in three dimensions. This conjecture was finally settled by Hales and Ferguson in 1998; see [Szp03] for an exposition. Conway and Sloane [CS93] is the best reference on sphere packing and related problems.

Milenkovic has worked extensively on two-dimensional bin-packing problems for the apparel industry, minimizing the amount of material needed to manufacture pants and other clothing. Reports of this work include [DM97, Mil97].

Related Problems: Knapsack problem (see page 427), set packing (see page 625).

INPUT OUTPUT

17.10 Medial-Axis Transform

Input description: A polygon or polyhedron P.

Problem description: What are the set of points within P that have more than one closest point on the boundary of P?

Discussion: The medial-axis transformation is useful in *thinning* a polygon, or as is sometimes said, finding its *skeleton*. The goal is to extract a simple, robust representation of the shape of the polygon. The thinned versions of the letters A and B capture the essence of their shape, and would be relatively unaffected by changing the thickness of strokes or by adding font-dependent flourishes such as serifs. The skeleton also represents the center of the given shape, a property that leads to other applications like shape reconstruction and motion planning.

The medial-axis transformation of a polygon is always a tree, making it fairly easy to use dynamic programming to measure the "edit distance" between the skeleton of a known model and the skeleton of an unknown object. Whenever the two skeletons are close enough, we can classify the unknown object as an instance of our model. This technique has proven useful in computer vision and in optical character recognition. The skeleton of a polygon with holes (like the A and B) is not a tree but an embedded planar graph, but it remains easy to work with.

There are two distinct approaches to computing medial-axis transforms, depending upon whether your input is arbitrary geometric points or pixel images:

- *Geometric data* – Recall that the Voronoi diagram of a point set S (see Section 17.4 (page 576)) decomposes the plane into regions around each point $s_i \in S$ such that points within the region around s_i are closer to s_i than to any other site in S. Similarly, the Voronoi diagram of a set of line segments L decomposes the plane into regions around each line segment $l_i \in L$ such that all points within the region around l_i are closer to l_i than to any other site in L.

Any polygon is defined by a collection of line segments such that l_i shares a vertex with l_{i+1}. The medial-axis transform of a polygon P is simply the portion of the line-segment Voronoi diagram that lies within P. Any line-segment Voronoi diagram code thus suffices to do polygon thinning.

The *straight skeleton* is a structure related to the medial axis of a polygon, except that the bisectors are not equidistant to its defining edges but instead to the supporting lines of such edges. The straight skeleton, medial axis and Voronoi diagram are all identical for convex polygons, but in general skeleton bisectors may not be located in the center of the polygon. However, the straight skeleton is quite similar to a proper medial axis transform but is easier to compute. In particular, all edges in a straight skeleton are polygonal. See the Notes section for references with more details on how to compute it.

- *Image data* – Digitized images can be interpreted as points sitting at the lattice points on an integer grid. Thus, we could extract a polygonal description from boundaries in an image and feed it to the geometric algorithms just described. However, the internal vertices of the skeleton will most likely not lie at grid points. Geometric approaches to image processing problems often flounder because images are pixel-based and not continuous.

  A direct pixel-based approach for constructing a skeleton implements the "brush fire" view of thinning. Imagine a fire burning along all edges of the polygon, racing inward at a constant speed. The skeleton is marked by all points where two or more fires meet. The resulting algorithm traverses all the boundary pixels of the object, identifies those vertices as being in the skeleton, deletes the rest of the boundary, and repeats. The algorithm terminates when all pixels are extreme, leaving an object only one or two pixels thick. When implemented properly, this takes linear time in the number of pixels in the image.

  Algorithms that explicitly manipulate pixels tend to be easy to implement, because they avoid complicated data structures. However, the geometry doesn't work out exactly right in such pixel-based approaches. For example, the skeleton of a polygon is no longer always a tree or even necessarily connected, and the points in the skeleton will be close-to-but-not-quite equidistant to two boundary edges. Since you are trying to do continuous geometry in a discrete world, there is no way to solve the problem completely. You just have to live with it.

Implementations: CGAL (*www.cgal.org*) includes a package for computing the straight skeleton of a polygon P. Associated with it are routines for constructing offset contours defining the polygonal regions within P whose points are at least distance d from the boundary.

VRONI [Hel01] is a robust and efficient program for computing Voronoi diagrams of line segments, points, and arcs in the plane. It can readily compute

medial-axis transforms of polygons since it can construct Voronoi diagrams of arbitrary line segments. *VRONI* has been tested on thousands of synthetic and real-world data sets, some with over a million vertices. For more information, see *http://www.cosy.sbg.ac.at/~held/projects/vroni/vroni.html*. Other programs for constructing Voronoi diagrams are discussed in Section 17.4 (page 576).

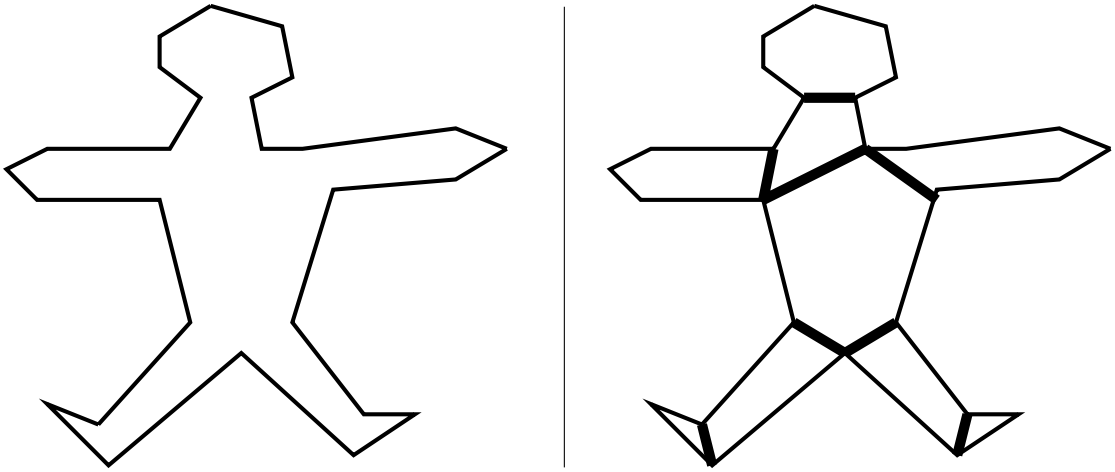
Programs that reconstruct or interpolate point clouds often are based on medial axis transforms. *Cocone* (*http://www.cse.ohio-state.edu/~tamaldey/cocone.html*) constructs an approximate medial-axis transform of the polyhedral surface it interpolates from points in E^3. See [Dey06] for the theory behind *Cocone*. *Powercrust* [ACK01a, ACK01b] constructs a discrete approximation to the medial-axis transform, and then reconstructs the surface from this transform. When the point samples are sufficiently dense, the algorithm is guaranteed to produce a geometrically and topologically correct approximation to the surface. It is available at *http://www.cs.utexas.edu/users/amenta/powercrust/*.

Notes: For a comprehensive survey of thinning approaches in image processing, see [LLS92, Ogn93]. The medial axis transformation was introduced for shape similarity studies in biology [Blu67]. Applications of the medial-axis transformation in pattern recognition are discussed in [DHS00]. The medial axis transformation is fundamental to the power crust algorithm for surface reconstruction from sampled points; see [ACK01a, ACK01b]. Good expositions on the medial-axis transform include [dBvKOS00, O'R01, Pav82].

The medial-axis of a polygon can be computed in $O(n \lg n)$ time for arbitrary n-gons [Lee82], although linear-time algorithms exist for convex polygons [AGSS89]. An $O(n \lg n)$ algorithm for constructing medial-axis transforms in curved regions was given by Kirkpatrick [Kir79].

Straight skeletons were introduced in [AAAG95], with a subquadratic algorithm due to [EE99]. See [LD03] for an interesting application of straight skeletons to defining the roof structures in virtual building models.

Related Problems: Voronoi diagrams (see page 576), Minkowski sum (see page 617).



INPUT OUTPUT

17.11 Polygon Partitioning

Input description: A polygon or polyhedron P.

Problem description: Partition P into a small number of simple (typically convex) pieces.

Discussion: Polygon partitioning is an important preprocessing step for many geometric algorithms, because geometric problems tend to be simpler on convex objects than on nonconvex ones. It is often easier to work with a small number of convex pieces than with a single nonconvex polygon.

Several flavors of polygon partitioning arise, depending upon the particular application:

- *Should all the pieces be triangles?* – Triangulation is the mother of all polygon partitioning problems, where we partition the interior of the polygon completely into triangles. Triangles are convex and have only three sides, making them the most elementary possible polygon.

  All triangulations of an n-vertex polygon contain exactly $n - 2$ triangles. Therefore, triangulation cannot be the answer if we seek a small number of convex pieces. A "nice" triangulation is judged by the shape of the triangles, not the count. See Section 17.3 (page 572) for a thorough discussion of triangulation.

- *Do I want to cover or partition my polygon?* – *Partitioning* a polygon means completely dividing the interior into nonoverlapping pieces. *Covering* a polygon means that our decomposition is permitted to contain mutually overlapping pieces. Both can be useful in different situations. In decomposing a complicated query polygon in preparation for a range search (Section 17.6 (page 584)), we seek a partitioning, so that each point we locate occurs in exactly one piece. In decomposing a polygon for painting purposes, a covering suffices, since there is no difficulty with filling in a region twice. We will concentrate here on partitioning, since it is simpler to do right, and any application needing a covering will accept a partitioning. The only drawback is that partitions can be larger than coverings.

- *Am I allowed to add extra vertices?* – A final issue is whether we are allowed to add Steiner vertices to the polygon, either by splitting edges or adding interior points. Otherwise, we are restricted to adding chords between two existing vertices. The former may result in a smaller number of pieces, at the cost of more complicated algorithms and perhaps messier results.

The Hertel-Mehlhorn heuristic for convex decomposition using diagonals is simple and efficient. It starts with an arbitrary triangulation of the polygon and then deletes any chord that leaves only convex pieces. A chord deletion creates a nonconvex piece only if it creates an internal angle that is greater than 180 degrees. The decision of whether such an angle will be created can be made locally from the chords and edges surrounding the chord, in constant time. The result always contains no more than four times the minimum number of convex pieces.

I recommend using this heuristic unless it is critical for you to minimize the number of pieces. By experimenting with different triangulations and various deletion orders, you may be able to obtain somewhat better decompositions.

Dynamic programming may be used to find the absolute minimum number of diagonals used in the decomposition. The simplest implementation, which maintains the number of pieces for all $O(n^2)$ subpolygons split by a chord, runs in $O(n^4)$. Faster algorithms use fancier data structures, running in $O(n+r^2 \min(r^2, n))$ time, where r is the number of reflex vertices. An $O(n^3)$ algorithm that further reduces the number of pieces by adding interior vertices is cited below, although it is complex and presumably difficult to implement.

An alternate decomposition problem partitions polygons into *monotone* pieces. The vertices of a y-monotone polygon can be divided into two chains such that any horizontal line intersects either chain at most once.

Implementations: Many triangulation codes start by finding a trapezoidal or monotone decomposition of polygons. Further, a triangulation is a simple form of convex decomposition. Check out the codes in Section 17.3 (page 572) as a starting point.

CGAL (*www.cgal.org*) contains a polygon-partitioning library that includes (1) the Hertel-Mehlhorn heuristic for partitioning a polygon into convex pieces,

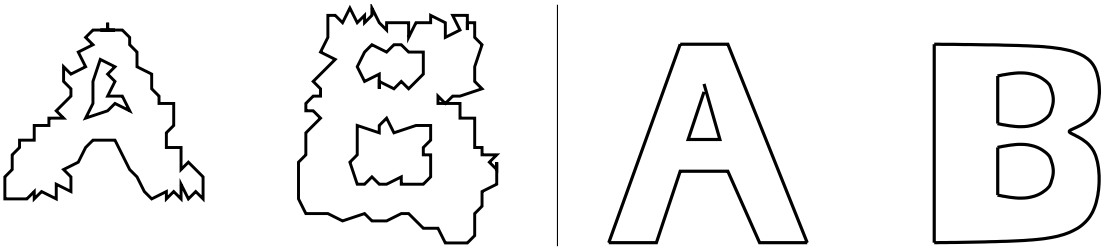
(2) finding a optimal convex partitioning using the $O(n^4)$ dynamic programming algorithm, and (3) an $O(n \log n)$ sweepline heuristic for partitioning into monotone polygons.

A triangulation code of particular relevance here is GEOMPACK—a suite of Fortran 77 codes by Barry Joe for 2- and 3-dimensional triangulation and convex decomposition problems. In particular, it does both Delaunay triangulation and convex decompositions of polygonal and polyhedral regions, as well as arbitrary-dimensional Delaunay triangulations.

Notes: Recent survey articles on polygon partitioning include [Kei00, OS04]. Keil and Sack [KS85] give an excellent survey on what is known about partitioning and covering polygons. Expositions on the Hertel-Mehlhorn heuristic [HM83] include [O'R01]. The $O(n+r^2 \min(r^2, n))$ dynamic programming algorithm for minimum convex decomposition using diagonals is due to Keil and Snoeyink [KS02]. The $O(r^3 + n)$ algorithm minimizing the number of convex pieces with Steiner points appears in [CD85]. Lien and Amato [LA06] provide an efficient heuristic for decomposing polygons with holes into "almost convex" polygons in $O(nr)$ time, with later work generalizing this to polyhedra.

Art gallery problems are an interesting topic related to polygon covering, where we seek to position the minimum number of guards in a given polygon such that every point in the interior of the polygon is watched by at least one guard. This corresponds to covering the polygon with a minimum number of star-shaped polygons. O'Rourke [O'R87] is a beautiful (although sadly out of print) book that presents the art gallery problem and its many variations.

Related Problems: Triangulation (see page 572), set cover (see page 621).



INPUT OUTPUT

17.12 Simplifying Polygons

Input description: A polygon or polyhedron p, with n vertices.

Problem description: Find a polygon or polyhedron p' containing only n' vertices, such that the shape of p' is as close as possible to p.

Discussion: Polygon simplification has two primary applications. The first involves cleaning up a noisy representation of a shape, perhaps obtained by scanning a picture of an object. Simplifying it can remove the noise and reconstruct the original object. The second involves data compression, where we seek to reduce detail on a large and complicated object yet leave it looking essentially the same. This can be a big win in computer graphics, where the smaller model might be significantly faster to render.

Several issues arise in shape simplification:

- *Do you want the convex hull?* – The simplest simplification is the convex hull of the object's vertices (see Section 17.2 (page 568)). The convex hull removes all internal concavities from the polygon. If you were simplifying a robot model for motion planning, this is almost certainly a good thing. But using the convex hull in an OCR system would be disastrous, because the concavities of characters provide most of the interesting features. An 'X' would be identical to an 'I', since both hulls are boxes. Another problem is that taking the convex hull of a convex polygon does nothing to simplify it further.

- *Am I allowed to insert or just delete points?* – The typical goal of simplification is to to represent the object as well as possible using a given number of vertices. The simplest approaches do local modifications to the boundary in order to reduce the vertex count. For example, if three consecutive vertices form a small-area triangle or define an extremely large angle, the center

vertex can be deleted and replaced with an edge without severely distorting the polygon.

Methods that only delete vertices quickly melt the shape into unrecognizability, however. More robust heuristics move vertices around to cover up the gaps that are created by deletions. Such "split-and-merge" heuristics can do a decent job, although nothing is guaranteed. Better results are likely by using the Douglas-Peucker algorithm, described next.

- *Must the resulting polygon be intersection-free?* – A serious drawback of incremental procedures is that they fail to ensure *simple* polygons, meaning they are without self-intersections. Thus "simplified" polygons may have ugly artifacts that may cause problems for subsequent routines. If simplicity is important, you should test all the line segments of your polygon for any pairwise intersections, as discussed in Section 17.8 (page 591).

  A polygon simplification approach that guarantees a simple approximation involves computing minimum-link paths. The *link distance* of a path between points s and t is the number of straight segments on the path. An as-the-crow-flies path has a link distance of one, while in general the link distance is one more than the number of turns on the path. The link distance between points s and t in a scene with obstacles is defined by the minimum link distance over all paths from s to t.

  The link distance approach "fattens" the boundary of the polygon by some acceptable error window ϵ (see Section 17.16 (page 617)) in order to construct a channel around the polygon. The minimum-link cycle in this channel represents the simplest polygon that never deviates from the original boundary by more than ϵ. An easy-to-compute approximation to link distance reduces it to breadth-first search, by placing a discrete set of possible turn points within the channel and connecting each pair of mutually visible points by an edge.

- *Are you given an image to clean up instead of a polygon to simplify?* – The conventional approach to cleaning up noise from a digital image is to take the Fourier transform of the image, filter out the high-frequency elements, and then take the inverse transform to recreate the image. See Section 13.11 (page 431) for details on the fast Fourier transform.

The Douglas-Puecker algorithm for shape simplification starts with a simple approximation and then refines it, instead of starting with a complicated polygon and trying to simplify it. Start by selecting two vertices v_1 and v_2 of polygon P, and propose the degenerate polygon v_1, v_2, v_1 as a simple approximation P'. Scan through each of the vertices of P, and select the one that is farthest from the corresponding edge of the polygon P'. Inserting this vertex adds the triangle to P' to minimize the maximum deviation from P. Points can be inserted until satisfactory results are achieved. This takes $O(kn)$ to insert k points when $|P| = n$.

Simplification becomes considerably more difficult in three dimensions. Indeed, it is NP-complete to find the minimum-size surface separating two polyhedra. Higher-dimensional analogies of the planar algorithms discussed here can be used to heuristically simplify polyhedra. See the Notes section.

Implementations: The Douglas-Peucker algorithm is readily implemented. Snoeyink provides a C implementation of his algorithm with efficient worst-case performance [HS94] at *http://www.cs.unc.edu/∼snoeyink/papers/DPsimp.arch*.

Simplification envelopes are a technique for automatically generating level-of-detail hierarchies for polygonal models. The user specifies a single error tolerance, and the maximum surface deviation of the simplified model from the original model, and a new, simplified model is generated. An implementation is available from *http://www.cs.unc.edu/∼geom/envelope.html*. This code preserves holes and prevents self-intersection.

QSlim is a quadric-based simplification algorithm that can produce high quality approximations of triangulated surfaces quite rapidly. It is available at *http://graphics.cs.uiuc.edu/∼garland/software.html*.

Yet another approach to polygonal simplification is based on simplifying and expanding the medial-axis transform of the polygon. The medial-axis transform (see Section 17.10 (page 598)) produces a skeleton of the polygon, which can be trimmed before inverting the transform to yield a simpler polygon. *Cocone* (*http://www.cse.ohio-state.edu/∼tamaldey/cocone.html*) constructs an approximate medial-axis transform of the polyhedral surface it interpolates from points in E^3. See [Dey06] for the theory behind *Cocone*. *Powercrust* [ACK01a, ACK01b] constructs a discrete approximation to the medial-axis transform, and then reconstructs the surface from this transform. When the point samples are sufficiently dense, the algorithm is guaranteed to produce a geometrically and topologically correct approximation to the surface. It is available at *http://www.cs.utexas.edu/users/amenta/powercrust/*.

CGAL (*www.cgal.org*) provides support for the most extreme polygon/polyhedral simplification, finding the smallest enclosing circle/sphere.

Notes: The Douglas-Peucker incremental refinement algorithm [DP73] is the basis for most shape simplification schemes, with faster implementations due to [HS94, HS98]. The link distance approach to polygon simplification is presented in [GHMS93]. Shape simplification problems become considerably more complex in three dimensions. Even finding the minimum-vertex convex polyhedron lying between two nested convex polyhedra is NP-complete [DJ92], although approximation algorithms are known [MS95b].

Heckbert and Garland [HG97] survey algorithms for shape simplification. Shape simplification using medial-axis transformations (see 17.10) are presented in [TH03].

Testing whether a polygon is simple can be performed in linear time, at least in theory, as a consequence of Chazelle's linear-time triangulation algorithm [Cha91].

Related Problems: Fourier transform (see page 431), convex hull (see page 568).

INPUT OUTPUT

17.13 Shape Similarity

Input description: Two polygonal shapes, P_1 and P_2.

Problem description: How similar are P_1 and P_2?

Discussion: Shape similarity is a problem that underlies much of pattern recognition. Consider a system for optical character recognition (OCR). We are given a library of shape models representing letters, and unknown shapes obtained by scanning a page. We seek to identify the unknown shapes by matching them to the most similar shape models.

Shape similarity is an inherently ill-defined problem, because what "similar" means is application dependent. Thus, no single algorithmic approach can solve all shape-matching problems. Whichever method you select, expect to spend a large chunk of time tweaking it to achieve maximum performance.

Among your possible approaches are

- *Hamming distance* – Assume that your two polygons have been overlaid one on top of the other. *Hamming distance* measures the area of symmetric difference between the two polygons—in other words, the area lying within one of the two polygons but not both of them. When two polygons are identical and properly aligned, the Hamming distance is zero. If the polygons differ only in a little noise at the boundary, then the Hamming distance of properly aligned polygons will be small.

  Computing the area of the symmetric difference reduces to finding the intersection or union of two polygons (discussed in Section 17.8 (page 591)) and then computing areas (discussed in Section 17.1). But the difficult part of

computing Hamming distance is finding the right alignment of the two polygons. This overlay problem is simplified in applications such as OCR because the characters are inherently aligned within lines on the page and are not free to rotate. Efficient algorithms for optimizing the overlap of convex polygons without rotation are cited below. Simple but reasonably effective heuristics are based on identifying reference landmarks on each polygon (such as the centroid, bounding box, or extremal vertices) and then matching a subset of these landmarks to define the alignment.

Hamming distance is particularly simple and efficient to compute on bit-mapped images, since after alignment all we do is sum the differences of the corresponding pixels. Although Hamming distance makes sense conceptually and can be simple to implement, it captures only a crude notion of shape and is likely to be ineffective in most applications.

- *Hausdorff distance* – An alternative distance metric is *Hausdorff distance*, which identifies the point on P_1 that is the maximum distance from P_2 and returns this distance. The Hausdorff distance is not symmetrical, for the tip of a long but thin protrusion from P_1 can imply a large Hausdorff distance P_1 to P_2, even though every point on P_2 is close to some point on P_1. A fattening of the entire boundary of one of the models (as is liable to happen with boundary noise) by a small amount may substantially increase the Hamming distance yet have little effect on the Hausdorff distance.

  Which is better, Hamming or Hausdorff? It depends upon your application. As with Hamming distance, computing the right alignment between the polygons can be difficult and time-consuming.

- *Comparing Skeletons* – A more powerful approach to shape similarity uses thinning (see Section 17.10 (page 598)) to extract a tree-like skeleton for each object. This skeleton captures many aspects of the original shape. The problem now reduces to comparing the shape of two such skeletons, using such features as the topology of the tree and the lengths/slopes of the edges. This comparison can be modeled as a form of subgraph isomorphism (see Section 16.9 (page 550)), with edges allowed to match whenever their lengths and slopes are sufficiently similar.

- *Support Vector Machines* – A final approach for pattern recognition/matching problems uses a learning-based technique such as neural networks or the more powerful *support vector machines*. These prove a reasonable approach to recognition problems when you have a lot of data to experiment with and no particular ideas of what to do with it. First, you must identify a set of easily computed features of the shape, such as area, number of sides, and number of holes. Based on these features, a black-box program (the support vector machine training algorithm) takes your training data and produces a classification function. This classification function accepts as input the values

of these features and returns a measure of what the shape is, or how close it is to a particular shape.

How good are the resulting classifiers? It depends upon the application. Like any ad hoc method, SVMs usually take a fair amount of tweaking and tuning to realize their full potential.

There is one caveat. If you don't know how/why black-box classifiers are making their decisions, you can't know when they will fail. An interesting case was a system built for the military to distinguish between images of cars and tanks. It performed very well on test images but disastrously in the field. Eventually, someone realized that the car images had been filmed on a sunnier day than the tanks, and the program was classifying solely on the presence of clouds in the background of the image!

Implementations: A Hausdorff-based image comparison implementation in C is available at *http://www.cs.cornell.edu/vision/hausdorff/hausmatch.html*. An alternate distance metric between polygons can be based on its angle-turning function [ACH+91]. An implementation in C of this turning function metric by Eugene K. Ressler is provided at *http://www.cs.sunysb.edu/∼algorith*.

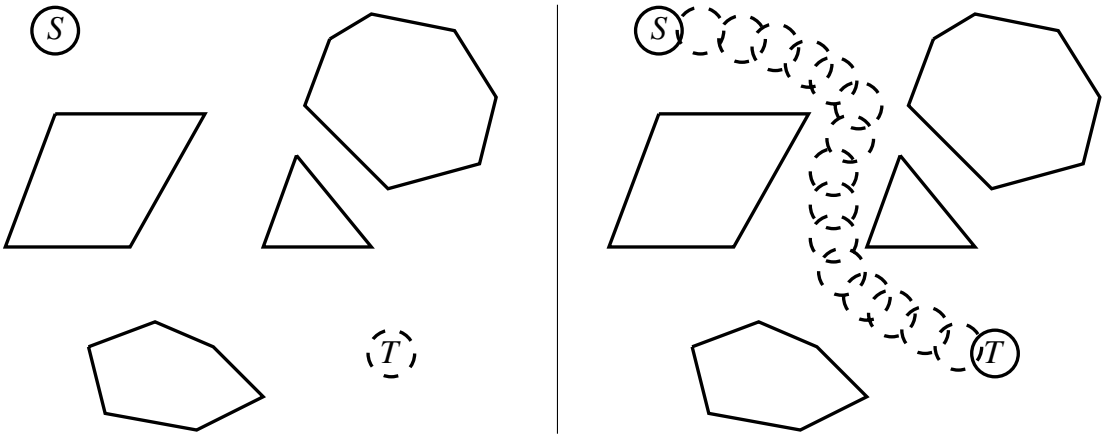
Several excellent support vector machine classifiers are available. These include the kernal-machine library (*http://www.terborg.net/research/kml/*), SVM^{light} (*http://svmlight.joachims.org/*) and the widely used and well-supported LIBSVM (*http://www.csie.ntu.edu.tw/∼cjlin/libsvm/*).

