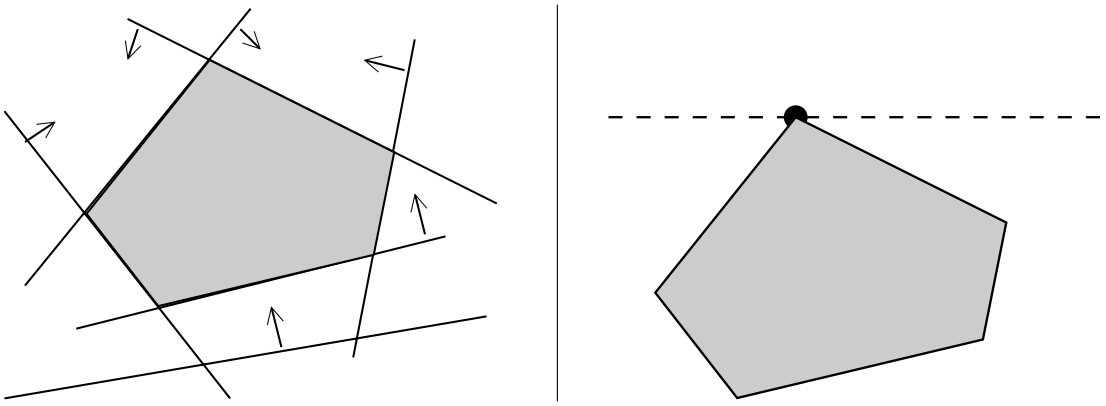


INPUT OUTPUT

13.6 Linear Programming

Input description: A set S of n linear inequalities on m variables

$$S_i := \sum_{j=1}^{m} c_{ij} \cdot x_j \geq b_i,\ 1 \leq i \leq n$$

and a linear optimization function $f(X) = \sum_{j=1}^{m} c_j \cdot x_j$.

Problem description: Which variable assignment X' maximizes the objective function f while satisfying all inequalities S?

Discussion: Linear programming is the most important problem in mathematical optimization and operations research. Applications include:

- *Resource allocation* – We seek to invest a given amount of money to maximize our return. Often our possible options, payoffs, and expenses can be expressed as a system of linear inequalities such that we seek to maximize our possible profit given the constraints. Very large linear programming problems are routinely solved by airlines and other corporations.
- *Approximating the solution of inconsistent equations* – A set of m linear equations on n variables x_i, $1 \leq i \leq n$ is overdetermined if $m > n$. Such overdetermined systems are often *inconsistent*, meaning that there does not exist an assignment to the variables that simultaneously solves all the equations. To find the assignment that best fits the equations, we can replace each variable x_i by $x'_i + \epsilon_i$ and solve the new system as a linear program, minimizing the sum of the error terms.

- *Graph algorithms* – Many of the graph problems described in this book, including shortest path, bipartite matching, and network flow, can be solved as special cases of linear programming. Most of the rest, including traveling salesman, set cover, and knapsack, can be solved using *integer linear programming*.

The *simplex method* is the standard algorithm for linear programming. Each constraint in a