Notes: General books on pattern classification algorithms include [DHS00, JD88]. A wide variety of computational geometry approaches to shape similarity testing have been proposed, including [AMWW88, ACH+91, Ata84, AE83, BM89, OW85]. See the survey by Alt and Guibas [AG00].

The optimal alignment of n and m-vertex convex polygons subject to translation (but not rotation) can be computed in $O((n+m)\log(n+m))$ time [dBDK+98]. An approximation of the optimal overlap under translation and rotation is due to Ahn, et al. [ACP+07].

A linear-time algorithm for computing the Hausdorff distance between two convex polygons is given in [Ata83], with algorithms for the general case reported in [HK90].

Related Problems: Graph isomorphism (see page 550), thinning (see page 598).



INPUT OUTPUT

17.14 Motion Planning

Input description: A polygonal-shaped robot starting in a given position s in a room containing polygonal obstacles, and a goal position t.

Problem description: Find the shortest route taking s to t without intersecting any obstacles.

Discussion: That motion planning is a complex problem is obvious to anyone who has tried to move a large piece of furniture into a small apartment. It also arises in systems for molecular docking. Many drugs are small molecules that act by binding to a given target model. Identifying which binding sites are accessible to a candidate drug is clearly an instance of motion planning. Plotting paths for mobile robots is another canonical motion-planning application.

Finally, motion planning provides a tool for computer animation. Given the set of object models and where they appear in scenes s_1 and s_2, a motion planning algorithm can construct a short sequence of intermediate motions to transform s_1 to s_2. These motions can serve to fill in the intermediate scenes between s_1 and s_2, with such scene interpolation greatly reducing the workload on the animator.

Many factors govern the complexity of motion planning problems:

- *Is your robot a point?* – For point robots, motion planning becomes finding the shortest path from s to t around the obstacles. This is also known as geometric shortest path. The most readily implementable approach constructs the *visibility graph* of the polygonal obstacles, plus the points s and t. This

visibility graph has a vertex for each obstacle vertex and an edge between two obstacle vertices iff they "see" each other without being blocked by some obstacle edge.

The visibility graph can be constructed by testing each of the $\binom{n}{2}$ vertex-pair edge candidates for intersection against each of the n obstacle edges, although faster algorithms are known. Assign each edge of this visibility graph with weight equal to its length. Then the shortest path from s to t can be found using Dijkstra's shortest-path algorithm (see Section 15.4 (page 489)) in time bounded by the time required to construct the visibility graph.

- *What motions can your robot perform?* – Motion planning becomes considerably more difficult when the robot becomes a polygon instead of a point. Now all of the corridors that we use must be wide enough to permit the robot to pass through.

  The algorithmic complexity depends upon the number of *degrees of freedom* that the robot can use to move. Is it free to rotate as well as to translate? Does the robot have links that are free to bend or to rotate independently, as in an arm with a hand? Each degree of freedom corresponds to a dimension in the search space of possible configurations. Additional freedom makes it more likely that a short path exists from start to goal, although it also becomes harder to find this path.

- *Can you simplify the shape of your robot?* – Motion planning algorithms tend to be complex and time-consuming. Anything you can do to simplify your environment is a win. In particular, consider replacing your robot in an enclosing disk. If there is a start-to-goal path for this disk, it defines such a path for the robot inside of it. Furthermore, any orientation of a disk is equivalent to any other orientation, so rotation provides no help in finding a path. All movements can thus be limited to the simpler case of translation.

- *Are motions limited to translation only?* – When rotation is not allowed, the *expanded obstacles* approach can be used to reduce the problem of polygonal motion planning to the previously-resolved case of a point robot. Pick a reference point on the robot, and replace each obstacle by its Minkowski sum with the robot polygon (see Section 17.16 (page 617)). This creates a larger, fatter obstacle, defined by the shadow traced as the robot walks a loop around the object while maintaining contact with it. Finding a path from the initial reference position to the goal amidst these fattened obstacles defines a legal path for the polygonal robot in the original environment.

- *Are the obstacles known in advance?* – We have assumed that the robot starts out with a map of its environment. But this can't be true, say, in applications where the obstacles move. There are two approaches to solving motion-planning problems without a map. The first approach explores the

environment, building a map of what has been seen, and then uses this map to plan a path to the goal. A simpler strategy proceeds like a sightless man with a compass. Walk in the direction towards the goal until progress is blocked by an obstacle, and then trace a path along the obstacle until the robot is again free to proceed directly towards the goal. Unfortunately, this will fail in environments of sufficient complexity.

The most practical approach to general motion planning involves randomly sampling the *configuration space* of the robot. The configuration space defines the set of legal positions for the robot using one dimension for each degree of freedom. A planar robot capable of translation and rotation has three degrees of freedom, namely the x- and y-coordinates of a reference point on the robot and the angle θ relative to this point. Certain points in this space represent legal positions, while others intersect obstacles.

Construct a set of legal configuration-space points by random sampling. For each pair of points p_1 and p_2, decide whether there exists a direct, nonintersecting path between them. This defines a graph with vertices for each legal point and edges for each such traversable pair. Motion planning now reduces to finding a direct path from the initial/final position to some vertex in the graph, and then solving a shortest-path problem between the two vertices.

There are many ways to enhance this basic technique, such as adding additional vertices to regions of particular interest. Building such a road map provides a nice, clean approach to solving problems that would otherwise get very messy.

Implementations: The *Motion Planning Toolkit* (MPK) is a C++ library and toolkit for developing single- and multi-robot motion planners. It includes SBL, a fast single-query probabilistic roadmap path planner, and is available at *http://robotics.stanford.edu/∼mitul/mpk/*.

The University of North Carolina GAMMA group has produced several efficient collision detection libraries (not really motion planning) of which *SWIFT++* [EL01] is the most recent member of this family. It can detect intersection, compute approximate/exact distances between objects, and determine object-pair contacts in scenes composed of rigid polyhedral models. See *http://www.cs.unc.edu/∼geom/collide/* for pointers to all of these libraries.

The computational geometry library CGAL (*www.cgal.org*) contains many algorithms related to motion planning including visibility graph construction and Minkowski sums. O'Rourke [O'R01] gives a toy implementation of an algorithm to plot motion for a two-jointed robot arm in the plane. See Section 19.1.10 (page 662).

Notes: Latombe's book [Lat91] describes practical approaches to motion planning, including the random sampling method described above. Two other worthy books on motion planning are available freely on line, by LaValle [LaV06] (*http://planning.cs.uiuc.edu/*) and Laumond [Lau98] (*http://www.laas.fr/∼jpl/book.html*).

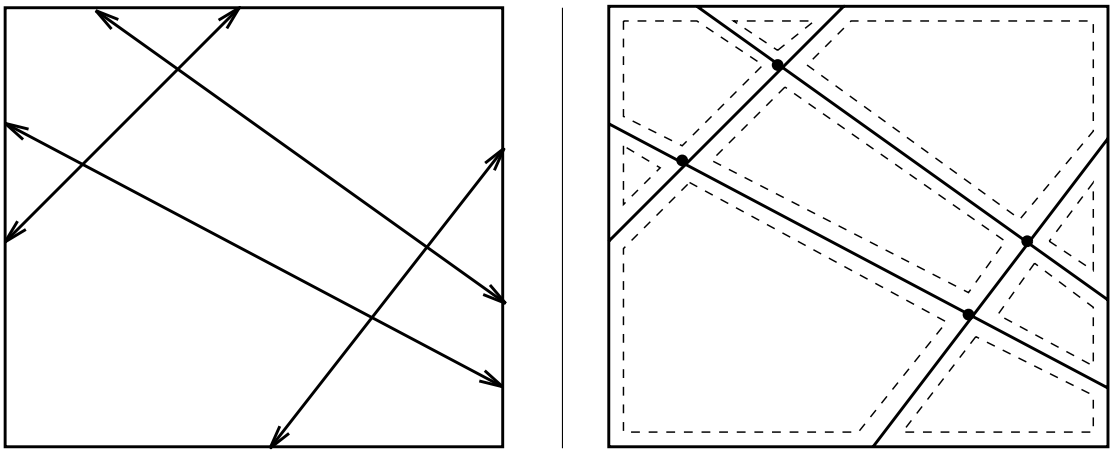
Motion planning was originally studied by Schwartz and Sharir as the "piano mover's problem." Their solution constructs the complete free space of robot positions that do not intersect obstacles, and then finds the shortest path within the proper connected component. These free space descriptions are very complicated, involving arrangements of higher-degree algebraic surfaces. The fundamental papers on the piano mover's problem appear in [HSS87], with [Sha04] a survey of current results.

The best general result for this free-space approach to motion planning is due to Canny [Can87], who showed that any problem with d degrees of freedom can be solved in $O(n^d \lg n)$, although faster algorithms exist for special cases of the general motion-planning problem. The expanded obstacle approach to motion planning is due to Lozano-Perez and Wesley [LPW79]. The heuristic, sightless man's approach to motion planning discussed previously has been studied by Lumelski [LS87].

The time complexity of algorithms based on the free-space approach to motion planning depends intimately on the combinatorial complexity of the arrangement of surfaces defining the free space. Algorithms for maintaining arrangements are presented in Section 17.15 (page 614). Davenport-Schinzel sequences often arise in the analysis of such arrangements. Sharir and Agarwal [SA95] provide a comprehensive treatment of Davenport-Schinzel sequences and their relevance to motion planning.

The visibility graph of n line segments with E pairs of visible vertices can be constructed in $O(n \lg n + E)$ time [GM91, PV96], which is optimal. Hershberger and Suri [HS99] have an $O(n \lg n)$ algorithm for finding shortest paths for point-robots with polygonal obstacles. Chew [Che85] provides an $O(n^2 \lg n)$ for finding shortest paths for a disk-robot in such a scene.

Related Problems: Shortest path (see page 489), Minkowski sum (see page 617).



INPUT OUTPUT

17.15 Maintaining Line Arrangements

Input description: A set of lines and line segments $l_1, \ldots, l_n$.

Problem description: What is the decomposition of the plane defined by $l_1, \ldots, l_n$?

Discussion: A fundamental problem in computational geometry is explicitly constructing the regions formed by the intersections of a set of n lines. Many problems reduce to constructing and analyzing such an arrangement of a specific set of lines. Examples include:

- *Degeneracy testing* – Given a set of n lines in the plane, do any three of them pass through the same point? Brute-force testing of all triples takes $O(n^3)$ time. Instead, we can construct the arrangement of the lines and then walk over each vertex and explicitly count its degree, all in quadratic time.
- *Satisfying the maximum number of linear constraints* – Suppose that we are given a set of n linear constraints, each of the form $y \leq a_i x + b_i$. Which point in the plane satisfies the largest number of them? Construct the arrangement of the lines. All points in any region or *cell* of this arrangement satisfy exactly the same set of constraints, so we need to test only one point per cell to find the global maximum.

Thinking of geometric problems in terms of features in an arrangement can be very useful in formulating algorithms. Unfortunately, it must be admitted that

arrangements are not as popular in practice as might be supposed. Primarily, this is because some depth of understanding is required to apply them correctly. The computational geometry library CGAL now provides a general and robust enough implementation to justify the effort to do so. Issues arising in arrangements include

- *What is the right way to construct a line arrangement?* – Algorithms for constructing arrangements are incremental. Begin with an arrangement of one or two lines. Subsequent lines are inserted into the arrangement one at a time, yielding larger and larger arrangements. To insert a new line, we start on the leftmost cell containing the line and walk over the arrangement to the right, moving from cell to neighboring cell and splitting into two pieces those cells that contain the new line.

- *How big will your arrangement be?* – A geometric fact called the *zone theorem* implies that the kth line inserted cuts through k cells of the arrangement, and further that $O(k)$ total edges form the boundary of these cells. This means that we can scan through each edge of every cell we encounter on our insertion walk, confident that only linear total work will be performed while inserting the line into the arrangement. Therefore, the total time to insert all n lines in constructing the full arrangement is $O(n^2)$.

- *What do you want to do with your arrangement?* – Given an arrangement and a query point, we are often interested in identifying which cell of the arrangement contains the point. This is the problem of point location, discussed in Section 17.7 (page 587). Given an arrangement of lines or line segments, we are often interested in computing all points of intersection of the lines. The problem of intersection detection is discussed in Section 17.8 (page 591).

- *Does your input consist of points instead of lines?* – Although lines and points seem to be different geometric objects, appearances can be misleading. Through the use of *duality transformations*, we can turn line L into point p and vice versa:

  $$L : y = 2ax - b \leftrightarrow p : (a, b)$$

  Duality is important because we can now apply line arrangements to point problems, often with surprising results.

  For example, suppose we are given a set of n points, and we want to know whether any three of them all lie on the same line. This sounds similar to the degeneracy testing problem discussed above. In fact it is *exactly the same*, with only the role of points and lines exchanged. We can dualize our points into lines as above, construct the arrangement, and then search for a vertex with three lines passing through it. The dual of this vertex defines the line on which the three initial vertices lie.

It often becomes useful to traverse each face of an existing arrangement exactly once. Such traversals are called *sweepline algorithms*, and are discussed in some detail in Section 17.8 (page 591). The basic procedure sorts the intersection points by x-coordinate and then walks from left to right while keeping track of all we have seen.

Implementations: CGAL (*www.cgal.org*) provides a generic and robust package for arrangements of curves (not just lines) in the plane. This should be the starting point for any serious project using arrangements.

A robust code for constructing and topologically sweeping an arrangement in C++ is provided at *http://www.cs.tufts.edu/research/geometry/other/sweep/*. An extension of topological sweep to deal with the visibility complex of a collection of pairwise disjoint convex planar sets has been provided in CGAL.

Arrange is a package for maintaining arrangements of polygons in either the plane or on the sphere. Polygons may be degenerate, and hence represent arrangements of lines. A randomized incremental construction algorithm is used, and efficient point location on the arrangement is supported. *Arrange* is written in C by Michael Goldwasser and is available from *http://euler.slu.edu/∼goldwasser/publications/*.

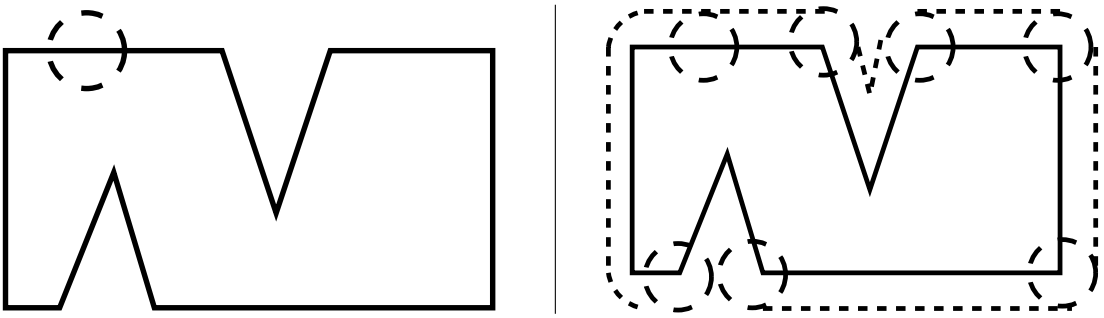
Notes: Edelsbrunner [Ede87] provides a comprehensive treatment of the combinatorial theory of arrangements, plus algorithms on arrangements with applications. It is an essential reference for anyone seriously interested in the subject. Recent surveys of combinatorial and algorithmic results include [AS00, Hal04]. Good expositions on constructing arrangements include [dBvKOS00, O'R01]. Implementation issues related to arrangements as implemented in CGAL are discussed in [FWH04, HH00].

Arrangements generalize naturally beyond two dimensions. Instead of lines, the space decomposition is defined by planes (or beyond 3-dimensions, *hyperplanes*). The zone theorem states that any arrangement of n d-dimensional hyperplanes has total complexity $O(n^d)$, and any single hyperplane intersects cells of complexity $O(n^{d-1})$. This provides the justification for the incremental construction algorithm for arrangements. Walking around the boundary of each cell to find the next cell that the hyperplane intersects takes time proportional to the number of cells created by inserting the hyperplane.

The history of the zone theorem has become somewhat muddled, because the original proofs were later found to be in error in higher dimensions. See [ESS93] for a discussion and a correct proof. The theory of Davenport-Schinzel sequences is intimately tied into the study of arrangements, which is presented in [SA95].

The naive algorithm for sweeping an arrangement of lines sorts the n^2 intersection points by x-coordinate and hence requires $O(n^2 \lg n)$ time. The *topological sweep* [EG89, EG91] eliminates the need to sort, and so traverses the arrangement in quadratic time. This algorithm is readily implementable and can be applied to speed up many sweepline algorithms. See [RSS02] for a robust implementation with experimental results.

Related Problems: Intersection detection (see page 591), point location (see page 587).



INPUT OUTPUT

17.16 Minkowski Sum

Input description: Point sets or polygons A and B, containing n and m vertices respectively.

Problem description: What is the convolution of A and B—i.e. , the Minkowski sum $A + B = \{x + y | x \in A, y \in B\}$?

Discussion: Minkowski sums are useful geometric operations that can *fatten* objects in appropriate ways. For example, a popular approach to motion planning for polygonal robots in a room with polygonal obstacles (see Section 17.14 (page 610)) fattens each of the obstacles by taking the Minkowski sum of them with the shape of the robot. This reduces the problem to the (more easily solved) case of point robots. Another application is in shape simplification (see Section 17.12 (page 604)). Here we fatten the boundary of an object to create a channel around it, and then let the minimum link path lying within this channel define the simplified shape. Finally, convolving an irregular object with a small circle will help smooth out the boundaries by eliminating minor nicks and cuts.

The definition of a Minkowski sum assumes that the polygons A and B have been positioned on a coordinate system:

$$A + B = \{x + y \mid x \in A, y \in B\}$$

where $x+y$ is the vector sum of two points. Thinking of this in terms of translation, the Minkowski sum is the union of all translations of A by a point defined within B. Issues arising in computing Minkowski sums include

- *Are your objects rasterized images or explicit polygons?* – The definition of Minkowski summation suggests a simple algorithm if A and B are rasterized images. Initialize a sufficiently large matrix of pixels by determining the size

of the convolution of the bounding boxes of A and B. For each pair of points in A and B, sum up their coordinates and darken the appropriate pixel. These algorithms get more complicated if an explicit polygonal representation of the Minkowski sum is needed.

- *Do you want to fatten your object by a fixed amount?* – The most common fattening operation expands a model M by a given tolerance t, known as *offsetting*. As shown in the figures above, this is accomplished by computing the Minkowski sum of M with a disk of radius t. The basic algorithms still work, although the offset is not a polygon. Its boundary is instead composed of circular arcs and line segments.

- *Are your objects convex or non-convex?* – The complexity of computing Minkowski sums depends in a serious way on the shape of the polygons. If both A and B are convex, the Minkowski sum can be found in $O(n+m)$ time by tracing the boundary of one polygon with another. If one of them is nonconvex, the *size* of the sum alone can be as large as $\Theta(nm)$. Even worse is when both A and B are nonconvex, in which case the *size* of the sum can be as large as $\Theta(n^2m^2)$. Minkowski sums of nonconvex polygons are often ugly in a majestic sort of way, with holes either created or destroyed in surprising fashion.

A straightforward approach to computing the Minkowski sum is based on triangulation and union. First, triangulate both polygons, then compute the Minkowski sum of each triangle of A against each triangle of B. The sum of a triangle against another triangle is easy to compute and is a special case of convex polygons, discussed below. The union of these $O(nm)$ convex polygons will be $A+B$. Algorithms for computing the union of polygons are based on plane sweep, as discussed in Section 17.8 (page 591).

Computing the Minkowski sum of two convex polygons is easier than the general case, because the sum will always be convex. For convex polygons it is easiest to slide A along the boundary of B and compute the sum edge by edge. Partitioning each polygon into a small number of convex pieces (see Section 17.11 (page 601)), and then unioning the Minkowski sum for each pair of pieces, will usually be much more efficient than working with two fully triangulated polygons.

Implementations: The CGAL (*www.cgal.org*) Minkowski sum package provides an efficient and robust code to find the Minkowski sums of two arbitrary polygons, as well as compute both exact and approximate offsets.

An implementation for computing the Minkowski sums of two convex polyhedra in $3D$ is described in [FH06] and available at *http://www.cs.tau.ac.il/∼efif/CD/*.

Notes: Good expositions on algorithms for Minkowski sums include [dBvKOS00, O'R01]. The fastest algorithms for various cases of Minkowski sums include [KOS91, Sha87].

The practical efficiency of Minkowski sum in the general case depends upon how the polygons are decomposed into convex pieces. The optimal solution is not necessarily the

partition into the fewest number of convex pieces. Agarwal et al. [AFH02] give a through study of decomposition methods for Minkowski sum.

The combinatorial complexity of the Minkowski sum of two convex polyhedra in three dimensions is completely resolved in [FHW07]. An implementation of Minkowski sum for such polyhedra is described in [FH06].

Kedem and Sharir [KS90] present an efficient algorithm for translational motion planning for polygonal robots, based on Minkowski sums.

Related Problems: Thinning (see page 598), motion planning (see page 610), simplifying polygons (see page 604).

18

Set and String Problems

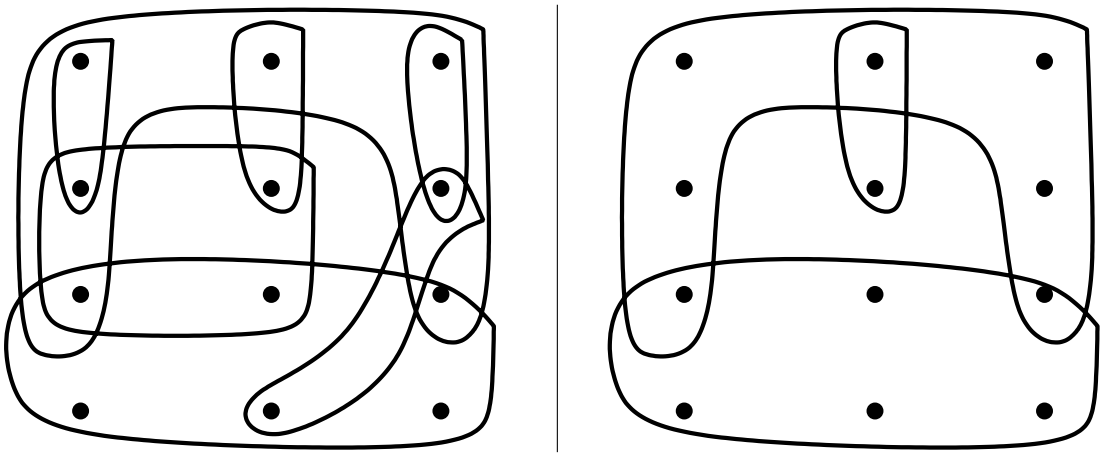
Sets and strings both represent collections of objects—the difference is whether order matters. Sets are collections of symbols whose order is assumed to carry no significance, while strings are defined by the sequence or arrangement of symbols.

The assumption of a fixed order makes it possible to solve string problems much more efficiently than set problems, through techniques such as dynamic programming and advanced data structures like suffix trees. The interest in and importance of string-processing algorithms have been increasing due to bioinformatics, Web searches, and other text-processing applications. Recent books on string algorithms include:

- *Gusfield* [Gus97] – To my taste, this remains is the best introduction to string algorithms. It contains a thorough discussion on suffix trees, with clear and innovative formulations of classical exact string-matching algorithms.
- *Crochemore, Hancart, and Lecroq* [CHL07] – A comprehensive treatment of string algorithms, written by a true leader in the field. Translated from the French, but clear and accessible.
- *Navarro and Raffinot* [NR07] – A concise but practical and implementation-oriented treatment of pattern-matching algorithms, with particularly thorough treatment of bit-parallel approaches.
- *Crochemore and Rytter* [CR03] – A survey of specialized topics in string algorithmics emphasizing theory.

Theoreticians working in string algorithmics sometimes refer to their field as *Stringology*. The annual *Combinatorial Pattern Matching* (CPM) conference is the primary venue devoted to both practical and theoretical aspects of string algorithmics and related areas.

S.S. Skiena, *The Algorithm Design Manual*, 2nd ed., DOI: 10.1007/978-1-84800-070-4_18,



INPUT OUTPUT

18.1 Set Cover

Input description: A collection of subsets $S = \{S_1, \ldots, S_m\}$ of the universal set $U = \{1, \ldots, n\}$.

Problem description: What is the smallest subset T of S whose union equals the universal set—i.e. , $\cup_{i=1}^{|T|} T_i = U$?

Discussion: Set cover arises when you try to efficiently acquire items that have been packaged in a fixed set of lots. You seek a collection of at least one of each distinct type of item, while buying as few lots as possible. Finding *a* set cover is easy, because you can always buy one of each possible lot. However, identifying a small set cover let you do the same job for less money. Set cover provided a natural formulation of the Lotto ticket optimization problem discussed in Section 1.6 (page 23). There we seek to buy the smallest number of tickets needed to cover all of a given set of combinations.

Boolean logic minimization is another interesting application of set cover. We are given a particular Boolean function of k variables, which describes whether the desired output is 0 or 1 for each of the 2^k possible input vectors. We seek the simplest circuit that exactly implements this function. One approach is to find a disjunctive normal form (DNF) formula on the variables and their complements, such as $x_1\bar{x}_2 + \bar{x}_1\bar{x}_2$. We could build one "and" term for each input vector and then "or" them all together, but we might save considerably by factoring out common subsets of variables. Given a set of feasible "and" terms, each of which covers a

subset of the vectors we need, we seek to "or" together the smallest number of terms that realize the function. This is exactly the set cover problem.

There are several variations of set cover problems to be aware of:

- *Are you allowed to cover elements more than once?* – The distinction here is between *set cover* and *set packing*, which is discussed in Section 18.2 (page 625). We should take advantage of the freedom to cover elements multiple times if we have it, as it usually results in a smaller covering.

- *Are your sets derived from the edges or vertices of a graph?* – Set cover is a very general problem, and includes several useful graph problems as special cases. Suppose instead that you seek the smallest set of edges in a graph that will cover each vertex at least once. The solution is the maximum *matching* in the graph (see Section 15.6 (page 498)), plus arbitrary edges to cover any unmatched vertices. Now suppose instead that you seek the smallest set of vertices that cover each edge at least once. This is the *vertex cover* problem, discussed in Section 16.3 (page 530).

  It is instructive to show how to model vertex cover as an instance of set cover. Let the universal set U correspond to the set of edges $\{e_1, \ldots, e_m\}$. Construct n subsets, with S_i consisting of the edges incident on vertex v_i. Although vertex cover is just an instance of set cover in disguise, you should take advantage of the superior heuristics that exist for the more restricted vertex cover problem.

- *Do your subsets contain only two elements each?* – You are in luck if all of your subsets have at most two elements each. This special case can be solved efficiently to optimality because it reduces to finding a maximum matching in a graph. Unfortunately, the problem becomes NP-complete as soon as your subsets have three elements each.

- *Do you want to cover elements with sets, or sets with elements?* – In the *hitting set* problem, we seek a small number of items that together represent each subset in a given population. Hitting set is illustrated in Figure 18.1. The input is identical to set cover, but instead we seek the smallest subset of elements $T \subset U$ such that each subset S_i contains at least one element of T. Thus, $S_i \cap T \neq \emptyset$ for all $1 \leq i \leq m$. Suppose we desire a small Congress with at least one representative for each ethnic group. If each ethnic group is defined by a subset of people, the minimum hitting set gives the smallest possible politically correct Congress.

  Hitting set is *dual* to set cover, meaning that it is exactly the same problem in disguise. Replace each element of U by a set of the names of the subsets that contain it. Now S and U have exchanged roles, for we seek a set of subsets from U to cover all the elements of S. This is exactly set cover, so we can use any set cover code to solve hitting set after performing this simple translation. See Figure 18.1 for an example.

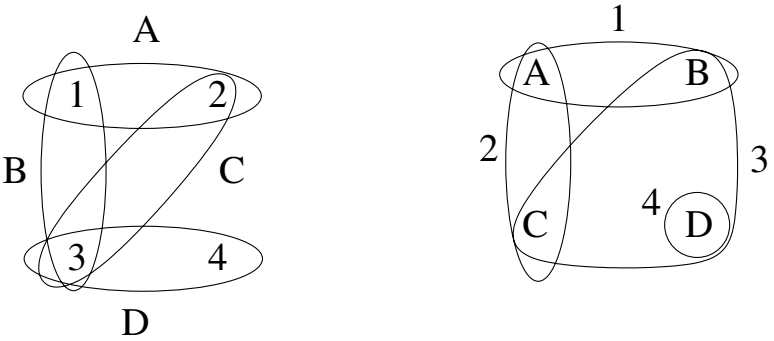


Figure 18.1: A hitting set instance optimally solved by selecting elements 1 and 3 or 2 and 3 (l). This problem converted to a dual set cover instance optimally solved by selecting subsets 1 and 3 or 2 and 4 (r).

Set cover must be at least as hard as vertex cover, so it is also NP-complete. In fact, it is somewhat harder. Approximation algorithms do no worse than twice optimal for vertex cover, but the best approximation algorithm for set cover is $\Theta(\lg n)$ times optimal.

Greedy is the most natural and effective heuristic for set cover. Begin by selecting the largest subset for the cover, and then delete all its elements from the universal set. We add the subset containing the largest number of remaining uncovered elements repeatedly until all are covered. This heuristic always gives a set cover using at most $\ln n$ times as many sets as optimal. In practice it usually does a lot better.

The simplest implementation of the greedy heuristic sweeps through the entire input instance of m subsets for each greedy step. However, by using such data structures as linked lists and a bounded-height priority queue (see Section 12.2 (page 373)), the greedy heuristic can be implemented in $O(S)$ time, where $S = \cup_{i=1}^{m} |S_i|$ is the size of the input representation.

It pays to check whether or not there exist elements that exist in only a few subsets—ideally only one. If so, we should select the biggest subsets containing these elements at the very beginning. We must take such a subset eventually, and they carry along other elements that we might have paid extra to cover if we wait until later.

Simulated annealing is likely to produce somewhat better set covers than these simple heuristics. Backtracking can be used to guarantee you an optimal solution, but it is usually not worth the computational expense.

An alternate and more powerful approach rests on the integer programming formulation of set cover. Let the integer 0-1 variable s_i denote whether subset S_i is selected for a given cover. Each universal set element x adds the constraint

$$\sum_{x \in S_i} s_i \geq 1$$

to ensure that it is covered by at least one selected subset. The minimum set cover satisfies all constraints while minimizing $\sum_i s_i$. This integer program can be easily generalized to weighted set cover (allowing nonuniform costs for different subsets. Relaxing this to a linear program (i.e. , allowing $0 \leq s_i \leq 1$ instead of constricting each variable to be either 0 or 1) allows efficient and effective heuristics using rounding techniques.

Implementations: Both the greedy heuristic and the above ILP formulation are sufficiently simple in their respective worlds that one has to implement them from scratch.

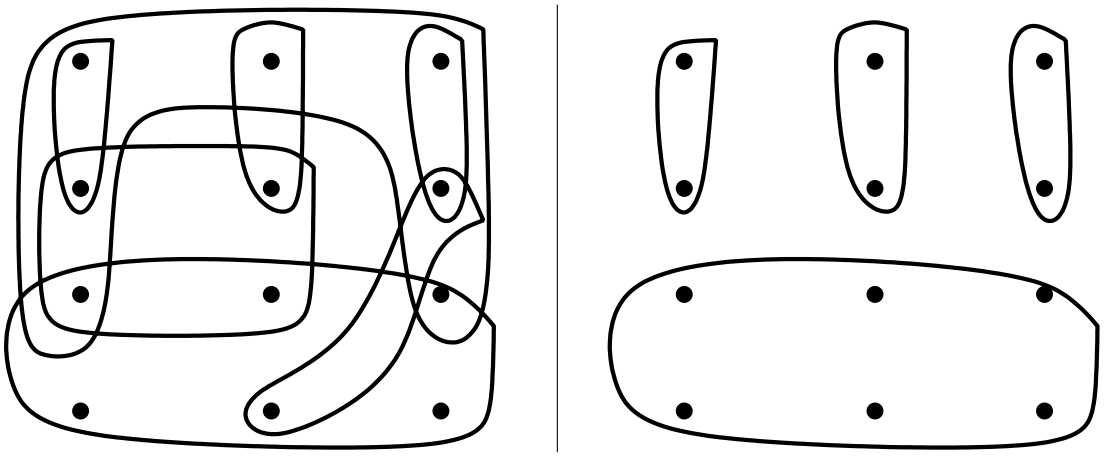
Pascal implementations of an exhaustive search algorithm for set packing, as well as heuristics for set cover, appear in [SDK83]. See Section 19.1.10 (page 662).

SYMPHONY is a mixed-integer linear programming solver that includes a set partitioning solver. It is available at *http://branchandcut.org/SPP/*

Notes: An old but classic survey article on set cover is [BP76], with more recent approximation and complexity analysis surveyed in [Pas97]. See [CFT99, CFT00] for extensive computational studies of integer programming-based set cover heuristics and exact algorithms. An excellent exposition on algorithms and reduction rules for set cover is presented in [SDK83].

Good expositions of the greedy heuristic for set cover include [CLRS01, Hoc96]. An example demonstrating that the greedy heuristic for set cover can be as bad as $\lg n$ is presented in [Joh74, PS98]. This is not a defect of the heuristic. Indeed, it is provably hard to approximate set cover to within an approximation factor better than $(1 - o(1)) \ln n$ [Fei98].

Related Problems: Matching (see page 498), vertex cover (see page 530), set packing (see page 625).



INPUT OUTPUT

18.2 Set Packing

Input description: A set of subsets $S = \{S_1, \ldots, S_m\}$ of the universal set $U = \{1, \ldots, n\}$.

Problem description: Select (an ideally small) collection of mutually disjoint subsets from S whose union is the universal set.

Discussion: Set-packing problems arise in applications where we have strong constraints on what is an allowable partition. The key feature of packing problems is that no elements are permitted to be covered by more than one selected subset.

Some flavor of this is captured by the independent set problem in graphs, discussed in Section 16.2 (page 528). There we seek a large subset of vertices from graph G such that each edge is adjacent to at most one of the selected vertices. To model this as set packing, let the universal set consist of all edges of G, and subset S_i consist of all edges incident on vertex v_i. Finally, define an additional singleton set for each edge. Any set packing defines a set of vertices with no edge in common—in other words, an independent set. The singleton sets are used to pick up any edges not covered by the selected vertices.

Scheduling airline flight crews is another application of set packing. Each airplane in the fleet needs to have a crew assigned to it, consisting of a pilot, copilot, and navigator. There are constraints on the composition of possible crews, based on their training to fly different types of aircraft, personality conflicts, and work schedules. Given all possible crew and plane combinations, each represented by a subset of items, we need an assignment such that each plane and each person is

in exactly one chosen combination. After all, the same person cannot be on two different planes simultaneously, and every plane needs a crew. We need a perfect packing given the subset constraints.

Set packing is used here to represent several problems on sets, all of which are NP-complete:

- *Must every element appear in exactly one selected subset?* – In the *exact cover* problem, we seek some collection of subsets such that each element is covered exactly once. The airplane scheduling problem above has the flavor of exact covering, since every plane and crew has to be employed.

  Unfortunately, exact cover puts us in a situation similar to that of Hamiltonian cycle in graphs. If we really *must* cover all the elements exactly once, and this existential problem is NP-complete, then all we can do is exponential search. The cost will be prohibitive unless we happen to stumble upon a solution quickly.

- *Does each element have its own singleton set?* – Things will be far better if we can be content with a partial solution, say by including each element of U as a singleton subset of S. Thus, we can expand any set packing into an exact cover by mopping up the unpacked elements of U with singleton sets. Now our problem is reduced to finding a minimum-cardinality set packing, which can be attacked via heuristics.

- *What is the penalty for covering elements twice?* – In set cover (see Section 18.1 (page 621)), there is no penalty for elements existing in many selected subsets. In exact cover, any such violation is forbidden. For many applications, the truth lies somewhere in between. Such problems can be approached by charging the greedy heuristic more to select a subset that contains previously covered elements.

The right heuristics for set packing are greedy, and similar to those of set cover (see Section 18.1 (page 621)). If we seek a packing with many (few) sets, then we repeatedly select the smallest (largest) subset, delete all subsets from S that clash with it, and repeat. As usual, augmenting this approach with some exhaustive search or randomization (in the form of simulated annealing) is likely to yield better packings at the cost of additional computation.

An alternate and more powerful approach rests on an integer programming formulation akin to that of set cover. Let the integer 0-1 variable s_i denote whether subset S_i is selected for a given cover. Each universal set element x adds the constraint

$$\sum_{x \in S_i} s_i = 1$$

to ensure that it is covered by *exactly* one selected subset. Minimizing or maximizing $\sum_i s_i$ while respecting these constraints enables us modulate the desired number of sets in the cover.

Implementations: Since set cover is a more popular and more tractable problem than set packing, it might be easier to find an appropriate implementation to solve the cover problem. Such implementations discussed in Section 18.1 (page 621) should be readily modifiable to support certain packing constraints.

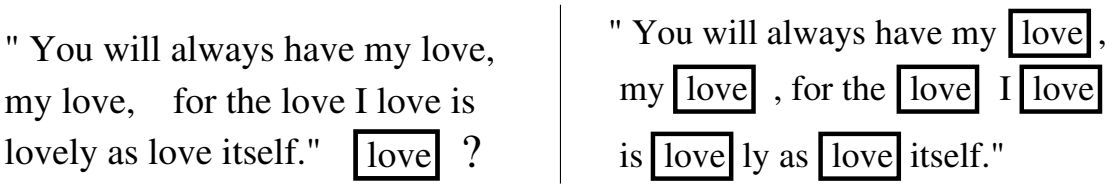
Pascal implementations of an exhaustive search algorithm for set packing, as well as heuristics for set cover, appear in [SDK83]. See Section 19.1.10 (page 662) for details on FTP-ing these codes.

SYMPHONY is a mixed-integer linear programming solver that includes a set partitioning solver. It is available at *http://branchandcut.org/SPP/*.

Notes: Survey articles on set packing include [BP76, Pas97]. Bidding strategies for combinatorial auctions typically reduce to solving set-packing problems, as described in [dVV03].

Set-packing relaxations for integer programs are presented in [BW00]. An excellent exposition on algorithms and reduction rules for set packing is presented in [SDK83], including the airplane scheduling application discussed previously.

Related Problems: Independent set (see page 528), set cover (see page 621).



18.3 String Matching

Input description: A text string t of length n. A pattern string p of length m.

Problem description: Find the first (or all) instances of pattern p in the text.

Discussion: String matching arises in almost all text-processing applications. Every text editor contains a mechanism to search the current document for arbitrary strings. Pattern-matching programming languages such as Perl and Python derive much of their power from their built-in string matching primitives, making it easy to fashion programs that filter and modify text. Spelling checkers scan an input text for words appearing in the dictionary and reject any strings that do not match.

Several issues arise in identifying the right string matching algorithm for a given application:

- *Are your search patterns and/or texts short?* – If your strings are sufficiently short and your queries sufficiently infrequent, the simple $O(mn)$-time search algorithm will suffice. For each possible starting position $1 \leq i \leq n - m + 1$, it tests whether the m characters starting from the ith position of the text are identical to the pattern. An implementation of this algorithm (in C) is given in Section 2.5.3 (page 43).

  For very short patterns (say $m \leq 5$), you can't hope to beat this simple algorithm by much, so you shouldn't try. Further, we expect much better than $O(mn)$ behavior for typical strings, because we advance the pattern the instant we observe a text/pattern mismatch, Indeed, the trivial algorithm *usually* runs in linear time. But the worst case certainly can occur, as with pattern $p = a^m$ and text $t = (a^{m-1}b)^{n/m}$.

- *What about longer texts and patterns?* – String matching can in fact be performed in worst-case linear time. Observe that we need not begin the search from scratch on finding a character mismatch, since the pattern prefix and text must exactly match up to the point of mismatch. Given a long partial

match ending at position i, we jump ahead to the first character position in the pattern/text that can provide new information about the text in position $i + 1$. The Knuth-Morris-Pratt algorithm preprocesses the search pattern to construct such a jump table efficiently. The details are tricky to get correct, but the resulting algorithm yields short, simple programs.

- *Do I expect to find the pattern or not?* – The Boyer-Moore algorithm matches the pattern against the text from right to left, and can avoid looking at large chunks of text on a mismatch. Suppose the pattern is *abracadabra*, and the eleventh character of the text is x. This pattern cannot match in any of the first eleven starting positions of the text, and so the next necessary position to test is the 22nd character. If we get very lucky, only n/m characters need ever be tested. The Boyer-Moore algorithm involves two sets of jump tables in the case of a mismatch: one based on pattern matched so far, the other on the text character seen in the mismatch.

  Although somewhat more complicated than Knuth-Morris-Pratt, it is worth it in practice for patterns of length $m > 5$, unless the pattern is expected to occur many times in the text. Its worst-case performance is $O(n + rm)$, where r is the number of occurrences of p in t.

- *Will you perform multiple queries on the same text?* – Suppose you are building a program to repeatedly search a particular text database, such as the Bible. Since the text remains fixed, it pays to build a data structure to speed up search queries. The suffix tree and suffix array data structures, discussed in Section 12.3 (page 377), are the right tools for the job.

- *Will you search many texts using the same patterns?* – Suppose you are building a program to screen out dirty words from a text stream. Here, the set of patterns remains stable, while the search texts are free to change. In such applications, we may need to find all occurrences of any of k different patterns where k can be quite large.

  Performing a linear-time scan for each pattern yields an $O(k(m + n))$ algorithm. If k is large, a better solution builds a single finite automaton that recognizes all of these patterns and returns to the appropriate start state on any character mismatch. The Aho-Corasick algorithm builds such an automaton in linear time. Space savings can be achieved by optimizing the pattern recognition automaton, as discussed in Section 18.7 (page 646). This approach was used in the original version of *fgrep*.

  Sometimes multiple patterns are specified not as a list of strings, but concisely as a regular expression. For example, the regular expression $a(a + b + c)^*a$ matches any string on (a, b, c) that begins and ends with a distinct a. The best way to test whether an input string is described by a given regular expression R constructs the finite automaton equivalent to R and then simulates

the machine on the string. Again, see Section 18.7 (page 646) for details on constructing automata from regular expressions.

When the patterns are specified by context-free grammars instead of regular expressions, the problem becomes one of parsing, discussed in Section 8.6 (page 298).

- *What if our text or pattern contains a spelling error?* – The algorithms discussed here work only for exact string matching. If you want to allow some tolerance for spelling errors, your problem becomes *approximate string matching*, which is thoroughly discussed in Section 18.4 (page 631).

Implementations: Strmat is a collection of C programs implementing exact pattern matching algorithms in association with [Gus97], including several variants of the KMP and Boyer-Moore algorithms. It is available at *http://www.cs.ucdavis.edu/~gusfield/strmat.html*.

SPARE Parts [WC04a] is a C++ string pattern recognition toolkit that provides production-quality implementations of all major variants of the classical string-matching algorithms for single patterns (both Knuth-Morris-Pratt and Boyer-Moore) and multiple patterns (both Aho-Corasick and Commentz-Walter). It is available at *http://www.fastar.org/*.

Several versions of the general regular expression pattern matcher (grep) are readily available. GNU *grep* found at *http://directory.fsf.org/project/grep/*, and supersedes variants such as *egrep* and *fgrep*. GNU *grep* uses a fast lazy-state deterministic matcher hybridized with a Boyer-Moore search for fixed strings.

The Boost string algorithms library provides C++ routines for basic operations on strings, including search. See *http://www.boost.org/doc/html/string_algo.html*.

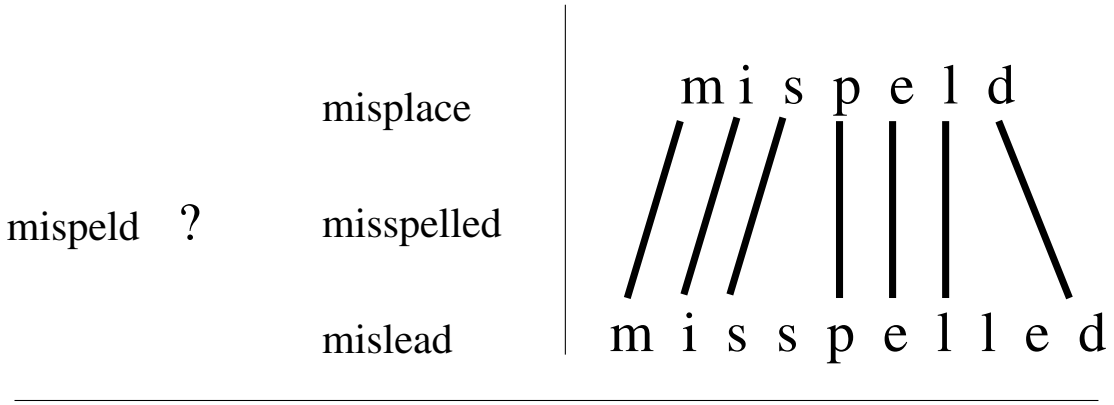
Notes: All books on string algorithms contain thorough discussions of exact string matching, including [CHL07, NR07, Gus97]. Good expositions on the Boyer-Moore [BM77] and Knuth-Morris-Pratt algorithms [KMP77] include [BvG99, CLRS01, Man89]. The history of string matching algorithms is somewhat checkered because several published proofs were incorrect or incomplete. See [Gus97] for clarification.

Aho [Aho90] provides a good survey on algorithms for pattern matching in strings, particularly where the patterns are regular expressions instead of strings. The Aho-Corasick algorithm for multiple patterns is described in [AC75].

Empirical comparisons of string matching algorithms include [DB86, Hor80, Lec95, dVS82]. Which algorithm performs best depends upon the properties of the strings and the size of the alphabet. For long patterns and texts, I recommend that you use the best implementation of Boyer-Moore that you can find.

The Karp-Rabin algorithm [KR87] uses a hash function to perform string matching in linear expected time. Its worst-case time remains quadratic, and its performance in practice appears somewhat worse than the character comparison methods described above. This algorithm is presented in Section 3.7.2 (page 91).

Related Problems: Suffix trees (see page 377), approximate string matching (see page 631).



18.4 Approximate String Matching

Input description: A text string t and a pattern string p.

Problem description: What is the minimum-cost way to transform t to p using insertions, deletions, and substitutions?

Discussion: Approximate string matching is a fundamental problem because we live in an error-prone world. Spelling correction programs must be able to identify the closest match for any text string not found in a dictionary. By supporting efficient sequence similarity (homology) searches on large databases of DNA sequences, the computer program BLAST has revolutionized the study of molecular biology. Suppose you were interested in a particular gene in man, and discovered that it is similar to the hemoglobin gene in rats. Likely this new gene also produces hemoglobin, and any differences are the result of genetic mutations during evolution.

I once encountered approximate string matching when evaluating the performance of an optical character-recognition system. We needed to compare the answers produced by our system on a test document with the correct results. To improve our system, we needed to identify (1) which letters were getting misidentified and (2) gibberish when the program found letters that didn't exist. The solution was to do an alignment between the two texts. Insertions and deletions corresponded to gibberish, while substitutions signaled errors in our recognizer. This same principle is used in file difference programs, which identify the lines that have changed between two versions of a file.

When no errors are permitted, our problem reduces to exact string matching, which is presented in Section 18.3 (page 628). Here, we restrict our discussion to matching with errors.

Dynamic programming provides the basic approach to approximate string matching. Let $D[i,j]$ denote the cost of editing the first i characters of the pattern string p into the first j characters of the text t. The recurrence follows because we must have done *something* with the tail characters p_i and t_j. Our only options are matching / substituting one for the other, deleting p_i, or inserting a match for t_j. Thus, $D[i,j]$ is the minimum of the costs of these possibilities:

1. If $p_i = t_j$ then $D[i-1,j-1]$ else $D[i-1,j-1]$ + substitution cost.
2. $D[i-1,j]$ + deletion cost of p_i.
3. $D[i,j-1]$ + deletion cost of t_j.

A general implementation in C and more complete discussion appears in Section 8.2 (page 280). Several issues remain before we can make full use of this recurrence:

- *Do I match the pattern against the full text, or against a substring?* – The boundary conditions of this recurrence distinguishes between algorithms for string matching and substring matching. Suppose we want to align the full pattern against the full text. Then the cost of $D[i,0]$ must be that of deleting the first i characters of the pattern, so $D[i,0] = i$. Similarly, $D[0,j] = j$.

  Now suppose that the pattern can occur anywhere within the text. The proper cost of $D[0,j]$ is now 0, since there should be no penalty for starting the alignment in the jth position of the text. The cost of $D[i,0]$ remains i, because the only way to match the first i pattern characters with nothing is to delete all of them. The cost of the best substring pattern match against the text will be given by $\min_{k=1}^{n} D[m,k]$.

- *How should I select the substitution and insertion/deletion costs?* – The basic algorithm can be easily modified to use different costs for insertion, deletion, and the substitutions of specific pairs of characters. Which costs you should use depend on what you are planning to do with the alignment.

  The most common cost assignment charges the same for insertion, deletion, or substitution. Charging a substitution cost of more than insertion + deletion ensures that substitutions never get performed, since it will always be cheaper to edit both characters out of the string. If we just have insertion and deletion to work with, the problem reduces to *longest common subsequence*, discussed in Section 18.8 (page 650). It often pays to tweak the edit distance costs and study the resulting alignments until you find the best parameters for the job.

- *How do I find the actual alignment of the strings?* – As thus far described, the recurrence only gives the cost of the optimal string/pattern alignment, not the sequence of editing operations to achieve it. To obtain such a transcript, we can work backwards from the complete cost matrix D. We had to come from one of $D[m-1,n]$ (pattern deletion/text insertion), $D[m,n-1]$

(text deletion/pattern insertion), or $D[m-1, n-1]$ (substitution/match) to get to cell $D[m, n]$. The option which was chosen can be reconstructed from these costs and the given characters p_m and t_n. By continuing to work backwards to the previous cell, we can reconstruct the entire alignment. Again, an implementation in C appears in Section 8.2 (page 280).

- *What if the two strings are very similar to each other?* – The dynamic programming algorithm finds a shortest path across an $m \times n$ grid, where the cost of each edge depends upon which operation it represents. To seek an alignment involving a combination of at most d insertions, deletions, and substitutions, we need only traverse the band of $O(dn)$ cells within a distance d of the central diagonal. If no low-cost alignment exists within this band, then no low-cost alignment can exist in the full cost matrix.

  Another idea we can use is *filtration*, quickly eliminating the parts of the string where there is no hope of finding the pattern. Carve the m-length pattern into $d+1$ pieces. If there is a match with at most d differences, then at least one of these pieces must be an exact match in the optimal alignment. Thus, we can identify all possible approximate match points by conducting an exact multi-pattern search on the pieces, and then evaluate only the possible candidates more carefully.

- *Is your pattern short or long?* – A recent approach to string-matching exploits the fact that modern computers can do operations on (say) 64-bit words in a single gulp. This is long enough to hold eight 8-bit ASCII characters, providing motivation to design *bit-parallel algorithms*, which do more than one comparison with each operation.

  The basic idea is quite clever. Construct a bit-mask B_α for each letter α of the alphabet, such that ith-bit $B_\alpha[i] = 1$ iff the ith character of the pattern is α. Now suppose you have a match bit-vector M_j for position j in the text string, such that $M_j[i] = 1$ iff the first i bits of the pattern exactly match the $(j-i+1)$st through jth character of the text. We can find *all* the bits of M_{j+1} using just two operations by (1) shifting M_j one bit to the right, and then (2) doing a bitwise AND with B_α, where α is the character in position $j+1$ of the text.

  The *agrep* program, discussed below, uses such a bit-parallel algorithm generalized to approximate matching. Such algorithms are easy to program and many times faster than dynamic programming.

- *How can I minimize the required storage?* – The quadratic space used to store the dynamic programming table is usually a more serious problem than its running time. Fortunately, only $O(\min(m, n))$ space is needed to compute $D[m, n]$. We need only maintain two active rows (or columns) of the matrix to compute the final value. The entire matrix will be required only if we need to reconstruct the actual sequence alignment.

We can use Hirschberg's clever recursive algorithm to efficiently recover the optimal alignment in linear space. During one pass of the linear-space algorithm above to compute $D[m, n]$, we identify which middle-element cell $D[m/2, x]$ was used to optimize $D[m, n]$. This reduces our problem to finding the best paths from $D[1, 1]$ to $D[m/2, x]$ and from $D[m/2, x]$ to $D[m/2, n]$, both of which can be solved recursively. Each time we remove half of the matrix elements from consideration, so the total time remains $O(mn)$. This linear-space algorithm proves to be a big win in practice on long strings, although it is somewhat more difficult to program.

- *Should I score long runs of indels differently?* – Many string matching applications look more kindly on alignments where insertions/deletions are bunched in a small number of runs or gaps. Deleting a word from a text should presumably cost less than a similar number of scattered single-character deletions, because the word represents a single (albeit substantial) edit operation.

  String matching with *gap penalties* provides a way to properly account for such operations. Typically, we assign a cost of $A + Bt$ for each indel of t consecutive characters, where A is the cost of starting the gap and B is the per-character deletion cost. If A is large relative to B, the alignment has incentive to create relatively few runs of deletions.

  String matching under such *affine* gap penalties can be done in the same quadratic time as regular edit distance. We will use separate insertion and deletion recurrences E and F to encode the cost of being in gap mode, meaning we have already paid the cost of initiating the gap:

$$V(i, j) = \max(E(i, j), F(i, j), G(i, j))$$

$$G(i, j) = V(i - 1, j - 1) + match(i, j)$$

$$E(i, j) = \max(E(i, j - 1), V(i, j - 1) - A) - B$$

$$F(i, j) = \max(F(i - 1, j), V(i - 1, j) - A) - B$$

  With a constant amount of work per cell, this algorithm takes $O(mn)$ time, same as without gap costs.

- *Does similarity mean strings that sound alike?* – Other models of approximate pattern matching become more appropriate for certain applications. Particularly interesting is *Soundex*, a hashing scheme that attempts to pair up English words that sound alike. This can be useful in testing whether two names that have been spelled differently are likely to be the same. For example, my last name is often spelled "Skina", "Skinnia", "Schiena", and occasionally "Skiena." All of these hash to the same Soundex code, *S25*.

The algorithm drops vowels and silent letters, removes doubled letters, and then assigns the remaining letters numbers from the following classes: *BFPV* gets a 1, *CGJKQSXZ* gets a 2, *DT* gets a 3, *L* gets a 4, *MN* gets a 5, and *R* gets a 6. The code starts with the first letter and contains at most three digits. Although this sounds very hokey, experience shows that it works reasonably well. Experience indeed: Soundex has been used since the 1920's.

Implementations: Several excellent software tools are available for approximate pattern matching. Manber and Wu's *agrep* [WM92a, WM92b] (approximate general regular expression pattern matcher) is a tool supporting text search with spelling errors. A recent version is available from *http://www.tgries.de/agrep/*. Navarro's *nrgrep* [Nav01b] combines bit-parallelism and filtration, resulting in running times that are more constant than *agrep*, although not always faster. It is available at *http://www.dcc.uchile.cl/~gnavarro/software/*.

TRE is a general regular-expression matching library for exact and approximate matching, which is more general than *agrep*. The worst-case complexity is $O(nm^2)$, where m is the list of the regular expressions involved. *TRE* is available at *http://laurikari.net/tre/*.

Wikipedia gives programs for computing edit (Levenshtein) distance in a dizzying array of languages (including Ada, C++, Emacs Lisp, Io, JavaScript, Java, PHP, Python, Ruby VB, and C#) Check it out at:

http://en.wikibooks.org/wiki/Algorithm_implementation/Strings/Levenshtein_distance

Notes: There have been many recent advances in approximate string matching, particularly in bit-parallel algorithms. Navarro and Raffinot [NR07] is the best reference on these recent techniques, which are also treated in other recent books on string algorithmics [CHL07, Gus97]. String matching with gap penalties is particularly well treated in [Gus97].

The basic dynamic programming alignment algorithm is attributed to [WF74], although it is apparently folklore. The wide range of applications for approximate string matching was made apparent in Sankoff and Kruskal's book [SK99], which remains a useful historical reference for the problem. Surveys on approximate pattern matching include [HD80, Nav01a]. The edit distance between two strings is sometimes referred to as the *Levenshtein distance*. Expositions of Hirschberg's linear-space algorithm [Hir75] include [CR03, Gus97].

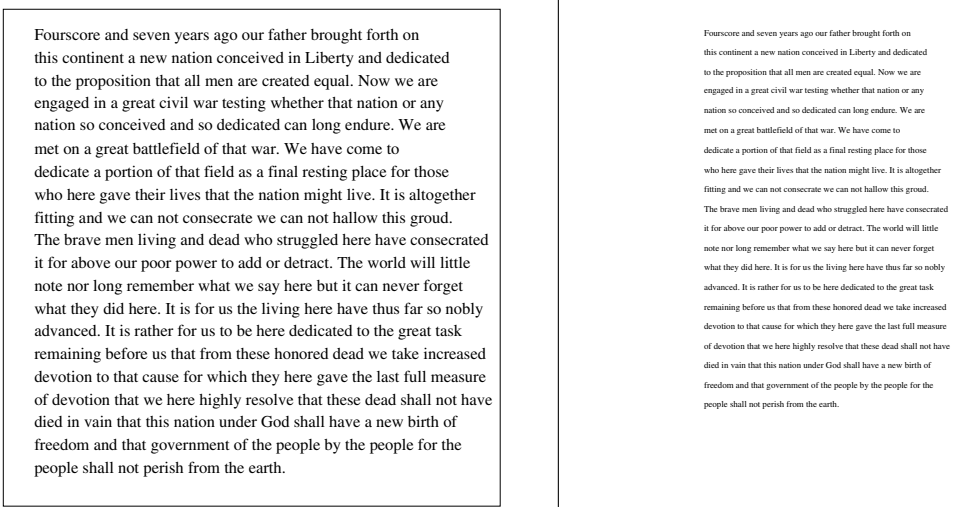
Masek and Paterson [MP80] compute the edit distance between m- and n-length strings in time $O(mn/\log(\min\{m,n\}))$ for constant-sized alphabets, using ideas from the four Russians algorithm for Boolean matrix multiplication [ADKF70].

The shortest path formulation leads to a variety of algorithms that are good when the edit distance is small, including an $O(n \lg n + d^2)$ algorithm due to Myers [Mye86] and an $O(dn)$ algorithm due to Landau and Vishkin [LV88]. Longest increasing subsequence can be done in $O(n \lg n)$ time [HS77], as presented in [Man89].

Bit-parallel algorithms for approximate matching include Myers's [Mye99b] algorithm for approximate matching in $O(mn/w)$ time, where w is the number of bits in the computer word. Experimental studies of bit-parallel algorithms include [FN04, HFN05, NR00].

Soundex was invented and patented by M. K. Odell and R. C. Russell. Expositions on Soundex include [BR95, Knu98]. Metaphone is a recent attempt to improve on Soundex [BR95, Par90]. See [LMS06] for an application of such phonetic hashing techniques to the problem entity name unification.

Related Problems: String matching (see page 628), longest common substring (see page 650).



INPUT OUTPUT

18.5 Text Compression

Input description: A text string S.

Problem description: Create a shorter text string S' such that S can be correctly reconstructed from S'.

Discussion: Secondary storage devices fill up quickly on every computer system, even though their capacity continues to double every year. Decreasing storage prices only seem to have increased interest in data compression, probably because there is more data to compress than ever before. *Data compression* is the algorithmic problem of finding space-efficient encodings for a given data file. The rise of computer networks provided a new mission for data compression, that of increasing the effective network bandwidth by reducing the number of bits before transmission.

People seem to *like* inventing ad hoc data-compression methods for their particular application. Sometimes these outperform general methods, but often they don't. Several issues arise in selecting the right compression algorithm:

- *Must we recover the exact input text after compression? – Lossy* versus *lossless* encoding is the primary issue in data compression. Document storage applications typically demand lossless encodings, as users become disturbed

whenever their data files are altered. Fidelity is not such an issue in image or video compression, because the presence of small artifacts are imperceptible to the viewer. Significantly greater compression ratios can be obtained using lossy compression, which is why most image/video/audio compression algorithms exploit this freedom.

- *Can I simplify my data before I compress it?* – The most effective way to free space on a disk is to delete files you don't need. Likewise, any preprocessing you can do to reduce the information content of a file pays off later in better compression. Can we eliminate redundant white space from the file? Might the document be converted entirely to uppercase characters, or have formatting information removed?

  A particularly interesting simplification results from applying the *Burrows-Wheeler transform* to the input string. This transform sorts all n cyclic shifts of the n character input, and then reports the last character of each shift. As an example, the cyclic shifts of $ABAB$ are $ABAB$, $BABA$, $ABAB$, and $BABA$. After sorting, these become $ABAB$, $ABAB$, $BABA$, and $BABA$. Reading the last character of each of these strings yields the transform result: $BBAA$.

  Provided the last character of the input string is unique (e.g., end-of-string), this transform is perfectly reversible to the original input! The Burrows-Wheeler string is typically 10-15% more compressible than the original text, because repeated words turn into blocks of repeated characters. Further, this transform can be computed in linear time.

- *Does it matter whether the algorithm is patented?* – Certain data compression algorithms have been patented—most notoriously the LZW variation of the Lempel-Ziv algorithm discussed below. Mercifully, this patent has now expired, although legal battles are still being fought over JPEG. Typically there are unrestricted variations of any compression algorithm that perform about as well as the patented variant.

- *How do I compress image data* – The simplest lossless compression algorithm for image data is *run-length coding*. Here we replace runs of identical pixel values with a single instance of the pixel and an integer giving the length of the run. This works well on binary images with large contiguous regions of similar pixels, like scanned text. It performs badly on images with many quantization levels and random noise. Correctly selecting (1) the number of bits to allocate to the count field, and (2) the right traversal order to reduce a two-dimensional image into a stream of pixels, has a surprisingly important impact on compression.

  For serious audio/image/video compression applications, I recommend that you use a popular lossy coding method and not fool around with implementing it yourself. JPEG is the standard high-performance image compression

method, while MPEG is designed to exploit the frame-to-frame coherence of video.

- *Must compression run in real time?* – Fast decompression is often more important than fast compression. A YouTube video is compressed only once, but decompressed every time someone plays it. In contrast, an operating system that increases effective disk capacity by automatically compressing files will need a symmetric algorithm with fast compression times, as well.

Literally dozens of text compression algorithms are available, but they can be classified into two distinct approaches. *Static algorithms*, such as Huffman codes, build a single coding table by analyzing the entire document. *Adaptive algorithms*, such as Lempel-Ziv, build a coding table on the fly that adapts to the local character distribution of the document. Adaptive algorithms usually prove to be the correct answer:

- *Huffman codes* – Huffman codes replace each alphabet symbol by a variable-length code string. Using eight bits-per-symbol to encode English text is wasteful, since certain characters (such as "e") occur far more frequently than others (such as "q"). Huffman codes assign "e" a short code word, and "q" a longer one to compress text.

  Huffman codes can be constructed using a greedy algorithm. Sort the symbols in increasing order by frequency. We merge the two least-frequently used symbols x and y into a new symbol xy, whose frequency is the sum of the frequencies of its two child symbols. Replacing x and y by xy leaves a smaller set of symbols. We now repeat this operation $n - 1$ times until all symbols have been merged. These merging operations define a rooted binary tree, with the original alphabet symbols as leaves. The left or right choices on the root-to-leaf path define the bits of the binary code word for each symbol. Priority queues can efficiently maintain the symbols by frequency during construction, yielding Huffman codes in $O(n \lg n)$ time.

  Huffman codes are popular but have three disadvantages. Two passes must be made over the document on encoding, first to build the coding table, and then to actually encode the document. The coding table must be explicitly stored with the document to decode it, which eats into any space savings on short documents. Finally, Huffman codes only exploit nonuniform symbol distributions, while adaptive algorithms can recognize the higher-order redundancies such as in *0101010101...*.

- *Lempel-Ziv algorithms* – Lempel-Ziv algorithms (including the popular LZW variant) compress text by building a coding table on the fly as we read the document. The coding table changes at every position in the text. A clever protocol ensures that the encoder and decoder are both always working with the exact same code table, so no information is lost.

> Lempel-Ziv algorithms build coding tables of frequent substrings, which can get arbitrarily long. Thus they can exploit often-used syllables, words, and phrases to build better encodings. It adapts to local changes in the text distribution, which is important because many documents exhibit significant locality of reference.
>
> The truly amazing thing about the Lempel-Ziv algorithm is how robust it is on different types of data. It is quite difficult to beat Lempel-Ziv by using an application-specific algorithm. My recommendation is not to try. If you can eliminate application-specific redundancies with a simple preprocessing step, go ahead and do it. But don't waste much time fooling around. You are unlikely to get significantly better text compression than with *gzip* or some other popular program, and you might well do worse.

Implementations: Perhaps the most popular text compression program is *gzip*, which implements a public domain variation of the Lempel-Ziv algorithm. It is distributed under the GNU software license and can be obtained from *http://www.gzip.org*.

There is a natural tradeoff between compression ratio and compression time. Another choice is *bzip2*, which uses the Burrows-Wheeler transform. It produces tighter encodings than *gzip* at somewhat greater cost in running time. Going to the extreme, other compression algorithms devote enormous run times to squeeze every bit out of a file. Representative programs of this genre are collected at *http://www.cs.fit.edu/~mmahoney/compression/*.

Reasonably authoritative comparisons of compression programs are presented at *http://www.maximumcompression.com/*, including links to all available software.

Notes: A large number of books on data compression are available. Recent and comprehensive books include Sayood [Say05] and Salomon [Sal06]. Also recommended is the older book by Bell, Cleary, and Witten [BCW90]. Surveys on text compression algorithms include [CL98].

Good expositions on Huffman codes [Huf52] include [AHU83, CLRS01, Man89]. The Lempel-Ziv algorithm and variants are described in [Wel84, ZL78]. The Burrows-Wheeler transform was introduced in [BW94].

The annual IEEE Data Compression Conference (*http://www.cs.brandeis.edu/~dcc/*) is the primary research venue in this field. This is a mature technical area where most current work is shooting for fairly marginal improvements, particularly in the case of text compression. More encouragingly, we note that the conference is held annually at a world-class ski resort in Utah.

Related Problems: Shortest common superstring (see page 654), cryptography (see page 641).

The magic words are Squeamish Ossifrage.	I5&AE<&UA9VEC'=0 <F1s"F%R92!3<75E96UI<V V@*3W-S:69R86=E+@K_

INPUT OUTPUT

18.6 Cryptography

Input description: A plaintext message T or encrypted text E, and a key k.

Problem description: Encode T (decode E) using k giving E (T).

Discussion: Cryptography has grown substantially in importance as computer networks make confidential documents more vulnerable to prying eyes. Cryptography increases security by making messages difficult to read even if they fall into the wrong hands. Although the discipline of cryptography is at least two thousand years old, its algorithmic and mathematical foundations have only recently solidified to the point where provably secure cryptosystems can be envisioned.

Cryptographic ideas and applications go beyond the commonly known concepts of "encryption" and "authentication." The field now includes such important mathematical constructs such as cryptographic hashes, digital signatures, and useful primitive protocols that provide associated security assurances.

There are three classes of cryptosystems everyone should be aware of:

- *Caesar shifts* – The oldest ciphers involve mapping each character of the alphabet to a different letter. The weakest such ciphers rotate the alphabet by some fixed number of characters (often 13), and thus have only 26 possible keys. Better is to use an arbitrary permutation of the letters, giving 26! possible keys. Even so, such systems can be easily attacked by counting the frequency of each symbol and exploiting the fact that "e" occurs more often than "z". While there are variants that will make this more difficult to break, none will be as secure as AES or RSA.

- *Block Shuffle Ciphers* – This class of algorithms repeatedly shuffle the bits of your text as governed by the key. The classic example of such a cipher is the *Data Encryption Standard* (DES). Although approved as a Federal Information Processing Standard in 1976, its 56-bit key length is now considered too short for applications requiring substantial levels of security. Indeed, a special purpose machine named "Deep Crack" demonstrated that it is possible to decrypt messages without a key in less than a day. As of May 19,

2005, *DES* has been officially withdrawn as a federal standard, replaced by the stronger *Advanced Encryption Standard* (AES).

However, a simple variant called *triple DES* permits an effective key length of 112 bits by using three rounds of DES with two 56-bit keys. In particular, first encrypt with key1, then *decrypt* with key2, before finally encrypting with key1. There is a mathematical reason for using three rounds instead of two; the encrypt-decrypt-encrypt pattern is used so that the scheme is equivalent to single DES when $key1 = key2$. This is enough to keep "Deep Crack" at bay. Indeed, *triple DES* has recently been approved by the National Institute of Standards and Technology (NIST) for sensitive government information through the year 2030.

- *Public Key Cryptography* – If you fear bad guys reading your messages, you should be afraid to tell anyone else the key needed to decrypt them. Public-key systems use different keys to encode and decode messages. Since the encoding key is of no help in decoding, it can be made public at no risk to security. This solution to the key distribution problem is literally its key to success.

  RSA is the classic example of a public key cryptosystem, named after its inventors Rivest, Shamir, and Adelman. The security of RSA is based on the relative computational complexities of factoring and primality testing (see Section 13.8 (page 420)). Encoding is (relatively) fast because it relies on primality testing to construct the key, while the hardness of decryption follows from that of factoring. Still, RSA is slow relative to other cryptosystems—roughly 100 to 1,000 times slower than DES.

The critical issue in selecting a cryptosystem is identifying your paranoia level—i.e. , deciding how much security you need. Who are you trying to stop from reading your stuff: your grandmother, local thieves, the Mafia, or the NSA? If you can use an accepted implementation of AES or RSA, you should feel pretty safe against anybody, at least for now. Increasing computer power often lays waste to cryptosystems surprisingly quickly; recall that DES lived less than 30 years as a strong system. Be sure to use the longest possible keys and keep abreast of algorithmic development if you are a planning long-term storage of criminal material.

That said, I will confess that I use DES to encrypt my final exam each semester. It proved more than sufficient the time an ambitious student broke into my office looking for it. The story would have been different had the NSA had been breaking in, but it is important to understand that *the most serious security holes are human, not algorithmic*. Ensuring that your password is long enough, hard to guess, and not written down is far more important than obsessing about the encryption algorithm.

Most symmetric key encryption mechanisms are harder to crack than public key ones for the same key size. This means one can get away with much shorter key lengths for symmetric key than for public key encryption. NIST and RSA Labs

both provide schedules of recommended key sizes for secure encryption, and as of this writing they recommend 80-bit symmetric keys as equivalent to 1024-bit asymetric keys. This difference helps explain why symmetric key algorithms are typically orders of magnitude faster than public key algorithms.

Simple ciphers like the Caesar shift are fun and easy to program. For this reason, it is healthy to use them for applications needing only a casual level of security (such as hiding the punchlines of jokes). Since they are easy to break, they should never be used for serious security applications.

Another thing you should *never* do is try to develop your own novel cryptosystem. The security of triple DES and RSA is accepted because these systems have survived many years of public scrutiny. In this time, many other cryptosystems have been proposed, proven vulnerable to attack, and then abandoned. This is not a field for amateurs. If you are charged with implementing a cryptosystem, carefully study a respected program such as PGP to see how they handle issues such as key selection and key distribution. Any cryptosystem is as strong as its weakest link.

Certain other problems related to cryptography arise often in practice:

- *How can I validate the integrity of data against random corruption?* – There is often a need to validate that transmitted data is identical to that which has been received. One solution is for the receiver to transmit the data back to the source and have the original sender confirm that the two texts are identical. This fails when the exact inverse of an error is made in the retransmission, but a more serious problem is that your available bandwidth is cut in half with such a scheme.

  A more efficient method uses a *checksum*, a simple mathematical function that hashes a long text down to a simple number or digit. We then transmit the checksum along with the text. The checksum can be recomputed on the receiving end and bells set off if the computed checksum is not identical to what was received. The simplest checksum scheme just adds up the byte or character values and takes the sum modulo of some constant, say $2^8 = 256$. Unfortunately, an error transposing two or more characters would go undetected under such a scheme, since addition is commutative.

  Cyclic-redundancy check (CRC) provides a more powerful method for computing checksums that is used in most communications systems and internally in computers to validate disk drive transfers. These codes compute the remainder in the ratio of two polynomials, the numerator of which is a function of the input text. The design of these polynomials involves considerable mathematical sophistication, but ensures that all reasonable errors are detected. The details of efficient computation are sufficiently complicated that we recommend that you start from an existing implementation, described below.

- *How can I validate the integrity of data against deliberate corruption?* – CRC is good at detecting random errors, but not malicious changes to a document. *Cryptographic hashing functions* such as MD5 and SHA-256 are (in principle) easy to compute for a document but hard to invert. This means that given a particular hash code value x, it is difficult to construct a document d such that $H(d) = x$. The property makes them valuable for digital signatures and other applications.

- *How can I prove that a file has not been changed?* – If I send you a contract in electronic form, what is to stop you from editing the file and then claiming that your version was what we had really agreed to? I need a way to prove that any modification to a document is fraudulent. *Digital signatures* are a cryptographic way for me to stamp my document as genuine.

  Given a file, I can compute a checksum for it, and then encrypt this checksum using my own private key. I send you the file and the encrypted checksum. You can now edit the file, but to fool the judge you must also edit the encrypted checksum such that it can be decrypted to yield the correct checksum. With a suitably good checksum function, designing a file that yields the same checksum becomes an insurmountable problem. For full security, we need a trusted third party to authenticate the timestamp and associate the private key with me.

- *How can I restrict access to copyrighted material?* – An important emerging application for cryptography is digital rights management for audio and video. A key issue here is speed of decryption, as it must keep up with data transmission or retrieval in real time. Such *stream ciphers* usually involve efficiently generating a stream of pseudorandom bits, say using a shift-register generator. The exclusive-or of these bits with the data stream gives the encrypted sequence. The original data is recovered by exclusive-oring the result with the same stream of pseudorandom bits.

  High-speed cryptosystems have proven to be relatively easy to break. The state-of-the-art solution to this problem involves erecting laws like the Digital Millennium Copyright Act to make it illegal to try to break them.

Implementations: *Nettle* is a comprehensive low-level cryptographic library in C. Cryptographic hash functions include MD5 and SHA-256. Block ciphers include DES, AES, and some more recently developed codes. An implementation of RSA is also provided. *Nettle* is available at *http://www.lysator.liu.se/∼nisse/nettle*.

A comprehensive overview of cryptographic algorithms with assessments of strength is available at *http://www.cryptolounge.org/wiki/Category:Algorithm*. See *http://csrc.nist.gov/groups/ST/toolkit* for related cryptographic resources provided by NIST.

Crypto++ is a large C++ class library of cryptographic schemes, including all we have mentioned in this section. It is available at *http://www.cryptopp.com/*.

Many popular open source utilities employ serious cryptography, and serve as good models of current practice. *GnuPG*, an open source version of PGP, is available at *http://www.gnupg.org/*. *OpenSSL*, for authenticating access to computer systems, is available at *http://www.openssl.org/*.

The *Boost CRC Library* provides multiple implementations of cyclic redundancy check algorithms. It is available at *http://www.boost.org/libs/crc/*.

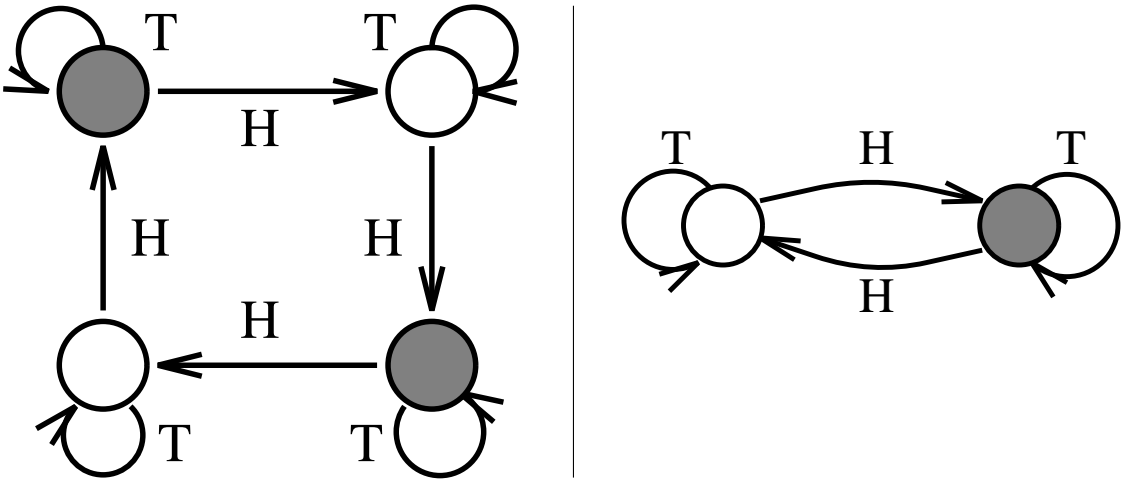
Notes: The *Handbook of Applied Cryptography* [MOV96] provides technical surveys of all aspects of cryptography, and has been generously made available online at *http://www.cacr.math.uwaterloo.ca/hac/*. Schneier [Sch96] provides a thorough overview of different cryptographic algorithms, with [FS03] as perhaps a better introduction. Kahn [Kah67] presents the fascinating history of cryptography from ancient times to 1967 and is particularly noteworthy in light of the secretive nature of the subject.

Expositions on the RSA algorithm [RSA78] include [CLRS01]. The RSA Laboratories home page *http://www.rsa.com/rsalabs/* is very informative.

Of course, the NSA (National Security Agency) is the place to go to learn the real state of the art in cryptography. The history of DES is well presented in [Sch96]. Particularly controversial was the decision by the NSA to limit key length to 56 bits.

MD5 [Riv92] is the hashing function used by PGP to compute digital signatures. Expositions include [Sch96, Sta06]. Serious problems with the security of MD5 have recently been exposed [WY05]. The SHA family of hash functions appears more secure, particularly SHA-256 and SHA-512.

Related Problems: Factoring and primality testing (see page 420), text compression (see page 637)).



INPUT OUTPUT

18.7 Finite State Machine Minimization

Input description: A deterministic finite automaton M.

Problem description: Create the smallest deterministic finite automaton M' such that M' behaves identically to M.

Discussion: Constructing and minimizing finite state machines arises repeatedly in software and hardware design applications. Finite state machines are very useful for specifying and recognizing patterns. Modern programming languages such as Java and Python provide built-in support for *regular expressions*, a particularly natural way of defining automata. Control systems and compilers often use finite state machines to encode the current state and possible associated actions/transitions. Minimizing the size of these automata reduces both the storage and execution costs of dealing with such machines.

Finite state machines are defined by directed graphs. Each vertex represents a state, and each character-labeled edge defines a transition from one state to another on receipt of the given alphabet symbol. The automata shown analyzes a sequence of coin tosses, with dark states signifying that an even number of heads have been observed. Such automata can be represented using any graph data structure (see Section 12.4 (page 381)), or by an $n \times |\Sigma|$ *transition matrix* where $|\Sigma|$ is the size of the alphabet.

Finite state machines are often used to specify search patterns in the guise of regular expressions, which are patterns formed by and-ing, or-ing, and looping

over smaller regular expressions. For example, the regular expression $a(a+b+c)^*a$ matches any string on (a, b, c) that begins and ends with distinct as. The best way to test whether a string s is recognized by a given regular expression R constructs the finite automaton equivalent to R, and then simulates this machine on S. See Section 18.3 (page 628) for alternative approaches to string matching.

We consider three different problems on finite automata:

- *Minimizing deterministic finite state machines* – Transition matrices for finite automata become prohibitively large for sophisticated machines, thus fueling the need for tighter encodings. The most direct approach is to eliminate redundant states in the automaton. As the example above illustrates, automata of widely varying sizes can compute the same function.

  Algorithms for minimizing the number of states in a deterministic finite automaton (DFA) appear in any book on automata theory. The basic approach partitions the states into gross equivalence classes and then refines the partition. Initially, the states are partitioned into accepting, rejecting, and other classes. The transitions from each node now branch to a given class on a given symbol. Whenever two states s, t from the same class C branch to elements of different classes, the class C must be partitioned into two subclasses, one containing s, the other containing t.

  This algorithm makes a sweep though all the classes looking for a new partition, and repeats the process from scratch if it finds one. This yields an $O(n^2)$ algorithm, since at most $n-1$ sweeps need ever be performed. The final equivalence classes correspond to the states in the minimum automaton. In fact, a more efficient $O(n \log n)$ algorithm is known. Implementations are cited below.

- *Constructing deterministic machines from nondeterministic machines* – DFAs are simple to work with, because the machine is always in exactly one state at any given time. *Nondeterministic automata* (NFAs) can be in multiple states at a time, so their current "state" represents a subset of all possible machine states.

  In fact, any NFA can be mechanically converted to an equivalent DFA, which can then be minimized as above. However, converting an NFA to a DFA might cause an exponential blowup in the number of states, which perversely might then be eliminated when minimizing the DFA. This exponential blowup makes most NFA minimization problems PSPACE-hard, which is even worse than NP-complete.

  The proofs of equivalence between NFAs, DFAs, and regular expressions are elementary enough to be covered in undergraduate automata theory classes. However, they are surprisingly nasty to actually code. Implementations are discussed below.

- *Constructing machines from regular expressions* – There are two approaches for translating a regular expression to an equivalent finite automaton. The difference is whether the output automaton will be a nondeterministic or deterministic machine. NFAs are easier to construct but less efficient to simulate.

  The nondeterministic construction uses ϵ-moves, which are optional transitions that require no input to fire. On reaching a state with an ϵ-move, we must assume that the machine can be in either state. Using ϵ-moves, it is straightforward to construct an automaton from a depth-first traversal of the parse tree of the regular expression. This machine will have $O(m)$ states, if m is the length of the regular expression. Furthermore, simulating this machine on a string of length n takes $O(mn)$ time, since we need consider each state/prefix pair only once.

  The deterministic construction starts with the parse tree for the regular expression, observing that each leaf represents an alphabet symbol in the pattern. After recognizing a prefix of the text, we can be left in some subset of these possible positions, which would correspond to a state in the finite automaton. The *derivatives* method builds up this automaton state by state as it is needed. Even so, some regular expressions of length m require $O(2^m)$ states in any DFA implementing them, such as $(a+b)^*a(a+b)(a+b)\ldots(a+b)$. There is no way to avoid this exponential space blowup. Fortunately it takes linear time to simulate an input string on any DFA, regardless of the size of the automaton.

Implementations: *Grail+* is a C++ package for symbolic computation with finite automata and regular expressions. Grail enables one to convert between different machine representations and to minimize automata. It can handle large machines defined on large alphabets. All code and documentation are accessible from *http://www.csd.uwo.ca/Research/grail*, as well as pointers to a variety of other automaton packages. Commercial use of Grail is not allowed without approval, although it is freely available to students and educators.

The AT&T Finite State Machine Library (FSM) is a set of general-purpose UNIX software tools for building, combining, optimizing, and searching weighted finite-state acceptors and transducers. It supports automata with more than ten million states and transitions. See *http://www.research.att.com/~fsmtools/fsm/*.

JFLAP (Java Formal Languages and Automata Package) is a package of graphical tools for learning the basic concepts of automata theory. Included are functions to convert between DFAs, NFAs, and regular expressions, and minimize the resulting automata. High-level automata are also supported, including context-free languages and Turing machines. *JFLAP* is available at *http://www.jflap.org/*. A related book [RF06] is also available.

FIRE Engine provides production-quality implementations of finite automata and regular expression algorithms. Several finite automaton

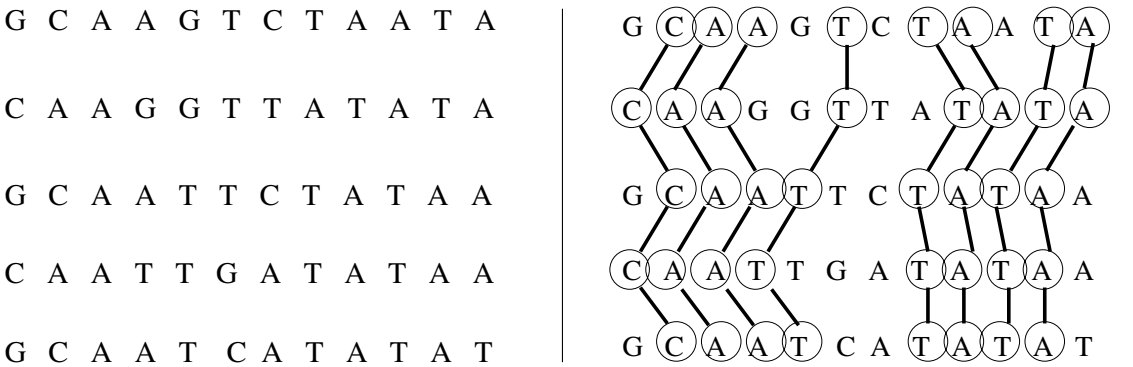
minimization algorithms have been implemented, including Hopcroft's $O(n \lg n)$ algorithm. Both deterministic and nondeterministic automata are supported. It is available at *http://www.fastar.org/* and, with certain enhancements, at *www.eti.pg.gda.pl/∼jandac/minim.html*.

Notes: Aho [Aho90] provides a good survey on algorithms for pattern matching, with a particularly clear exposition for the case where the patterns are regular expressions. The technique for regular expression pattern matching with ϵ-moves is due to Thompson [Tho68]. Other expositions on finite automaton pattern matching include [AHU74]. Expositions on finite automata and the theory of computation include [HMU06, Sip05]

The major annual meeting of interest in this field is the *Conference on Implementations and Applications of Automata* (CIAA). Pointers to current and previous meetings with associated software are available at *http://tln.li.univ-tours.fr/ciaa/*.

Hopcroft [Hop71] gave an optimal $O(n \lg n)$ algorithm for minimizing the number of states in DFAs. The derivatives method of constructing a finite state machine from a regular expression is due to Brzozowski [Brz64] and has been expanded upon in [BS86]. Expositions on the derivatives method includes Conway [Con71]. Recent work on incremental construction and optimization of automata includes [Wat03]. The problems of compressing a DFA to a minimum NFA [JR93] and testing the equivalence of two nondeterministic finite state machines [SM73] are both PSPACE-complete.

Related Problems: Satisfiability (see page 472). string matching (see page 628).



INPUT OUTPUT

18.8 Longest Common Substring/Subsequence

Input description: A set S of strings $S_1, \ldots, S_n$.

Problem description: What is the longest string S' such that all the characters of S' appear as a substring or subsequence of each S_i, $1 \leq i \leq n$?

Discussion: The problem of longest common substring/subsequence arises whenever we search for similarities across multiple texts. A particularly important application is finding a consensus among biological sequences. The genes for building proteins evolve with time, but the functional regions must remain consistent in order for them to work correctly. The longest common subsequence of the same gene in different species provides insight into what has been conserved over time.

The longest common subsequence problem for two strings is a special case of edit distance (see Section 18.4 (page 631)), when substitutions are forbidden and exact character match, insert, and delete are the only allowable edit operations. Under these conditions, the edit distance between P and T is $n + m - 2|lcs(P, T)|$, since we can delete the missing characters from P to the $lcs(P, T)$ and insert the missing characters from T to transform P to T.

Issues arising include

- *Are you looking for a common substring?* – In detecting plagiarism, we might need to find the longest phrase shared between two or more documents. Since phrases are strings of consecutive characters, here we need the longest common *substring* between the texts.

  The longest common substring of a set of strings can be identified in linear time using suffix trees, as discussed in Section 12.3 (page 377). The trick is to build a suffix tree containing all the strings, label each leaf with the input

string it represents, and then do a depth-first traversal to identify the deepest node with descendants from each input string.

- *Are you looking for a common scattered subsequence?* – For the rest of our discussion here, we restrict attention to finding common scattered subsequences. This algorithm is a special case of the dynamic program edit-distance computation. Indeed, an implementation in C is given on page 288.

  Let $M[i,j]$ denote the number of characters in the longest common substring of $S[1],\ldots,S[i]$ and $T[1],\ldots,T[j]$. When $S[i] \neq T[j]$, there is no way the last pair of characters could match, so $M[i,j] = \max(M[i,j-1], M[i-1,j])$. But if $S[i] = T[j]$, we have the option to select this character for our substring, so $M[i,j] = \max(M[i-1,j-1]+1, M[i-1,j], M[i,j-1])$.

  This recurrence computes the length of the longest common subsequence in $O(nm)$ time. We can reconstruct the actual common substring by walking backward from $M[n,m]$ and establishing which characters were matched along the way.

- *What if there are relatively few sets of matching characters?* – There is a faster algorithm for strings that do not contain too many copies of the same character. Let r be the number of pairs of positions (i,j) such that $S_i = T_j$. Thus, r can be as large as mn if both strings consist entirely of the same character, but $r = n$ if both strings are permutations of $\{1,\ldots,n\}$. This technique treats the pairs of r as defining points in the plane.

  The complete set of r such points can be found in $O(n+m+r)$ time using bucketing techniques. We create a bucket for each alphabet symbol c and each string (S or T), then partition the positions of each character of the string into the appropriate bucket. We then create a point (s,t) from every pair $s \in S_c$ and $t \in T_c$ in the buckets S_c and T_c.

  A common subsequence describes a monotonically nondecreasing path through these points, meaning the path only moves up and to the right. The longest such path can be found in $O((n+r)\lg n)$ time. We sort the points in order of increasing x-coordinate, breaking ties in favor of increasing y-coordinate. We insert points one by one in this order, and maintain the minimum terminal y-coordinate of any path going through exactly k points for each k, for $1 \leq k \leq n$. The new point (p_x, p_y) changes exactly one of these paths, either identifying a new longest subsequence or reducing the y-coordinate of the shortest path whose endpoint lies above p_y.

- *What if the strings are permutations?* – Permutations are strings without repeating characters. Two permutations define n pairs of matching characters, and so the above algorithm runs in $O(n \lg n)$ time. A particularly important case occurs in finding the longest *increasing* subsequence of a numerical sequence. Sorting the sequence and then replacing each number by

its rank defines a permutation p. The longest common subsequence of p and $\{1, 2, 3, \ldots, n\}$ gives the longest increasing subsequence.

- *What if we have more than two strings to align?* – The basic dynamic programming algorithm can be generalized to k strings, taking $O(2^k n^k)$ time, where n is the length of the longest string. This algorithm is exponential in the number of strings k, and so it will likely be too expensive for more than a few strings. Furthermore, the problem is NP-complete, so no better exact algorithm is destined to come along soon.

  Many heuristics have been proposed for multiple sequence alignment. They often start by computing the pairwise alignment for each of the $\binom{k}{2}$ pairs of strings. One approach then replaces the two most similar sequences with a single merged sequence, and repeats until all these alignments have been merged into one. The catch is that two strings often have many different alignments of optimal cost. The "right" alignment to pick depends upon the remaining sequences to merge, and is hence unknowable to the heuristic.

Implementations: Several programs are available for multiple sequence alignment of DNA/protein sequence data. *ClustalW* [THG94] is a popular and well-regarded program for multiple alignment of protein sequences. It is available at *http://www.ebi.ac.uk/Tools/clustalw/*. Another respectable option is the *MSA* package for multiple sequence alignment [GKS95], which is available at *http://www.ncbi.nlm.nih.gov/CBBresearch/Schaffer/msa.html*.

Any of the dynamic programming-based approximate string matching programs of Section 18.4 (page 631) can be used to find the longest common subsequence of two strings. More specialized implementations in Perl, Java, and C are available at *http://www.bioalgorithms.info/downloads/code/*.

Combinatorica [PS03] provides a Mathematica implementation of an algorithm to construct the longest increasing subsequence of a permutation, which is a special case of longest common subsequence. This algorithm is based on Young tableaux rather than dynamic programming. See Section 19.1.9 (page 661).

Notes: Surveys of algorithmic results on longest common subsequence (LCS) problems include [BHR00, GBY91]. The algorithm for the case where all the characters in each sequence are distinct or infrequent is due to Hunt and Szymanski [HS77]. Expositions of this algorithm include [Aho90, Man89]. There has been a surprising amount of recent work on this problem, including efficient bit-parallel algorithms for LCS [CIPR01]. Masek and Paterson [MP80] solve longest common subsequence in $O(mn/\log(\min\{m, n\}))$ for constant-sized alphabets, using the four Russians technique.

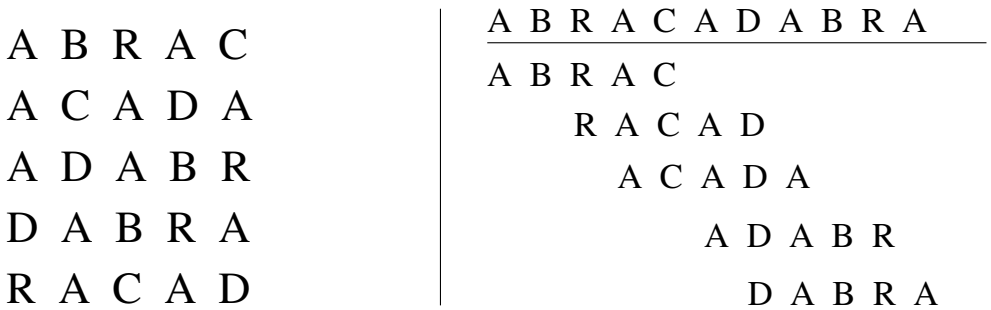
Construct two random n-character strings on an alphabet of size α. What is the expected length of their LCS? This problem has been extensively studied, with an excellent survey by Dancik [Dan94].

Multiple sequence alignment for computational biology is large field, with the books of Gusfield [Gus97] and Durbin [DEKM98] serving as excellent introductions. See [Not02]

for a more recent survey. The hardness of multiple sequence alignment follows from that of shortest common subsequence for large sets of strings [Mai78].

We motivated the problem of longest common substring with the application of plagiarism detection. See [SWA03] for the interesting details of how to implement a plagiarism detector for computer programs.

Related Problems: Approximate string matching (see page 631), shortest common superstring (see page 654).



INPUT OUTPUT

18.9 Shortest Common Superstring

Input description: A set of strings $S = \{S_1, \ldots, S_m\}$.

Problem description: Find the shortest string S' that contains each string S_i as a substring of S'.

Discussion: Shortest common superstring arises in a variety of applications. A casino gambling addict once asked me how to reconstruct the pattern of symbols on the wheels of a slot machine. On every spin, each wheel turns to a random position, displaying the selected symbol as well as the symbols immediately before/after it. Given enough observations of the slot machine, the symbol order for each wheel can be determined as the shortest common (circular) superstring of the observed symbol triples.

Another application of shortest common superstring is data/matrix compression. Suppose we are given a sparse $n \times m$ matrix M, meaning that most elements are zero. We can partition each row into m/k runs of k elements, and construct the shortest common superstring S' of all these runs. We can now represent the matrix by this superstring plus an $n \times m/k$ array of pointers denoting where each of these runs starts in S'. Any particular element $M[i, j]$ can still be accessed in constant time, but there will be substantial space savings when $|S| << mn$.

Perhaps the most compelling application is in DNA sequence assembly. Machines readily sequence fragments of about 500 base pairs or characters of DNA, but the real interest is in sequencing large molecules. Large-scale "shotgun" sequencing clones many copies of the target molecule, breaks them randomly into fragments, sequences the fragments, and then proposes the shortest superstring of the fragments as the correct sequence.

Finding a superstring of a set of strings is not difficult, since we can simply concatenate them together. Finding the *shortest* such string is what's problematic.

Indeed, shortest common superstring is NP-complete for all reasonable classes of strings.

Finding the shortest common superstring can easily be reduced to the traveling salesman problem (see Section 16.4 (page 533)). Create an overlap graph G where vertex v_i represents string S_i. Assign edge (v_i, v_j) weight equal to the length of S_i minus the overlap of S_j with S_i. Thus, $w(v_i, v_j) = 1$ for $S_i = abc$ and $S_j = bcd$. The minimum weight path visiting all the vertices defines the shortest common superstring. These edge weights are not symmetric; note that $w(v_j, v_i) = 3$ for the example above. Unfortunately, asymmetric TSP problems are much harder to solve in practice than symmetric instances.

The greedy heuristic provides the standard approach to approximating shortest common superstring. Identify which pair of strings have the maximum overlap. Replace them by the merged string, and repeat until only one string remains. This heuristic can actually be implemented in linear time. The seemingly most time-consuming part is in building the overlap graph. The brute-force approach to finding the maximum overlap of two length-l strings takes $O(l^2)$ for each of $O(n^2)$ string pairs. However, faster times are possible by using suffix trees (see Section 12.3 (page 377)). Build a tree containing all suffixes of all strings of S. String S_i overlaps with S_j iff a suffix of S_i matches the prefix of S_j—an event defined by a vertex of the suffix tree. Traversing these vertices in order of distance from the root defines the appropriate merging order.

How well does the greedy heuristic perform? It can certainly be fooled into creating a superstring that is twice as long as optimal. The optimal merging order for strings $c(ab)^k$, $(ba)^k$, and $(ab)^k c$ is left to right. But greedy starts by merging the first and third string, leaving the middle one no overlap possibility. The greedy superstring can never be worse than 3.5 times optimal, and usually will be a lot better in practice.

Building superstrings becomes more difficult when given both positive and negative strings, where each of the negative strings are forbidden to be a substring of the final result. The problem of deciding whether *any* such consistent substring exists is NP-complete, unless you are allowed to add an extra character to the alphabet to use as a spacer.

Implementations: Several high-performance programs for DNA sequence assembly are available. Such programs correct for sequencing errors, so the final result is not necessarily a superstring of the input reads. At the very least, they will serve as excellent models if you really need a short proper superstring.

CAP3 (Contig Assembly Program) [HM99] and *PCAP* [HWA+03] are the latest in a series of assemblers by Xiaoqiu Huang and his collaborators, which are available from *http://seq.cs.iastate.edu/*. They have been used on mammalian scale assembly projects involving hundreds of millions of bases.

The Celera assembler that originally sequenced the human genome is now available as open source. See *http://sourceforge.net/projects/wgs-assembler/*.

Notes: The shortest common superstring (SCS) problem and its application to DNA shotgun assembly are ably surveyed in [MKT07, Mye99a]. Kececioglu and Myers [KM95] report on an algorithm for this more general version of shortest common superstring, where the strings are assumed to have character substitution errors. Their paper is recommended reading to anyone interested in fragment assembly.

Blum et al. [BJL+94] gave the first constant-factor approximation algorithms for shortest common superstring, using a variation of the greedy heuristic. More recent research has beaten this constant down to 2.5 [Swe99], progress towards the expected factor-two result. The best approximation ratio so far proven for the standard greedy heuristic is 3.5 [KS05a]. Fast implementations of such heuristics are described in [Gus94].

Experiments on shortest common superstring heuristics are reported in [RBT04], which suggest that greedy heuristics typically produce solutions within 1.4% of optimal for a reasonable class of inputs. Experiments with genetic algorithm approaches are reported in [ZS04]. Analytical results [YZ99] demonstrate very little compression on the SCS of random sequences largely because the expected overlap length of any two random strings is small.

Related Problems: Suffix trees (see page 377), text compression (see page 637).

19

Algorithmic Resources

This chapter briefly describes resources that the practical algorithm designer should be familiar with. Although some of this information has appeared elsewhere in the catalog, the most important pointers are collected here for general reference.

19.1 Software Systems

In this section, we describe several particularly comprehensive implementations of combinatorial algorithms, all of which are downloadable over the Internet. Although these codes are mentioned in the relevant sections of the catalog, they are substantial enough to warrant further attention.

A good algorithm designer does not reinvent the wheel, and a good programmer does not rewrite code that other people have written. Picasso put it best: "Good artists borrow. Great artists steal."

However, here is a word of caution about stealing. Many of the codes described in this book have been made available for research or educational use only. Commercial use may require a licensing arrangement with the author. I urge you to respect this. Licensing terms from academic institutions are usually quite modest. The recognition that industry is using a particular code is important to the authors, often more important than the money involved. This can lead to enhanced support or future releases of the software. Do the right thing and get a license. Information about terms or whom to contact is usually available embedded within the documentation, or available at the source's website.

Although many of the systems we describe here may be available by accessing our algorithm repository, *http://www.cs.sunysb.edu/∼algorith* we encourage you to get them from the original sites instead of Stony Brook. There are three reasons. First, the version on the original site is *much* more likely to be up-to-date. Second,

S.S. Skiena, *The Algorithm Design Manual*, 2nd ed., DOI: 10.1007/978-1-84800-070-4_19,

there are often supporting files and documentation that we did not download that may be of interest to you. Finally, many authors monitor the downloads of their codes, and so you deny them a well-earned thrill if you don't take them from the homepage.

19.1.1 LEDA

LEDA, for *Library of Efficient Data types and Algorithms*, is perhaps the best single resource available to support combinatorial computing. *LEDA* was originally developed by a group at Max-Planck-Instutut in Saarbrücken, Germany, including Kurt Mehlhorn, Stefan Näher, Stefan Schirra, Christian Uhrig, and Christoph Burnikel. *LEDA* is unique because of (1) the algorithmic sophistication of its developers, and (2) the level of continuity and resources invested in the project.

What *LEDA* offers is a complete collection of well-implemented C++ data structures and types. Particularly useful is the graph type, which supports all the basic operations one needs in an intelligent way, although this generality comes at some cost in size and speed over handcrafted implementations. A useful library of graph algorithms is included, which illustrates how cleanly and concisely these algorithms can be implemented using the LEDA data types. Good implementations of the most important data structures supporting such common data types as dictionaries and priority queues are provided. There are also algorithms and data structures for computational geometry, including support for visualization. For more, see the book [MN99].

Since 2001, *LEDA* has been exclusively available from Algorithmic Solutions Software GmbH (*http://www.algorithmic-solutions.com/*). This ensures professional-quality support, and new releases appear often. The great news is that a free edition containing all the basic data structures (including dictionaries, priority queues, graphs, and numerical types) was released February 2008. No source code or advanced algorithms are provided with the free edition. However, the licencing fees for the full library are not outlandish and free trial downloads are available.

19.1.2 CGAL

The *Computational Geometry Algorithms Library* or *CGAL* provides efficient and reliable geometric algorithms in C++. It is extremely comprehensive, offering a rich variety of triangulations, Voronoi diagrams, operations on polygons and polyhedra, line/curve arrangements, alpha-shapes, convex-hull algorithms, geometric search structures, and more. Many work in three dimensions and some beyond.

CGAL (*www.cgal.org*) should be the first place to go for serious geometric computing, although you should expect to spend some start-up time getting oriented to the *CGAL* way of thinking. *CGAL* is distributed under a dual-license scheme. It can be used together with open source software free of charge but using *CGAL* in other contexts requires obtaining a commercial license.

19.1.3 Boost Graph Library

Boost (*www.boost.org*) provides a well-regarded collection of free peer-reviewed portable C++ source libraries. The Boost license encourages both commercial and noncommercial use.

The Boost Graph Library [SLL02] (*http://www.boost.org/libs/graph/doc*) is perhaps most relevant for the readers of this book. Implementations of adjacency lists, matrices, and edge lists are included, along with a reasonable library of basic graph algorithms. Its interface and components are generic in the same sense as the C++ Standard Template Library (STL). Other Boost libraries of interest include those for string/text processing and math/numeric computation.

19.1.4 GOBLIN

The *Graph Object Library for Network Programming Problems* (GOBLIN) is a C++ class library that broadly focuses on graph optimization problems. These include several varieties of shortest path, minimum spanning tree, and connected component algorithms, plus particularly strong coverage of network flows and matching. Finally, a generic branch-and-bound module is used to solve such hard problems as independent set and vertex coloring.

GOBLIN was written and is maintained by Christian Fremuth-Paeger at the University of Augsberg. It is available under GNU lesser public licence from *http://www.math.uni-augsburg.de/~fremuth/goblin.html*. A spiffy Tcl/Tk interface is provided. GOBLIN is presumably not as robust as *Boost* or *LEDA*, but contains several algorithms not present in either of them.

19.1.5 Netlib

Netlib (*www.netlib.org*) is an online repository of mathematical software that contains a large number of interesting codes, tables, and papers. Netlib is a compilation of resources from a variety of places, with fairly detailed indices and search mechanisms. Netlib is important because of its breadth and ease of access. Whenever you need a specialized piece of mathematical software, you should look here first.

The Guide to Available Mathematical Software (GAMS), is an indexing service for Netlib and other related software repositories that can help you find what you want. Check it out at *http://gams.nist.gov*. GAMS is a service of the National Institute of Standards and Technology (NIST).

19.1.6 Collected Algorithms of the ACM

An early mechanism for the distribution of useful algorithm implementations was the *Collected Algorithms of the ACM* (*CALGO*). It first appeared in *Communications of the ACM* in 1960, covering such famous algorithms as Floyd's linear-time build heap algorithm. More recently, it has been the province of the *ACM Transactions on Mathematical Software*. Each algorithm/implementation is described

in a brief journal article, with the implementation validated and collected. These implementations are maintained at *http://www.acm.org/calgo/* and at Netlib.

Over 850 algorithms have appeared to date. Most of the codes are in Fortran and are relevant to numerical computing, although several interesting combinatorial algorithms have slithered their way into CALGO. Since the implementations have been refereed, they are presumably more reliable than most comparable software.

19.1.7 SourceForge and CPAN

SourceForge (*http://sourceforge.net/*) is the largest open source software development website, with over 160,000 registered projects. Most are of highly limited interest, but there is a lot of good stuff to be found. These include graph libraries such as *JUNG* and *JGraphT*, optimization engines such as *lpsolve* and *JGAP*, and much more.

CPAN (*http://www.cpan.org/*) is the Comprehensive Perl Archive Network. This enormous collection of Perl modules and scripts is where you should look before trying to implement anything in Perl.

19.1.8 The Stanford GraphBase

The Stanford GraphBase is an interesting program for several reasons. First, it was composed as a "literate program," meaning that it was written to be read. If anybody's programs deserve to be read, it is Knuth's, and [Knu94] contains the full source code of the system. The programming language/environment is CWEB, which permits the mixing of text and code in particularly expressive ways.

The GraphBase contains implementations of several important combinatorial algorithms, including matching, minimum spanning trees, and Voronoi diagrams, as well as specialized topics like constructing expander graphs and generating combinatorial objects. Finally, it contains programs for several recreational problems, including constructing word ladders (flour-floor-flood-blood-brood-broad-bread) and establishing dominance relations among football teams. Check it out at *http://www-cs-faculty.stanford.edu/~knuth/sgb.html*.

Although the GraphBase is fun to play with, it is not really suited for building general applications on top of. The GraphBase is perhaps most useful as an instance generator for constructing a wide variety of graphs to serve as test data. It incorporates graphs derived from interactions of characters in famous novels, Roget's thesaurus, the Mona Lisa, and the economy of the United States. Furthermore, because of its machine-independent random number generators, the GraphBase provides a way to construct random graphs that can be reconstructed elsewhere, making them perfect for experimental comparisons of algorithms.

19.1.9 Combinatorica

Combinatorica [PS03] is a collection of over 450 algorithms for combinatorics and graph theory written in Mathematica. These routines have been designed to work together, enabling one to readily experiment with discrete structures. *Combinatorica* has been widely used for both research and education.

Although (in my totally unbiased opinion) *Combinatorica* is more comprehensive and better integrated than other libraries of combinatorial algorithms, it is also the slowest such system available. Credit for all of these properties is largely due to Mathematica, which provides a very high-level, functional, interpreted, and thus inefficient programming language. *Combinatorica* is best for finding quick solutions to small problems, and (if you can read Mathematica code) as a terse exposition of algorithms for translation into other languages.

Check out *http://www.combinatorica.com* for the latest release and associated resources. It is also included with the standard Mathematica distribution in the directory *Packages/DiscreteMath/Combinatorica.m*.

19.1.10 Programs from Books

Several books on algorithms include working implementations of the algorithms in a real programming language. Although these implementations are intended primarily for exposition, they can also be useful for computation. Since they are typically small and clean, they can prove the right foundation for simple applications.

The most useful codes of this genre are described below. Most are available from the algorithm repository, *http://www.cs.sunysb.edu/∼algorith*.

Programming Challenges

If you like the C code that appeared in the first half of the text, you should check out the programs I wrote for my book *Programming Challenges* [SR03]. Perhaps most useful are additional examples of dynamic programming, computational geometry routines like convex hull, and a bignum integer arithmetic package. This algorithm library is available at *http://www.cs.sunysb.edu/∼skiena/392/programs/* or at *http://www.programming-challenges.com*.

Combinatorial Algorithms for Computers and Calculators

Nijenhuis and Wilf [NW78] specializes in algorithms for constructing basic combinatorial objects such as permutations, subsets, and partitions. Such algorithms are often very short, but they are hard to locate and usually surprisingly subtle. Fortran routines for all of the algorithms are provided, as well as a discussion of the theory behind each of them. The programs are usually short enough that it is reasonable to translate them directly into a more modern programming language, as I did in writing *Combinatorica* (see Section 19.1.9). Both random and sequential

generation algorithms are provided. Descriptions of more recent algorithms for several problems, without code, are provided in [Wil89].

These programs are now available from our algorithm repository website. We tracked them down from Neil Sloane, who had them on a magnetic tape, while the original authors did not! In [NW78], Nijenhuis and Wilf set the proper standard of statistically testing the output distribution of each of the random generators to establish that they really appear uniform. We encourage you to do the same before using these programs to verify that nothing has been lost in transit.

Computational Geometry in C

O'Rourke [O'R01] is perhaps the best practical introduction to computational geometry, because of its careful and correct C language implementations of all the main algorithms of computational geometry. Fundamental geometric primitives, convex hulls, triangulations, Voronoi diagrams, and motion planning are all included. Although they were implemented primarily for exposition rather than production use, they should be quite reliable. The codes are available from *http://maven.smith.edu/∼orourke/code.html*.

Algorithms in C++

Sedgewick's popular algorithms text [Sed98, SS02] comes in several different language editions, including C, C++, and Java. This book distinguishes itself through its use of algorithm animation and in its broad topic coverage, including numerical, string, and geometric algorithms.

The language-specific parts of the text consist of many small code fragments, instead of full programs or subroutines. They are best thought of as models, instead of working implementations. Still, the program fragments from the C++ edition have been made available from *http://www.cs.princeton.edu/∼rs/*.

Discrete Optimization Algorithms in Pascal

This is a collection of 28 programs for solving discrete optimization problems, appearing in the book by Syslo, Deo, and Kowalik [SDK83]. The package includes programs for integer and linear programming, the knapsack and set cover problems, traveling salesman, vertex coloring, and scheduling, as well as standard network optimization problems. They have been made available from the algorithm repository at *http://www.cs.sunysb.edu/∼algorith*.

This package is noteworthy for the operations-research flavor of the problems and algorithms selected. The algorithms have been selected to solve problems, rather than purely expository purposes.

19.2 Data Sources

It is often important to have interesting data to feed your algorithms, to serve as test data to ensure correctness or to compare different algorithms for raw speed. Finding good test data can be surprisingly difficult. Here are some pointers:

- *TSPLIB* – This well-respected library of test instances for the traveling salesman problem [Rei91] provides the standard collection of hard instances of TSPs. TSPLIB instances are large, real-world graphs, derived from applications such as circuit boards and networks. TSPLIB is available from *http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/* Somewhat older instances are also available from Netlib.
- *Stanford GraphBase* – Discussed in Section 19.1.8, this suite of programs by Knuth provides portable generators for a wide variety of graphs. These include graphs arising from distance matrices, arts, and literature, as well as graphs of more theoretical interest.
- *DIMACS Challenge data* – A series of DIMACS Challenge workshops have focused on evaluating algorithm implementations of graph, logic, and data structure problems. Instance generators for each problem have been developed, with the focus on constructing difficult or representative test data. The products of the DIMACS Challenges are all available from *http://dimacs.rutgers.edu/Challenges*.

19.3 Online Bibliographic Resources

The Internet has proven to be a fantastic resource for people interested in algorithms, as it has for many other subjects. What follows is a highly selective list of the resources that I use most often. All should be in the tool chest of every algorist.

- *ACM Digital Library* – This collection of bibliographic references provides links to essentially every technical paper ever published in computer science. Check out what is available at *http://portal.acm.org/*.
- *Google Scholar* – This free resource (*http://scholar.google.com/*) restricts Web searches to things that look like academic papers, often making it a sounder search for serious information than a general Web search. Particularly useful is the ability to see which papers cite a given paper. This lets you update an old reference to see what has happened since publication, and helps to judge the significance of a particular article.
- *Amazon.com* – This comprehensive catalog of books (*www.amazon.com*) is surprisingly useful for finding relevant literature for algorithmic problems, particularly since many recent books have been digitized and placed in it's index.

19.4 Professional Consulting Services

Algorist Technologies (*http://www.algorist.com*) is a consulting firm that provides its clients with short-term, expert help in algorithm design and implementation. Typically, an Algorist consultant is called in for one to three days worth of intensive, onsite discussion and analysis with the client's own development staff. Algorist has built an impressive record of performance improvements with several companies and applications. We provide longer-term consulting and contracting services as well.

Call 212-222-9891 or email info@algorist.com for more information on services provided by Algorist Technologies.

Algorist Technologies
215 West 92nd St. Suite 1F
New York, NY 10025
http://www.algorist.com

Bibliography

[AAAG95] O. Aichholzer, F. Aurenhammer, D. Alberts, and B. Gartner. A novel type of skeleton for polygons. *J. Universal Computer Science*, 1:752–761, 1995.

[ABCC07] D. Applegate, R. Bixby, V. Chvatal, and W. Cook. *The Traveling Salesman Problem: A computational study*. Princeton University Press, 2007.

[Abd80] N. N. Abdelmalek. A Fortran subroutine for the L_1 solution of overdetermined systems of linear equations. *ACM Trans. Math. Softw.*, 6(2):228–230, June 1980.

[ABF05] L. Arge, G. Brodal, and R. Fagerberg. Cache-oblivious data structures. In D. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, pages 34:1–34:27. Chapman and Hall / CRC, 2005.

[AC75] A. Aho and M. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18:333–340, 1975.

[AC91] D. Applegate and W. Cook. A computational study of the job-shop scheduling problem. *ORSA Journal on Computing*, 3:149–156, 1991.

[ACG+03] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, S. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Springer, 2003.

[ACH+91] E. M. Arkin, L. P. Chew, D. P. Huttenlocher, K. Kedem, and J. S. B. Mitchell. An efficiently computable metric for comparing polygonal shapes. *IEEE Trans. PAMI*, 13(3):209–216, 1991.

[ACI92] D. Alberts, G. Cattaneo, and G. Italiano. An empirical study of dynamic graph algorithms. In *Proc. Seventh ACM-SIAM Symp. Discrete Algorithms (SODA)*, pages 192–201, 1992.

[ACK01a] N. Amenta, S. Choi, and R. Kolluri. The power crust. In *Proc. 6th ACM Symp. on Solid Modeling*, pages 249–260, 2001.

[ACK01b] N. Amenta, S. Choi, and R. Kolluri. The power crust, unions of balls, and the medial axis transform. *Computational Geometry: Theory and Applications*, 19:127–153, 2001.

[ACP+07] H. Ahn, O. Cheong, C. Park, C. Shin, and A. Vigneron. Maximizing the overlap of two planar convex sets under rigid motions. *Computational Geometry: Theory and Applications*, 37:3–15, 2007.

[ADGM04] L. Aleksandrov, H. Djidjev, H. Guo, and A. Maheshwari. Partitioning planar graphs with costs and weights. In *Algorithm Engineering and Experiments: 4th International Workshop, ALENEX 2002*, 2004.

[ADKF70] V. Arlazarov, E. Dinic, M. Kronrod, and I. Faradzev. On economical construction of the transitive closure of a directed graph. *Soviet Mathematics, Doklady*, 11:1209–1210, 1970.

[Adl94] L. M. Adleman. Molecular computations of solutions to combinatorial problems. *Science*, 266:1021–1024, November 11, 1994.

[AE83] D. Avis and H. ElGindy. A combinatorial approach to polygon similarity. *IEEE Trans. Inform. Theory*, IT-2:148–150, 1983.

[AE04] G. Andrews and K. Eriksson. *Integer Partitions*. Cambridge Univ. Press, 2004.

[AF96] D. Avis and K. Fukuda. Reverse search for enumeration. *Disc. Applied Math.*, 65:21–46, 1996.

[AFH02] P. Agarwal, E. Flato, and D. Halperin. Polygon decomposition for efficient construction of Minkowski sums. *Computational Geometry: Theory and Applications*, 21:39–61, 2002.

[AG00] H. Alt and L. Guibas. Discrete geometric shapes: Matching, interpolation, and approximation. In J. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 121–153. Elsevier, 2000.

[Aga04] P. Agarwal. Range searching. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 809–837. CRC Press, 2004.

[AGSS89] A. Aggarwal, L. Guibas, J. Saxe, and P. Shor. A linear-time algorithm for computing the Voronoi diagram of a convex polygon. *Discrete and Computational Geometry*, 4:591–604, 1989.

[AGU72] A. Aho, M. Garey, and J. Ullman. The transitive reduction of a directed graph. *SIAM J. Computing*, 1:131–137, 1972.

[Aho90] A. Aho. Algorithms for finding patterns in strings. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Algorithms and Complexity*, volume A, pages 255–300. MIT Press, 1990.

[AHU74] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading MA, 1974.

[AHU83] A. Aho, J. Hopcroft, and J. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading MA, 1983.

[Aig88] M. Aigner. *Combinatorial Search*. Wiley-Teubner, 1988.

[AITT00] Y. Asahiro, K. Iwama, H. Tamaki, and T. Tokuyama. Greedily finding a dense subgraph. *J. Algorithms*, 34:203–221, 2000.

[AK89] E. Aarts and J. Korst. *Simulated annealing and Boltzman machines: A stochastic approach to combinatorial optimization and neural computing*. John Wiley and Sons, 1989.

[AKD83] J. H. Ahrens, K. D. Kohrt, and U. Dieter. Sampling from gamma and Poisson distributions. *ACM Trans. Math. Softw.*, 9(2):255–257, June 1983.

[AKS04] M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P. *Annals of Mathematics*, 160:781–793, 2004.

[AL97] E. Aarts and J. K. Lenstra. *Local Search in Combinatorial Optimization*. John Wiley and Sons, West Sussex, England, 1997.

[AM93] S. Arya and D. Mount. Approximate nearest neighbor queries in fixed dimensions. In *Proc. Fourth ACM-SIAM Symp. Discrete Algorithms (SODA)*, pages 271–280, 1993.

[AMN+98] S. Arya, D. Mount, N. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *J. ACM*, 45:891 – 923, 1998.

[AMO93] R. Ahuja, T. Magnanti, and J. Orlin. *Network Flows*. Prentice Hall, Englewood Cliffs NJ, 1993.

[AMWW88] H. Alt, K. Mehlhorn, H. Wagener, and E. Welzl. Congruence, similarity and symmetries of geometric objects. *Discrete Comput. Geom.*, 3:237–256, 1988.

[And98] G. Andrews. *The Theory of Partitions*. Cambridge Univ. Press, 1998.

[And05] A. Andersson. Searching and priority queues in $o(\log n)$ time. In D. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, pages 39:1–39:14. Chapman and Hall / CRC, 2005.

[AP72] A. Aho and T. Peterson. A minimum distance error-correcting parser for context-free languages. *SIAM J. Computing*, 1:305–312, 1972.

[APT79] B. Aspvall, M. Plass, and R. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Info. Proc. Letters*, 8:121–123, 1979.

[Aro98] S. Arora. Polynomial time approximations schemes for Euclidean TSP and other geometric problems. *J. ACM*, 45:753–782, 1998.

[AS00] P. Agarwal and M. Sharir. Arrangements. In J. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 49–119. Elsevier, 2000.

[Ata83] M. Atallah. A linear time algorithm for the Hausdorff distance between convex polygons. *Info. Proc. Letters*, 8:207–209, 1983.

[Ata84] M. Atallah. Checking similarity of planar figures. *Internat. J. Comput. Inform. Sci.*, 13:279–290, 1984.

[Ata98] M. Atallah. *Algorithms and Theory of Computation Handbook*. CRC, 1998.

[Aur91] F. Aurenhammer. Voronoi diagrams: a survey of a fundamental data structure. *ACM Computing Surveys*, 23:345–405, 1991.

[Bar03] A. Barabasi. *Linked: How Everything Is Connected to Everything Else and What It Means*. Plume, 2003.

[BBF99] V. Bafna, P. Berman, and T. Fujito. A 2-approximation algorithm for the undirected feedback vertex set problem. *SIAM J. Discrete Math.*, 12:289–297, 1999.

[BBPP99] I. Bomze, M. Budinich, P. Pardalos, and M. Pelillo. The maximum clique problem. In D.-Z. Du and P.M. Pardalos, editors, *Handbook of Combinatorial Optimization*, volume A sup., pages 1–74. Kluwer, 1999.

[BCGR92] D. Berque, R. Cecchini, M. Goldberg, and R. Rivenburgh. The SetPlayer system for symbolic computation on power sets. *J. Symbolic Computation*, 14:645–662, 1992.

[BCPB04] J. Boyer, P. Cortese, M. Patrignani, and G. Di Battista. Stop minding your p's and q's: Implementing a fast and simple DFS-based planarity testing and embedding algorithm. In *Proc. Graph Drawing (GD '03)*, volume 2912 LNCS, pages 25–36, 2004.

[BCW90] T. Bell, J. Cleary, and I. Witten. *Text Compression*. Prentice Hall, Englewood Cliffs NJ, 1990.

[BD99] R. Bubley and M. Dyer. Faster random generation of linear extensions. *Disc. Math.*, 201:81–88, 1999.

[BDH97] C. Barber, D. Dobkin, and H. Huhdanpaa. The Quickhull algorithm for convex hulls. *ACM Trans. on Mathematical Software*, 22:469–483, 1997.

[BDN01] G. Bilardi, P. D'Alberto, and A. Nicolau. Fractal matrix multiplication: a case study on portability of cache performance. In *Workshop on Algorithm Engineering (WAE)*, 2001.

[BDY06] K. Been, E. Daiches, and C. Yap. Dynamic map labeling. *IEEE Trans. Visualization and Computer Graphics*, 12:773–780, 2006.

[Bel58] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[Ben75] J. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18:509–517, 1975.

[Ben90] J. Bentley. *More Programming Pearls*. Addison-Wesley, Reading MA, 1990.

[Ben92a] J. Bentley. Fast algorithms for geometric traveling salesman problems. *ORSA J. Computing*, 4:387–411, 1992.

[Ben92b] J. Bentley. Software exploratorium: The trouble with qsort. *UNIX Review*, 10(2):85–93, February 1992.

[Ben99] J. Bentley. *Programming Pearls*. Addison-Wesley, Reading MA, second edition edition, 1999.

[Ber89] C. Berge. *Hypergraphs*. North-Holland, Amsterdam, 1989.

[Ber02] M. Bern. Adaptive mesh generation. In T. Barth and H. Deconinck, editors, *Error Estimation and Adaptive Discretization Methods in Computational Fluid Dynamics*, pages 1–56. Springer-Verlag, 2002.

[Ber04a] M. Bern. Triangulations and mesh generation. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 563–582. CRC Press, 2004.

[Ber04b] D. Bernstein. Fast multiplication and its applications. http://cr.yp.to/arith.html, 2004.

[BETT99] G. Di Battista, P. Eades, R. Tamassia, and I. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice-Hall, 1999.

[BF00] M. Bender and M. Farach. The LCA problem revisited. In *Proc. 4th Latin American Symp. on Theoretical Informatics*, pages 88–94. Springer-Verlag LNCS vol. 1776, 2000.

[BFP+72] M. Blum, R. Floyd, V. Pratt, R. Rivest, and R. Tarjan. Time bounds for selection. *J. Computer and System Sciences*, 7:448–461, 1972.

[BFV07] G. Brodal, R. Fagerberg, and K. Vinther. Engineering a cache-oblivious sorting algorithm. *ACM J. of Experimental Algorithmics*, 12, 2007.

[BG95] J. Berry and M. Goldberg. Path optimization and near-greedy analysis for graph partitioning: An empirical study. In *Proc. 6th ACM-SIAM Symposium on Discrete Algorithms*, pages 223–232, 1995.

[BGS95] M Bellare, O. Goldreich, and M. Sudan. Free bits, PCPs, and non-approximability – towards tight results. In *Proc. IEEE 36th Symp. Foundations of Computer Science*, pages 422–431, 1995.

[BH90] F. Buckley and F. Harary. *Distances in Graphs*. Addison-Wesley, Redwood City, Calif., 1990.

[BH01] G. Barequet and S. Har-Peled. Efficiently approximating the minimum-volume bounding box of a point set in three dimensions. *J. Algorithms*, 38:91–109, 2001.

[BHR00] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *Proc. String Processing and Information Retreival (SPIRE)*, pages 39–48, 2000.

[BIK+04] H. Bronnimann, J. Iacono, J. Katajainen, P. Morin, J. Morrison, and G. Toussaint. Space-efficient planar convex hull algorithms. *Theoretical Computer Science*, 321:25–40, 2004.

[BJL+94] A. Blum, T. Jiang, M. Li, J. Tromp, and M. Yanakakis. Linear approximation of shortest superstrings. *J. ACM*, 41:630–647, 1994.

[BJL06] C. Buchheim, M. Jünger, and S. Leipert. Drawing rooted trees in linear time. *Software: Practice and Experience*, 36:651–665, 2006.

[BJLM83] J. Bentley, D. Johnson, F. Leighton, and C. McGeoch. An experimental study of bin packing. In *Proc. 21st Allerton Conf. on Communication, Control, and Computing*, pages 51–60, 1983.

[BK04] Y. Boykov and V. Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE Trans. Pattern Analysis and Machine Intelligence (PAMI)*, 26:1124–1137, 2004.

[BKRV00] A. Blum, G. Konjevod, R. Ravi, and S. Vempala. Semi-definite relaxations for minimum bandwidth and other vertex-ordering problems. *Theoretical Computer Science*, 235:25–42, 2000.

[BL76] K. Booth and G. Lueker. Testing for the consecutive ones property, interval graphs, and planarity using PQ-tree algorithms. *J. Computer System Sciences*, 13:335–379, 1976.

[BL77] B. P. Buckles and M. Lybanon. Generation of a vector from the lexicographical index. *ACM Trans. Math. Softw.*, 3(2):180–182, June 1977.

[BLS91] D. Bailey, K. Lee, and H. Simon. Using Strassen's algorithm to accelerate the solution of linear systems. *J. Supercomputing*, 4:357–371, 1991.

[Blu67] H. Blum. A transformation for extracting new descriptions of shape. In W. Wathen-Dunn, editor, *Models for the Perception of Speech and Visual Form*, pages 362–380. MIT Press, 1967.

[BLW76] N. L. Biggs, E. K. Lloyd, and R. J. Wilson. *Graph Theory 1736-1936*. Clarendon Press, Oxford, 1976.

[BM53] G. Birkhoff and S. MacLane. *A survey of modern algebra*. Macmillian, New York, 1953.

[BM77] R. Boyer and J. Moore. A fast string-searching algorithm. *Communications of the ACM*, 20:762–772, 1977.

[BM89] J. Boreddy and R. N. Mukherjee. An algorithm to find polygon similarity. *Inform. Process. Lett.*, 33(4):205–206, 1989.

[BM01] E. Bingham and H. Mannila. Random projection in dimensionality reduction: applications to image and text data. In *Proc. ACM Conf. Knowledge Discovery and Data Mining (KDD)*, pages 245–250, 2001.

[BM05] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1:485–509, 2005.

[BO79] J. Bentley and T. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, C-28:643–647, 1979.

[BO83] M. Ben-Or. Lower bounds for algebraic computation trees. In *Proc. Fifteenth ACM Symp. on Theory of Computing*, pages 80–86, 1983.

[Bol01] B. Bollobas. *Random Graphs*. Cambridge Univ. Press, second edition, 2001.

[BP76] E. Balas and M. Padberg. Set partitioning – a survey. *SIAM Review*, 18:710–760, 1976.

[BR80] I. Barrodale and F. D. K. Roberts. Solution of the constrained L_1 linear approximation problem. *ACM Trans. Math. Softw.*, 6(2):231–235, June 1980.

[BR95] A. Binstock and J. Rex. *Practical Algorithms for Programmers*. Addison-Wesley, Reading MA, 1995.

[Bra99] R. Bracewell. *The Fourier Transform and its Applications*. McGraw-Hill, third edition, 1999.

[Bre73] R. Brent. *Algorithms for minimization without derivatives*. Prentice-Hall, Englewood Cliffs NJ, 1973.

[Bre74] R. P. Brent. A Gaussian pseudo-random number generator. *Comm. ACM*, 17(12):704–706, December 1974.

[Brè79] D. Brèlaz. New methods to color the vertices of a graph. *Comm. ACM*, 22:251–256, 1979.

[Bri88] E. Brigham. *The Fast Fourier Transform*. Prentice Hall, Englewood Cliffs NJ, facimile edition, 1988.

[Bro95] F. Brooks. *The Mythical Man-Month*. Addison-Wesley, Reading MA, 20th anniversary edition, 1995.

[Bru07] P. Brucker. *Scheduling Algorithms*. Springer-Verlag, fifth edition, 2007.

[Brz64] J. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11:481–494, 1964.

[BS76] J. Bentley and M. Shamos. Divide-and-conquer in higher-dimensional space. In *Proc. Eighth ACM Symp. Theory of Computing*, pages 220–230, 1976.

[BS86] G. Berry and R. Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48:117–126, 1986.

[BS96] E. Bach and J. Shallit. *Algorithmic Number Theory: Efficient Algorithms*, volume 1. MIT Press, Cambridge MA, 1996.

[BS97] R. Bradley and S. Skiena. Fabricating arrays of strings. In *Proc. First Int. Conf. Computational Molecular Biology (RECOMB '97)*, pages 57–66, 1997.

[BS07] A. Barvinok and A. Samorodnitsky. Random weighting, asymptotic counting and inverse isoperimetry. *Israel Journal of Mathematics*, 158:159–191, 2007.

[BT92] J. Buchanan and P. Turner. *Numerical methods and analysis*. McGraw-Hill, New York, 1992.

[Buc94] A. G. Buckley. A Fortran 90 code for unconstrained nonlinear minimization. *ACM Trans. Math. Softw.*, 20(3):354–372, September 1994.

[BvG99] S. Baase and A. van Gelder. *Computer Algorithms*. Addison-Wesley, Reading MA, third edition, 1999.

[BW91] G. Brightwell and P. Winkler. Counting linear extensions. *Order*, 3:225–242, 1991.

[BW94] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[BW00] R. Borndorfer and R. Weismantel. Set packing relaxations of some integer programs. *Math. Programming A*, 88:425–450, 2000.

[Can87] J. Canny. *The complexity of robot motion planning*. MIT Press, Cambridge MA, 1987.

[Cas95] G. Cash. A fast computer algorithm for finding the permanent of adjacency matrices. *J. Mathematical Chemistry*, 18:115–119, 1995.

[CB04] C. Cong and D. Bader. The Euler tour technique and parallel rooted spanning tree. In *Int. Conf. Parallel Processing (ICPP)*, pages 448–457, 2004.

[CC92] S. Carlsson and J. Chen. The complexity of heaps. In *Proc. Third ACM-SIAM Symp. on Discrete Algorithms*, pages 393–402, 1992.

[CC97] W. Cook and W. Cunningham. *Combinatorial Optimization*. Wiley, 1997.

[CC05] S. Chapra and R. Canale. *Numerical Methods for Engineers*. McGraw-Hill, fifth edition, 2005.

[CCDG82] P. Chinn, J. Chvátolvá, A. K. Dewdney, and N. E. Gibbs. The bandwidth problem for graphs and matrices – a survey. *J. Graph Theory*, 6:223–254, 1982.

[CCPS98] W. Cook, W. Cunningham, W. Pulleyblank, and A. Schrijver. *Combinatorial Optimization*. Wiley, 1998.

[CD85] B. Chazelle and D. Dobkin. Optimal convex decompositions. In G. Toussaint, editor, *Computational Geometry*, pages 63–133. North-Holland, Amsterdam, 1985.

[CDL86] B. Chazelle, R. Drysdale, and D. Lee. Computing the largest empty rectangle. *SIAM J. Computing*, 15:300–315, 1986.

[CDT95] G. Carpento, M. Dell'Amico, and P. Toth. CDT: A subroutine for the exact solution of large-scale, asymmetric traveling salesman problems. *ACM Trans. Math. Softw.*, 21(4):410–415, December 1995.

[CE92] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments. *J. ACM*, 39:1–54, 1992.

[CFC94] C. Cheng, B. Feiring, and T. Cheng. The cutting stock problem — a survey. *Int. J. Production Economics*, 36:291–305, 1994.

[CFR06] D. Coppersmith, L. Fleischer, and A. Rudrea. Ordering by weighted number of wins gives a good ranking for weighted tournaments. In *Proc. 17th ACM-SIAM Symp. Discrete Algorithms (SODA)*, pages 776–782, 2006.

[CFT99] A. Caprara, M. Fischetti, and P. Toth. A heuristic method for the set covering problem. *Operations Research*, 47:730–743, 1999.

[CFT00] A. Caprara, M. Fischetti, and P. Toth. Algorithms for the set covering problem. *Annals of Operations Research*, 98:353–371, 2000.

[CG94] B. Cherkassky and A. Goldberg. On implementing push-relabel method for the maximum flow problem. Technical Report 94-1523, Department of Computer Science, Stanford University, 1994.

[CGJ96] E. G. Coffman, M. R. Garey, and D. S. Johnson. Approximation algorithms for bin packing: a survey. In D. Hochbaum, editor, *Approximation algorithms*. PWS Publishing, 1996.

[CGJ98] C.R. Coullard, A.B. Gamble, and P.C. Jones. Matching problems in selective assembly operations. *Annals of Operations Research*, 76:95–107, 1998.

[CGK+97] C. Chekuri, A. Goldberg, D. Karger, M. Levine, and C. Stein. Experimental study of minimum cut algorithms. In *Proc. Symp. on Discrete Algorithms (SODA)*, pages 324–333, 1997.

[CGL85] B. Chazelle, L. Guibas, and D. T. Lee. The power of geometric duality. *BIT*, 25:76–90, 1985.

[CGM+98] B. Cherkassky, A. Goldberg, P. Martin, J. Setubal, and J. Stolfi. Augment or push: a computational study of bipartite matching and unit-capacity flow algorithms. *J. Experimental Algorithmics*, 3, 1998.

[CGPS76] H. L. Crane Jr., N. F. Gibbs, W. G. Poole Jr., and P. K. Stockmeyer. Matrix bandwidth and profile reduction. *ACM Trans. Math. Softw.*, 2(4):375–377, December 1976.

[CGR99] B. Cherkassky, A. Goldberg, and T. Radzik. Shortest paths algorithms: theory and experimental evaluation. *Math. Prog.*, 10:129–174, 1999.

[CGS99] B. Cherkassky, A. Goldberg, and C. Silverstein. Buckets, heaps, lists, and monotone priority queues. *SIAM J. Computing*, 28:1326–1346, 1999.

[CH06] D. Cook and L. Holder. *Mining Graph Data*. Wiley, 2006.

[Cha91] B. Chazelle. Triangulating a simple polygon in linear time. *Discrete and Computational Geometry*, 6:485–524, 1991.

[Cha00] B. Chazelle. A minimum spanning tree algorithm with inverse-Ackerman type complexity. *J. ACM*, 47:1028–1047, 2000.

[Cha01] T. Chan. Dynamic planar convex hull operations in near-logarithmic amortized time. *J. ACM*, 48:1–12, 2001.

[Che85] L. P. Chew. Planing the shortest path for a disc in $O(n^2 \lg n)$ time. In *Proc. First ACM Symp. Computational Geometry*, pages 214–220, 1985.

[CHL07] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007.

[Chr76] N. Christofides. Worst-case analysis of a new heuristic for the traveling salesman problem. Technical report, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh PA, 1976.

[Chu97] F. Chung. *Spectral Graph Theory*. AMS, Providence RI, 1997.

[Chv83] V. Chvatal. *Linear Programming*. Freeman, San Francisco, 1983.

[CIPR01] M Crochemore, C. Iliopolous, Y. Pinzon, and J. Reid. A fast and practical bit-vector algorithm for the longest common subsequence problem. *Info. Processing Letters*, 80:279–285, 2001.

[CK94] A. Chetverin and F. Kramer. Oligonucleotide arrays: New concepts and possibilities. *Bio/Technology*, 12:1093–1099, 1994.

[CK07] W. Cheney and D. Kincaid. *Numerical Mathematics and Computing*. Brooks/Cole, Monterey CA, sixth edition, 2007.

[CKSU05] H. Cohn, R. Kleinberg, B. Szegedy, and C. Umans. Group-theoretic algorithms for matrix multiplication. In *Proc. 46th Symp. Foundations of Computer Science*, pages 379–388, 2005.

[CL98] M. Crochemore and T. Lecroq. Text data compression algorithms. In M. J. Atallah, editor, *Algorithms and Theory of Computation Handbook*, pages 12.1–12.23. CRC Press Inc., Boca Raton, FL, 1998.

[Cla92] K. L. Clarkson. Safe and effective determinant evaluation. In *Proc. 31st IEEE Symposium on Foundations of Computer Science*, pages 387–395, Pittsburgh, PA, 1992.

[CLRS01] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge MA, second edition, 2001.

[CM69] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proc. 24th Nat. Conf. ACM*, pages 157–172, 1969.

[CM96] J. Cheriyan and K. Mehlhorn. Algorithms for dense graphs and networks on the random access computer. *Algorithmica*, 15:521–549, 1996.

[CM99] G. Del Corso and G. Manzini. Finding exact solutions to the bandwidth minimization problem. *Computing*, 62:189–203, 1999.

[Coh94] E. Cohen. Estimating the size of the transitive closure in linear time. In *35th Annual Symposium on Foundations of Computer Science*, pages 190–200. IEEE, 1994.

[Con71] J. H. Conway. *Regular Algebra and Finite Machines*. Chapman and Hall, London, 1971.

[Coo71] S. Cook. The complexity of theorem proving procedures. In *Proc. Third ACM Symp. Theory of Computing*, pages 151–158, 1971.

[CP90] R. Carraghan and P. Pardalos. An exact algorithm for the maximum clique problem. In *Operations Research Letters*, volume 9, pages 375–382, 1990.

[CP05] R. Crandall and C. Pomerance. *Prime Numbers: A Computational Perspective*. Springer, second edition, 2005.

[CPW98] B. Chen, C. Potts, and G. Woeginger. A review of machine scheduling: Complexity, algorithms and approximability. In D.-Z. Du and P. Pardalos, editors, *Handbook of Combinatorial Optimization*, volume 3, pages 21–169. Kluwer, 1998.

[CR76] J. Cohen and M. Roth. On the implementation of Strassen's fast multiplication algorithm. *Acta Informatica*, 6:341–355, 1976.

[CR99] W. Cook and A. Rohe. Computing minimum-weight perfect matchings. *INFORMS Journal on Computing*, 11:138–148, 1999.

[CR01] G. Del Corso and F. Romani. Heuristic spectral techniques for the reduction of bandwidth and work-bound of sparse matrices. *Numerical Algorithms*, 28:117–136, 2001.

[CR03] M. Crochemore and W. Rytter. *Jewels of Stringology*. World Scientific, 2003.

[CS93] J. Conway and N. Sloane. *Sphere packings, lattices, and groups*. Springer-Verlag, New York, 1993.

[CSG05] A. Caprara and J. Salazar-González. Laying out sparse graphs with provably minimum bandwidth. *INFORMS J. Computing*, 17:356–373, 2005.

[CT65] J. Cooley and J. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[CT92] Y. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proc. IEEE*, 80:1412–1434, 1992.

[CW90] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *J. Symbolic Computation*, pages 251–280, 1990.

[Dan63] G. Dantzig. *Linear programming and extensions*. Princeton University Press, Princeton NJ, 1963.

[Dan94] V. Dancik. Expected length of longest common subsequences. PhD. thesis, Univ. of Warwick, 1994.

[DB74] G. Dahlquist and A. Bjorck. *Numerical Methods*. Prentice-Hall, Englewood Cliffs NJ, 1974.

[DB86] G. Davies and S. Bowsher. Algorithms for pattern matching. *Software – Practice and Experience*, 16:575–601, 1986.

[dBDK+98] M. de Berg, O. Devillers, M. Kreveld, O. Schwarzkopf, and M. Teillaud. Computing the maximum overlap of two convex polygons under translations. *Theoretical Computer Science*, 31:613–628, 1998.

[dBvKOS00] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, second edition, 2000.

[DEKM98] R. Durbin, S. Eddy, A. Krough, and G. Mitchison. *Biological Sequence Analysis*. Cambridge University Press, 1998.

[Den05] L. Y. Deng. Efficient and portable multiple recursive generators of large order. *ACM Trans. on Modeling and Computer Simulation*, 15:1–13, 2005.

[Dey06] T. Dey. *Curve and Surface Reconstruction: Algorithms with Mathematical Analysis*. Cambridge Univ. Press, 2006.

[DF79] E. Denardo and B. Fox. Shortest-route methods: 1. reaching, pruning, and buckets. *Operations Research*, 27:161–186, 1979.

[DFJ54] G. Dantzig, D. Fulkerson, and S. Johnson. Solution of a large-scale traveling-salesman problem. *Operations Research*, 2:393–410, 1954.

[dFPP90] H. de Fraysseix, J. Pach, and R. Pollack. How to draw a planar graph on a grid. *Combinatorica*, 10:41–51, 1990.

[DGH+02] E. Dantsin, A. Goerdt, E. Hirsch, R. Kannan, J. Kleinberg, C. Papadimitriou, P. Raghavan, and U. Schöning. A deterministic $(2 - 2/(k + 1))n$ algorithm for k-SAT based on local search. *Theoretical Computer Science*, 289:69–83, 2002.

[DGKK79] R. Dial, F. Glover, D. Karney, and D. Klingman. A computational analysis of alternative algorithms and labeling techniques for finding shortest path trees. *Networks*, 9:215–248, 1979.

[DH92] D. Du and F. Hwang. A proof of Gilbert and Pollak's conjecture on the Steiner ratio. *Algorithmica*, 7:121–135, 1992.

[DHS00] R. Duda, P. Hart, and D. Stork. *Pattern Classification*. Wiley-Interscience, New York, second edition, 2000.

[Die04] M. Dietzfelbinger. *Primality Testing in Polynomial Time: From Randomized Algorithms to "PRIMES Is in P"*. Springer, 2004.

[Dij59] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[DJ92] G. Das and D. Joseph. Minimum vertex hulls for polyhedral domains. *Theoret. Comput. Sci.*, 103:107–135, 1992.

[Dji00] H. Djidjev. Computing the girth of a planar graph. In *Proc. 27th Int. Colloquium on Automata, Languages and Programming (ICALP)*, pages 821–831, 2000.

[DJP04] E. Demaine, T. Jones, and M. Patrascu. Interpolation search for non-independent data. In *Proc. 15th ACM-SIAM Symp. Discrete Algorithms (SODA)*, pages 522–523, 2004.

[DL76] D. Dobkin and R. Lipton. Multidimensional searching problems. *SIAM J. Computing*, 5:181–186, 1976.

[DLR79] D. Dobkin, R. Lipton, and S. Reiss. Linear programming is log-space hard for P. *Info. Processing Letters*, 8:96–97, 1979.

[DM80] D. Dobkin and J. I. Munro. Determining the mode. *Theoretical Computer Science*, 12:255–263, 1980.

[DM97] K. Daniels and V. Milenkovic. Multiple translational containment. part I: an approximation algorithm. *Algorithmica*, 19:148–182, 1997.

[DMBS79] J. Dongarra, C. Moler, J. Bunch, and G. Stewart. *LINPACK User's Guide*. SIAM Publications, Philadelphia, 1979.

[DMR97] K. Daniels, V. Milenkovic, and D. Roth. Finding the largest area axis-parallel rectangle in a polygon. *Computational Geometry: Theory and Applications*, 7:125–148, 1997.

[DN07] P. D'Alberto and A. Nicolau. Adaptive Strassen's matrix multiplication. In *Proc. 21st Int. Conf. on Supercomputing*, pages 284–292, 2007.

[DP73] D. H. Douglas and T. K. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Canadian Cartographer*, 10(2):112–122, December 1973.

[DPS02] J. Diaz, J. Petit, and M. Serna. A survey of graph layout problems. *ACM Computing Surveys*, 34:313–356, 2002.

[DR90] N. Dershowitz and E. Reingold. Calendrical calculations. *Software – Practice and Experience*, 20:899–928, 1990.

[DR02] N. Dershowitz and E. Reingold. *Calendrical Tabulations: 1900-2200*. Cambridge University Press, New York, 2002.

[DRR+95] S. Dawson, C. R. Ramakrishnan, I. V. Ramakrishnan, K. Sagonas, S. Skiena, T. Swift, and D. S. Warren. Unification factoring for efficient execution of logic programs. In *22nd ACM Symposium on Principles of Programming Languages (POPL '95)*, pages 247–258, 1995.

[DSR00] D. Du, J. Smith, and J. Rubinstein. *Advances in Steiner Trees*. Kluwer, 2000.

[DT04] M. Dorigo and T.Stutzle. *Ant Colony Optimization*. MIT Press, Cambridge MA, 2004.

[dVS82] G. de V. Smit. A comparison of three string matching algorithms. *Software – Practice and Experience*, 12:57–66, 1982.

[dVV03] S. de Vries and R. Vohra. Combinatorial auctions: A survey. *Informs J. Computing*, 15:284–309, 2003.

[DY94] Y. Deng and C. Yang. Waring's problem for pyramidal numbers. *Science in China (Series A)*, 37:377–383, 1994.

[DZ99] D. Dor and U. Zwick. Selecting the median. *SIAM J. Computing*, pages 1722–1758, 1999.

[DZ01] D. Dor and U. Zwick. Median selection requires $(2+\epsilon)n$ comparisons. *SIAM J. Discrete Math.*, 14:312–325, 2001.

[Ebe88] J. Ebert. Computing Eulerian trails. *Info. Proc. Letters*, 28:93–97, 1988.

[ECW92] V. Estivill-Castro and D. Wood. A survey of adaptive sorting algorithms. *ACM Computing Surveys*, 24:441–476, 1992.

[Ede87] H. Edelsbrunner. *Algorithms for Combinatorial Geometry*. Springer-Verlag, Berlin, 1987.

[Ede06] H. Edelsbrunner. *Geometry and Topology for Mesh Generation*. Cambridge Univ. Press, 2006.

[Edm65] J. Edmonds. Paths, trees, and flowers. *Canadian J. Math.*, 17:449–467, 1965.

[Edm71] J. Edmonds. Matroids and the greedy algorithm. *Mathematical Programming*, 1:126–136, 1971.

[EE99] D. Eppstein and J. Erickson. Raising roofs, crashing cycles, and playing pool: applications of a data structure for finding pairwise interactions. *Disc. Comp. Geometry*, 22:569–592, 1999.

[EG60] P. Erdős and T. Gallai. Graphs with prescribed degrees of vertices. *Mat. Lapok (Hungarian)*, 11:264–274, 1960.

[EG89] H. Edelsbrunner and L. Guibas. Topologically sweeping an arrangement. *J. Computer and System Sciences*, 38:165–194, 1989.

[EG91] H. Edelsbrunner and L. Guibas. Corrigendum: Topologically sweeping an arrangement. *J. Computer and System Sciences*, 42:249–251, 1991.

[EGIN92] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification: A technique for speeding up dynamic graph algorithms. In *Proc. 33rd IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 60–69, 1992.

[EGS86] H. Edelsbrunner, L. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM J. Computing*, 15:317–340, 1986.

[EJ73] J. Edmonds and E. Johnson. Matching, Euler tours, and the Chinese postman. *Math. Programming*, 5:88–124, 1973.

[EK72] J. Edmonds and R. Karp. Theoretical improvements in the algorithmic efficiency for network flow problems. *J. ACM*, 19:248–264, 1972.

[EKA84] M. I. Edahiro, I. Kokubo, and T. Asano. A new point location algorithm and its practical efficiency – comparison with existing algorithms. *ACM Trans. Graphics*, 3:86–109, 1984.

[EKS83] H. Edelsbrunner, D. Kirkpatrick, and R. Seidel. On the shape of a set of points in the plane. *IEEE Trans. on Information Theory*, IT-29:551–559, 1983.

[EL01] S. Ehmann and M. Lin. Accurate and fast proximity queries between polyhedra using convex surface decomposition. *Comp. Graphics Forum*, 20:500–510, 2001.

[EM94] H. Edelsbrunner and E. Mücke. Three-dimensional alpha shapes. *ACM Transactions on Graphics*, 13:43–72, 1994.

[ENSS98] G. Even, J. Naor, B. Schieber, and M. Sudan. Approximating minimum feedback sets and multi-cuts in directed graphs. *Algorithmica*, 20:151–174, 1998.

[Epp98] D. Eppstein. Finding the k shortest paths. *SIAM J. Computing*, 28:652–673, 1998.

[ES86] H. Edelsbrunner and R. Seidel. Voronoi diagrams and arrangements. *Discrete and Computational Geometry*, 1:25–44, 1986.

[ESS93] H. Edelsbrunner, R. Seidel, and M. Sharir. On the zone theorem for hyperplane arrangements. *SIAM J. Computing*, 22:418–429, 1993.

[ESV96] F. Evans, S. Skiena, and A. Varshney. Optimizing triangle strips for fast rendering. In *Proc. IEEE Visualization '96*, pages 319–326, 1996.

[Eul36] L. Euler. Solutio problematis ad geometriam situs pertinentis. *Commentarii Academiae Scientiarum Petropolitanae*, 8:128–140, 1736.

[Eve79a] S. Even. *Graph Algorithms*. Computer Science Press, Rockville MD, 1979.

[Eve79b] G. Everstine. A comparison of three resequencing algorithms for the reduction of matrix profile and wave-front. *Int. J. Numerical Methods in Engr.*, 14:837–863, 1979.

[FÁ48] I. Fáry. On straight line representation of planar graphs. *Acta. Sci. Math. Szeged*, 11:229–233, 1948.

[Fei98] U. Feige. A threshold of $\ln n$ for approximating set cover. *J. ACM*, 45:634–652, 1998.

[FF62] L. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton NJ, 1962.

[FG95] U. Feige and M. Goemans. Approximating the value of two prover proof systems, with applications to max 2sat and max dicut. In *Proc. 3rd Israel Symp. on Theory of Computing and Systems*, pages 182–189, 1995.

[FH06] E. Fogel and D. Halperin. Exact and efficient construction of Minkowski sums for convex polyhedra with applications. In *Proc. 6th Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2006.

[FHW07] E. Fogel, D. Halperin, and C. Weibel. On the exact maximum complexity of minkowski sums of convex polyhedra. In *Proc. 23rd Symp. Computational Geometry*, pages 319–326, 2007.

[FJ05] M. Frigo and S. Johnson. The design and implementation of FFTW3. *Proc. IEEE*, 93:216–231, 2005.

[FJMO93] M. Fredman, D. Johnson, L. McGeoch, and G. Ostheimer. Data structures for traveling salesmen. In *Proc. 4th 7th Symp. Discrete Algorithms (SODA)*, pages 145–154, 1993.

[Fle74] H. Fleischner. The square of every two-connected graph is Hamiltonian. *J. Combinatorial Theory, B*, 16:29–34, 1974.

[Fle80] R. Fletcher. *Practical Methods of Optimization: Unconstrained Optimization*, volume 1. John Wiley, Chichester, 1980.

[Flo62] R. Floyd. Algorithm 97 (shortest path). *Communications of the ACM*, 7:345, 1962.

[Flo64] R. Floyd. Algorithm 245 (treesort). *Communications of the ACM*, 18:701, 1964.

[FLPR99] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. 40th Symp. Foundations of Computer Science*, 1999.

[FM71] M. Fischer and A. Meyer. Boolean matrix multiplication and transitive closure. In *IEEE 12th Symp. on Switching and Automata Theory*, pages 129–131, 1971.

[FM82] C. Fiduccia and R. Mattheyses. A linear time heuristic for improving network partitions. In *Proc. 19th IEEE Design Automation Conf.*, pages 175–181, 1982.

[FN04] K. Fredriksson and G. Navarro. Average-optimal single and multiple approximate string matching. *ACM J. of Experimental Algorithmics*, 9, 2004.

[For87] S. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[For04] S. Fortune. Voronoi diagrams and Delauney triangulations. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 513–528. CRC Press, 2004.

[FPR99] P. Festa, P. Pardalos, and M. Resende. Feedback set problems. In D.-Z. Du and P.M. Pardalos, editors, *Handbook of Combinatorial Optimization*, volume A. Kluwer, 1999.

[FPR01] P. Festa, P. Pardalos, and M. Resende. Algorithm 815: Fortran subroutines for computing approximate solution to feedback set problems using GRASP. *ACM Transactions on Mathematical Software*, 27:456–464, 2001.

[FR75] R. Floyd and R. Rivest. Expected time bounds for selection. *Communications of the ACM*, 18:165–172, 1975.

[FR94] M. Fürer and B. Raghavachari. Approximating the minimum-degree Steiner tree to within one of optimal. *J. Algorithms*, 17:409–423, 1994.

[Fra79] D. Fraser. An optimized mass storage FFT. *ACM Trans. Math. Softw.*, 5(4):500–517, December 1979.

[Fre62] E. Fredkin. Trie memory. *Communications of the ACM*, 3:490–499, 1962.

[Fre76] M. Fredman. How good is the information theory bound in sorting? *Theoretical Computer Science*, 1:355–361, 1976.

[FS03] N. Ferguson and B. Schneier. *Practical Cryptography*. Wiley, 2003.

[FSV01] P. Foggia, C. Sansone, and M. Vento. A performance comparison of five algorithms for graph isomorphism. In *3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*, 2001.

[FT87] M. Fredman and R. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34:596–615, 1987.

[FvW93] S. Fortune and C. van Wyk. Efficient exact arithmetic for computational geometry. In *Proc. 9th ACM Symp. Computational Geometry*, pages 163–172, 1993.

[FW77] S. Fiorini and R. Wilson. *Edge-colourings of graphs*. Research Notes in Mathematics 16, Pitman, London, 1977.

[FW93] M. Fredman and D. Willard. Surpassing the information theoretic bound with fusion trees. *J. Computer and System Sci.*, 47:424–436, 1993.

[FWH04] E. Folgel, R. Wein, and D. Halperin. Code flexibility and program efficiency by genericity: Improving CGAL's arrangements. In *Proc. 12th European Symposium on Algorithms (ESA'04)*, pages 664–676, 2004.

[Gab76] H. Gabow. An efficient implementation of Edmond's algorithm for maximum matching on graphs. *J. ACM*, 23:221–234, 1976.

[Gab77] H. Gabow. Two algorithms for generating weighted spanning trees in order. *SIAM J. Computing*, 6:139–150, 1977.

[Gal86] Z. Galil. Efficient algorithms for finding maximum matchings in graphs. *ACM Computing Surveys*, 18:23–38, 1986.

[Gal90] K. Gallivan. *Parallel Algorithms for Matrix Computations*. SIAM, Philadelphia, 1990.

[Gas03] S. Gass. *Linear Programming: Methods and Applications*. Dover, fifth edition, 2003.

[GBDS80] B. Golden, L. Bodin, T. Doyle, and W. Stewart. Approximate traveling salesman algorithms. *Operations Research*, 28:694–711, 1980.

[GBY91] G. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures*. Addison-Wesley, Wokingham, England, second edition, 1991.

[Gen04] J. Gentle. *Random Number Generation and Monte Carlo Methods*. Springer, second edition, 2004.

[GGJ77] M. Garey, R. Graham, and D. Johnson. The complexity of computing Steiner minimal trees. *SIAM J. Appl. Math.*, 32:835–859, 1977.

[GGJK78] M. Garey, R. Graham, D. Johnson, and D. Knuth. Complexity results for bandwidth minimization. *SIAM J. Appl. Math.*, 34:477–495, 1978.

[GH85] R. Graham and P. Hell. On the history of the minimum spanning tree problem. *Annals of the History of Computing*, 7:43–57, 1985.

[GH06] P. Galinier and A. Hertz. A survey of local search methods for graph coloring. *Computers and Operations Research*, 33:2547–2562, 2006.

[GHMS93] L. J. Guibas, J. E. Hershberger, J. S. B. Mitchell, and J. S. Snoeyink. Approximating polygons and subdivisions with minimum link paths. *Internat. J. Comput. Geom. Appl.*, 3(4):383–415, December 1993.

[GHR95] R. Greenlaw, J. Hoover, and W. Ruzzo. *Limits to Parallel Computation: P-completeness theory*. Oxford University Press, New York, 1995.

[GI89] D. Gusfield and R. Irving. *The Stable Marriage Problem: structure and algorithms*. MIT Press, Cambridge MA, 1989.

[GI91] Z. Galil and G. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys*, 23:319–344, 1991.

[Gib76] N. E. Gibbs. A hybrid profile reduction algorithm. *ACM Trans. Math. Softw.*, 2(4):378–387, December 1976.

[Gib85] A. Gibbons. *Algorithmic Graph Theory*. Cambridge Univ. Press, 1985.

[GJ77] M. Garey and D. Johnson. The rectilinear Steiner tree problem is NP-complete. *SIAM J. Appl. Math.*, 32:826–834, 1977.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the theory of NP-completeness*. W. H. Freeman, San Francisco, 1979.

[GJM02] M. Goldwasser, D. Johnson, and C. McGeoch, editors. *Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges*, volume 59. AMS, Providence RI, 2002.

[GJPT78] M. Garey, D. Johnson, F. Preparata, and R. Tarjan. Triangulating a simple polygon. *Info. Proc. Letters*, 7:175–180, 1978.

[GK95] A. Goldberg and R. Kennedy. An efficient cost scaling algorithm for the assignment problem. *Math. Programming*, 71:153–177, 1995.

[GK98] S. Guha and S. Khuller. Approximation algorithms for connected dominating sets. *Algorithmica*, 20:374–387, 1998.

[GKK74] F. Glover, D. Karney, and D. Klingman. Implementation and computational comparisons of primal-dual computer codes for minimum-cost network flow problems. *Networks*, 4:191–212, 1974.

[GKP89] R. Graham, D. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley, Reading MA, 1989.

[GKS95] S. Gupta, J. Kececioglu, and A. Schäffer. Improving the practical space and time efficiency of the shortest-paths approach to sum-of-pairs multiple sequence alignment. *J. Computational Biology*, 2:459–472, 1995.

[GKT05] D. Gibson, R. Kumar, and A. Tomkins. Discovering large dense subgraphs in massive graphs. In *Proc. 31st Int. Conf on Very Large Data Bases*, pages 721–732, 2005.

[GKW06] A. Goldberg, H. Kaplan, and R. Werneck. Reach for A*: Efficient point-to-point shortest path algorithms. In *Proc. 8th Workshop on Algorithm Engineering and Experimentation (ALENEX)*, 2006.

[GKW07] A. Goldberg, H. Kaplan, and R. Werneck. Better landmarks within reach. In *Proc. 9th Workshop on Algorithm Engineering and Experimentation (ALEN EX)*, pages 38–51, 2007.

[GL96] G. Golub and C. Van Loan. *Matrix Computations*. Johns Hopkins University Press, third edition, 1996.

[Glo90] F. Glover. Tabu search: A tutorial. *Interfaces*, 20 (4):74–94, 1990.

[GM86] G. Gonnet and J. I. Munro. Heaps on heaps. *SIAM J. Computing*, 15:964–971, 1986.

[GM91] S. Ghosh and D. Mount. An output-sensitive algorithm for computing visibility graphs. *SIAM J. Computing*, 20:888–910, 1991.

[GMPV06] F. Gomes, C. Meneses, P. Pardalos, and G. Viana. Experimental analysis of approximation algorithms for the vertex cover and set covering problems. *Computers and Operations Research*, 33:3520–3534, 2006.

[GO04] J. Goodman and J. O'Rourke, editors. *Handbook of Discrete and Computational Geometry*. CRC Press, second edition, 2004.

[Goe97] M. Goemans. Semidefinite programming in combinatorial optimization. *Mathematical Programming*, 79:143–161, 1997.

[Gol93] L. Goldberg. *Efficient Algorithms for Listing Combinatorial Structures*. Cambridge University Press, 1993.

[Gol97] A. Goldberg. An efficient implementation of a scaling minimum-cost flow algorithm. *J. Algorithms*, 22:1–29, 1997.

[Gol01] A. Goldberg. Shortest path algorithms: Engineering aspects. In *12th International Symposium on Algorithms and Computation*, number 2223 in LNCS, pages 502–513. Springer, 2001.

[Gol04] M. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*, volume 57 of *Annals of Discrete Mathematics*. North Holland, second edition, 2004.

[Gon07] T. Gonzalez. *Handbook of Approximation Algorithms and Metaheuristics*. Chapman-Hall / CRC, 2007.

[GP68] E. Gilbert and H. Pollak. Steiner minimal trees. *SIAM J. Applied Math.*, 16:1–29, 1968.

[GP79] B. Gates and C. Papadimitriou. Bounds for sorting by prefix reversals. *Discrete Mathematics*, 27:47–57, 1979.

[GP07] G. Gutin and A. Punnen. *The Traveling Salesman Problem and Its Variations*. Springer, 2007.

[GPS76] N. Gibbs, W. Poole, and P. Stockmeyer. A comparison of several bandwidth and profile reduction algorithms. *ACM Trans. Math. Software*, 2:322–330, 1976.

[Gra53] F. Gray. Pulse code communication. US Patent 2632058, March 17, 1953.

[Gra72] R. Graham. An efficient algorithm for determining the convex hull of a finite planar point set. *Info. Proc. Letters*, 1:132–133, 1972.

[Gri89] D. Gries. *The Science of Programming*. Springer-Verlag, 1989.

[GS62] D. Gale and L. Shapely. College admissions and the stability of marriages. *American Math. Monthly*, 69:9–14, 1962.

[GS02] R. Giugno and D. Shasha. Graphgrep : A fast and universal method for querying graphs. In *International Conference on Pattern Recognition (ICPR)*, volume 2, pages 112–115, 2002.

[GT88] A. Goldberg and R. Tarjan. A new approach to the maximum flow problem. *J. ACM*, pages 921–940, 1988.

[GT94] T. Gensen and B. Toft. *Graph Coloring Problems*. Wiley, 1994.

[GT05] M. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java*. Wiley, fourth edition, 2005.

[GTV05] M. Goodrich, R. Tamassia, and L. Vismara. Data structures in JDSL. In D. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, pages 43:1–43:22. Chapman and Hall / CRC, 2005.

[Gup66] R. P. Gupta. The chromatic index and the degree of a graph. *Notices of the Amer. Math. Soc.*, 13:719, 1966.

[Gus94] D. Gusfield. Faster implementation of a shortest superstring approximation. *Info. Processing Letters*, 51:271–274, 1994.

[Gus97] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[GW95] M. Goemans and D. Williamson. .878-approximation algorithms for MAX CUT and MAX 2SAT. *J. ACM*, 42:1115–1145, 1995.

[GW96] I. Goldberg and D. Wagner. Randomness and the Netscape browser. *Dr. Dobb's Journal*, pages 66–70, 1996.

[GW97] T. Grossman and A. Wool. Computational experience with approximation algorithms for the set covering problem. *European J. Operational Research*, 101, 1997.

[Hai94] E. Haines. Point in polygon strategies. In P. Heckbert, editor, *Graphics Gemes IV*, pages 24–46. Academic Press, 1994.

[Hal04] D. Halperin. Arrangements. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 24, pages 529–562. CRC Press, Boca Raton, FL, 2004.

[Ham87] R. Hamming. *Numerical Methods for Scientists and Engineers*. Dover, second edition, 1987.

[Has82] H. Hastad. Clique is hard to approximate within $n^{1-\epsilon}$. *Acta Mathematica*, 182:105–142, 182.

[Has97] J. Hastad. Some optimal inapproximability results. In *Proc. 29th ACM Symp. Theory of Comp.*, pages 1–10, 1997.

[HD80] P. Hall and G. Dowling. Approximate string matching. *ACM Computing Surveys*, 12:381–402, 1980.

[HDD03] M. Hilgemeier, N. Drechsler, and R. Drchsler. Fast heuristics for the edge coloring of large graphs. In *Proc. Euromicro Symp. on Digital Systems Design*, pages 230–239, 2003.

[HdlT01] J. Holm, K. de lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48:723–760, 2001.

[Hel01] M. Held. VRONI: An engineering approach to the reliable and efficient computation of Voronoi diagrams of points and line segments. *Computational Geometry: Theory and Applications*, 18:95–123, 2001.

[HFN05] H. Hyyro, K. Fredriksson, and G. Navarro. Increased bit-parallelism for approximate and multiple string matching. *ACM J. of Experimental Algorithmics*, 10, 2005.

[HG97] P. Heckbert and M. Garland. Survey of polygonal surface simplification algorithms. SIGGRAPH 97 Course Notes, 1997.

[HH00] I. Hanniel and D. Halperin. Two-dimensional arrangements in CGAL and adaptive point location for parametric curves. In *Proc. 4th International Workshop on Algorithm Engineering (WAE), LNCS v. 1982*, pages 171–182, 2000.

[HHS98] T. Haynes, S. Hedetniemi, and P. Slater. *Fundamentals of Domination in Graphs*. CRC Press, Boca Raton, 1998.

[Hir75] D. Hirschberg. A linear-space algorithm for computing maximum common subsequences. *Communications of the ACM*, 18:341–343, 1975.

[HK73] J. Hopcroft and R. Karp. An $n^{5/3}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Computing*, 2:225–231, 1973.

[HK90] D. P. Huttenlocher and K. Kedem. Computing the minimum Hausdorff distance for point sets under translation. In *Proc. 6th Annu. ACM Sympos. Comput. Geom.*, pages 340–349, 1990.

[HLD04] W. Hörmann, J. Leydold, and G. Derfinger. *Automatic Nonuniform Random Variate Generation*. Springer, 2004.

[HM83] S. Hertel and K. Mehlhorn. Fast triangulation of simple polygons. In *Proc. 4th Internat. Conf. Found. Comput. Theory*, pages 207–218. Lecture Notes in Computer Science, Vol. 158, 1983.

[HM99] X. Huang and A. Madan. Cap3: A DNA sequence assembly program. *Genome Research*, 9:868–877, 1999.

[HMS03] J. Hershberger, M. Maxel, and S. Suri. Finding the k shortest simple paths: A new algorithm and its implementation. In *Proc. 5th Workshop on Algorithm Engineering and Experimentation (ALEN EX)*, 2003.

[HMU06] J. Hopcroft, R. Motwani, and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, third edition, 2006.

[Hoa61] C. A. R. Hoare. Algorithm 63 (partition) and algorithm 65 (find). *Communications of the ACM*, 4:321–322, 1961.

[Hoa62] C. A. R. Hoare. Quicksort. *Computer Journal*, 5:10–15, 1962.

[Hoc96] D. Hochbaum, editor. *Approximation Algorithms for NP-hard Problems*. PWS Publishing, Boston, 1996.

[Hof82] C. M. Hoffmann. *Group-theoretic algorithms and graph isomorphism*, volume 136 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, 1982.

[Hol75] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975.

[Hol81] I. Holyer. The NP-completeness of edge colorings. *SIAM J. Computing*, 10:718–720, 1981.

[Hol92] J. H. Holland. Genetic algorithms. *Scientific American*, 267(1):66–72, July 1992.

[Hop71] J. Hopcroft. An $n \log n$ algorithm for minimizing the states in a finite automaton. In Z. Kohavi, editor, *The theory of machines and computations*, pages 189–196. Academic Press, New York, 1971.

[Hor80] R. N. Horspool. Practical fast searching in strings. *Software – Practice and Experience*, 10:501–506, 1980.

[HP73] F. Harary and E. Palmer. *Graphical enumeration*. Academic Press, New York, 1973.

[HPS+05] M. Holzer, G. Prasinos, F. Schulz, D. Wagner, and C. Zaroliagis. Engineering planar separator algorithms. In *Proc. 13th European Symp. on Algorithms (ESA)*, pages 628–637, 2005.

[HRW92] R. Hwang, D. Richards, and P. Winter. *The Steiner Tree Problem*, volume 53 of *Annals of Discrete Mathematics*. North Holland, Amsterdam, 1992.

[HS77] J. Hunt and T. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20:350–353, 1977.

[HS94] J. Hershberger and J. Snoeyink. An $O(n \log n)$ implementation of the Douglas-Peucker algorithm for line simplification. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pages 383–384, 1994.

[HS98] J. Hershberger and J. Snoeyink. Cartographic line simplification and polygon CSG formulae in $\mathrm{O}(n \log^* n)$ time. *Computational Geometry: Theory and Applications*, 11:175–185, 1998.

[HS99] J. Hershberger and S. Suri. An optimal algorithm for Euclidean shortest paths in the plane. *SIAM J. Computing*, 28:2215–2256, 1999.

[HSS87] J. Hopcroft, J. Schwartz, and M. Sharir. *Planning, geometry, and complexity of robot motion*. Ablex Publishing, Norwood NJ, 1987.

[HSS07] R. Hardin, N. Sloane, and W. Smith. Maximum volume spherical codes. http://www.research.att.com/∼njas/maxvolumes/, 2007.

[HSWW05] M. Holzer, F. Schultz, D. Wagner, and T. Willhalm. Combining speed-up techniques for shortest-path computations. *ACM J. of Experimental Algorithmics*, 10, 2005.

[HT73a] J. Hopcroft and R. Tarjan. Dividing a graph into triconnected components. *SIAM J. Computing*, 2:135–158, 1973.

[HT73b] J. Hopcroft and R. Tarjan. Efficient algorithms for graph manipulation. *Communications of the ACM*, 16:372–378, 1973.

[HT74] J. Hopcroft and R. Tarjan. Efficient planarity testing. *J. ACM*, 21:549–568, 1974.

[HT84] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13:338–355, 1984.

[Hub06] M. Huber. Fast perfect sampling from linear extensions. *Disc. Math.*, 306:420–428, 2006.

[Huf52] D. Huffman. A method for the construction of minimum-redundancy codes. *Proc. of the IRE*, 40:1098–1101, 1952.

[HUW02] E. Haunschmid, C. Ueberhuber, and P. Wurzinger. Cache oblivious high performance algorithms for matrix multiplication, 2002.

[HW74] J. E. Hopcroft and J. K. Wong. Linear time algorithm for isomorphism of planar graphs. In *Proc. Sixth Annual ACM Symposium on Theory of Computing*, pages 172–184, 1974.

[HWA+03] X. Huang, J. Wang, S. Aluru, S. Yang, and L. Hillier. PCAP: A whole-genome assembly program. *Genome Research*, 13:2164–2170, 2003.

[HWK94] T. He, S. Wang, and A. Kaufman. Wavelet-based volume morphing. In *Proc. IEEE Visualization '94*, pages 85–92, 1994.

[IK75] O. Ibarra and C. Kim. Fast approximation algorithms for knapsack and sum of subset problems. *J. ACM*, 22:463–468, 1975.

[IM04] P. Indyk and J. Matousek. Low-distortion embeddings of finite metric spaces. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*. CRC Press, 2004.

[Ind98] P. Indyk. Faster algorithms for string matching problems: matching the convolution bound. In *Proc. 39th Symp. Foundations of Computer Science*, 1998.

[Ind04] P. Indyk. Nearest neighbors in high-dimensional spaces. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 877–892. CRC Press, 2004.

[IR78] A. Itai and M. Rodeh. Finding a minimum circuit in a graph. *SIAM J. Computing*, 7:413–423, 1978.

[Ita78] A. Itai. Two commodity flow. *J. ACM*, 25:596–611, 1978.

[JÓ92] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[Jac89] G. Jacobson. Space-efficient static trees and graphs. In *Proc. Symp. Foundations of Computer Science (FOCS)*, pages 549–554, 1989.

[JAMS91] D. Johnson, C. Aragon, C. McGeoch, and D. Schevon. Optimization by simulated annealing: an experimental evaluation; part II, graph coloring and number partitioning. In *Operations Research*, volume 39, pages 378–406, 1991.

[Jar73] R. A. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Info. Proc. Letters*, 2:18–21, 1973.

[JD88] A. Jain and R. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, Englewood Cliffs NJ, 1988.

[JLR00] S. Janson, T. Luczak, and A. Rucinski. *Random Graphs*. Wiley, 2000.

[JM93] D. Johnson and C. McGeoch, editors. *Network Flows and Matching: First DIMACS Implementation Challenge*, volume 12. American Mathematics Society, Providence RI, 1993.

[JM03] M. Jünger and P. Mutzel. *Graph Drawing Software*. Springer-Verlag, 2003.

[Joh63] S. M. Johnson. Generation of permutations by adjacent transpositions. *Math. Computation*, 17:282–285, 1963.

[Joh74] D. Johnson. Approximation algorithms for combinatorial problems. *J. Computer and System Sciences*, 9:256–278, 1974.

[Joh90] D. S. Johnson. A catalog of complexity classes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Algorithms and Complexity*, volume A, pages 67–162. MIT Press, 1990.

[Jon86] D. W. Jones. An empirical comparison of priority-queue and event-set implementations. *Communications of the ACM*, 29:300–311, 1986.

[Jos99] N. Josuttis. *The C++ Standard Library: A tutorial and reference*. Addison-Wesley, 1999.

[JR93] T. Jiang and B. Ravikumar. Minimal NFA problems are hard. *SIAM J. Computing*, 22:1117–1141, 1993.

[JS01] A. Jagota and L. Sanchis. Adaptive, restart, randomized greedy heuristics for maximum clique. *J. Heuristics*, 7:1381–1231, 2001.

[JSV01] M. Jerrum, A. Sinclair, and E. Vigoda. A polynomial-time approximation algorithm for the permanent of a matrix with non-negative entries. In *Proc. 33rd ACM Symp. Theory of Computing*, pages 712–721, 2001.

[JT96] D. Johnson and M. Trick. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, volume 26. AMS, Providence RI, 1996.

[KA03] P. Ko and S. Aluru. Space-efficient linear time construction of suffix arrays,. In *Proc. 14th Symp. on Combinatorial Pattern Matching (CPM)*, pages 200–210. Springer-Verlag LNCS, 2003.

[Kah67] D. Kahn. *The Code breakers: the story of secret writing*. Macmillan, New York, 1967.

[Kar72] R. M. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[Kar84] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.

[Kar96] H. Karloff. How good is the Goemans-Williamson MAX CUT algorithm? In *Proc. Twenty-Eighth Annual ACM Symposium on Theory of Computing*, pages 427–434, 1996.

[Kar00] D. Karger. Minimum cuts in near-linear time. *J. ACM*, 47:46–76, 200.

[Kei00] M. Keil. Polygon decomposition. In J.R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 491–518. Elsevier, 2000.

[KGV83] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[Kha79] L. Khachian. A polynomial algorithm in linear programming. *Soviet Math. Dokl.*, 20:191–194, 1979.

[Kir79] D. Kirkpatrick. Efficient computation of continuous skeletons. In *Proc. 20th IEEE Symp. Foundations of Computing*, pages 28–35, 1979.

[Kir83] D. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Computing*, 12:28–35, 1983.

[KKT95] D. Karger, P. Klein, and R. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *J. ACM*, 42:321–328, 1995.

[KL70] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, pages 291–307, 1970.

[KM72] V. Klee and G. Minty. How good is the simplex algorithm. In *Inequalities III*, pages 159–172, New York, 1972. Academic Press.

[KM95] J. D. Kececioglu and E. W. Myers. Combinatorial algorithms for DNA sequence assembly. *Algorithmica*, 13(1/2):7–51, January 1995.

[KMP77] D. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM J. Computing*, 6:323–350, 1977.

[KMP+04] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C. Yap. Classroom examples of robustness problems in geometric computations. In *Proc. 12th European Symp. on Algorithms (ESA'04)*, pages 702–713. www.mpi-inf.mpg.de/∼mehlhorn/ftp/ClassRoomExamples.ps, 2004.

[KMS96] J. Komlos, Y. Ma, and E. Szemeredi. Matching nuts and bolts in $o(n \log n)$ time. In *Proc. 7th Symp. Discrete Algorithms (SODA)*, pages 232–241, 1996.

[KMS97] S. Khanna, M. Muthukrishnan, and S. Skiena. Efficiently partitioning arrays. In *Proc. ICALP '97*, volume 1256, pages 616–626. Springer-Verlag LNCS, 1997.

[Knu94] D. Knuth. *The Stanford GraphBase: a platform for combinatorial computing*. ACM Press, New York, 1994.

[Knu97a] D. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, Reading MA, third edition, 1997.

[Knu97b] D. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Reading MA, third edition, 1997.

[Knu98] D. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, Reading MA, second edition, 1998.

[Knu05a] D. Knuth. *The Art of Computer Programming, Volume 4 Fascicle 2: Generating All Tuples and Permutations*. Addison Wesley, 2005.

[Knu05b] D. Knuth. *The Art of Computer Programming, Volume 4 Fascicle 3: Generating All Combinations and Partitions*. Addison Wesley, 2005.

[Knu06] D. Knuth. *The Art of Computer Programming, Volume 4 Fascicle 4: Generating All Trees; History of Combinationatorial Generation*. Addison Wesley, 2006.

[KO63] A. Karatsuba and Yu. Ofman. Multiplication of multi-digit numbers on automata. *Sov. Phys. Dokl.*, 7:595–596, 1963.

[Koe05] H. Koehler. A contraction algorithm for finding minimal feedback sets. In *Proc. 28th Australasian Computer Science Conference (ACSC)*, pages 165–174, 2005.

[KOS91] A. Kaul, M. A. O'Connor, and V. Srinivasan. Computing Minkowski sums of regular polygons. In *Proc. 3rd Canad. Conf. Comput. Geom.*, pages 74–77, 1991.

[KP98] J. Kececioglu and J. Pecqueur. Computing maximum-cardinality matchings in sparse general graphs. In *Proc. 2nd Workshop on Algorithm Engineering*, pages 121–132, 1998.

[KPP04] H. Kellerer, U. Pferschy, and P. Pisinger. *Knapsack Problems*. Springer, 2004.

[KR87] R. Karp and M. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Research and Development*, 31:249–260, 1987.

[KR91] A. Kanevsky and V. Ramachandran. Improved algorithms for graph four-connectivity. *J. Comp. Sys. Sci.*, 42:288–306, 1991.

[Kru56] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. of the American Mathematical Society*, 7:48–50, 1956.

[KS74] D.E. Knuth and J.L. Szwarcfiter. A structured program to generate all topological sorting arrangements. *Information Processing Letters*, 2:153–157, 1974.

[KS85] M. Keil and J. R. Sack. *Computational Geometry: Minimum decomposition of geometric objects*, pages 197–216. North-Holland, 1985.

[KS86] D. Kirkpatrick and R. Siedel. The ultimate planar convex hull algorithm? *SIAM J. Computing*, 15:287–299, 1986.

[KS90] K. Kedem and M. Sharir. An efficient motion planning algorithm for a convex rigid polygonal object in 2-dimensional polygonal space. *Discrete and Computational Geometry*, 5:43–75, 1990.

[KS99] D. Kreher and D. Stinson. *Combinatorial Algorithms: Generation, Enumeration, and Search*. CRC Press, 1999.

[KS02] M. Keil and J. Snoeyink. On the time bound for convex decomposition of simple polygons. *Int. J. Comput. Geometry Appl.*, 12:181–192, 2002.

[KS05a] H. Kaplan and N. Shafrir. The greedy algorithm for shortest superstrings. *Info. Proc. Letters*, 93:13–17, 2005.

[KS05b] J. Kelner and D. Spielman. A randomized polynomial-time simplex algorithm for linear programming. *Electronic Colloquim on Computational Complexity*, 156:17, 2005.

[KS07] H. Kautz and B. Selman. The state of SAT. *Disc. Applied Math.*, 155:1514–1524, 2007.

[KSB05] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J. ACM*, 2005.

[KSBD07] H. Kautz, B. Selman, R. Brachman, and T. Dietterich. *Satisfiability Testing*. Morgan and Claypool, 2007.

[KSPP03] D Kim, J. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *Proc. 14th Symp. Combinatorial Pattern Matching (CPM)*, pages 186–199, 2003.

[KST93] J. Köbler, U. Schöning, and J. Túran. *The Graph Isomorphism Problem: its structural complexity*. Birhauser, Boston, 1993.

[KSV97] D. Keyes, A. Sameh, and V. Venkatarishnan. *Parallel Numerical Algorithms*. Springer, 1997.

[KT06] J. Kleinberg and E. Tardos. *Algorithm Design*. Addison Wesley, 2006.

[Kuh75] H. W. Kuhn. Steiner's problem revisited. In G. Dantzig and B. Eaves, editors, *Studies in Optimization*, pages 53–70. Mathematical Association of America, 1975.

[Kur30] K. Kuratowski. Sur le problème des courbes gauches en topologie. *Fund. Math.*, 15:217–283, 1930.

[KW01] M. Kaufmann and D. Wagner. *Drawing Graphs: Methods and Models*. Springer-Verlag, 2001.

[Kwa62] M. Kwan. Graphic programming using odd and even points. *Chinese Math.*, 1:273–277, 1962.

[LA04] J. Leung and J. Anderson, editors. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC/Chapman-Hall, 2004.

[LA06] J. Lien and N. Amato. Approximate convex decomposition of polygons. *Computational Geometry: Theory and Applications*, 35:100–123, 2006.

[Lam92] J.-L. Lambert. Sorting the sums $(x_i + y_j)$ in $o(n^2)$ comparisons. *Theoretical Computer Science*, 103:137–141, 1992.

[Lat91] J.-C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, Boston, 1991.

[Lau98] J. Laumond. *Robot Motion Planning and Control*. Springer-Verlag, Lectures Notes in Control and Information Sciences 229, 1998.

[LaV06] S. LaValle. *Planning Algorithms*. Cambridge University Press, 2006.

[Law76] E. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart, and Winston, Fort Worth TX, 1976.

[LD03] R. Laycock and A. Day. Automatically generating roof models from building footprints. In *Proc. 11th Int. Conf. Computer Graphics, Visualization and Computer Vision (WSCG)*, 2003.

[Lec95] T. Lecroq. Experimental results on string matching algorithms. *Software – Practice and Experience*, 25:727–765, 1995.

[Lee82] D. T. Lee. Medial axis transformation of a planar shape. *IEEE Trans. Pattern Analysis and Machine Intelligence*, PAMI-4:363–369, 1982.

[Len87a] T. Lengauer. Efficient algorithms for finding minimum spanning forests of hierarchically defined graphs. *J. Algorithms*, 8, 1987.

[Len87b] H. W. Lenstra. Factoring integers with elliptic curves. *Annals of Mathematics*, 126:649–673, 1987.

[Len89] T. Lengauer. Hierarchical planarity testing algorithms. *J. ACM*, 36(3):474–509, July 1989.

[Len90] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. Wiley, Chichester, England, 1990.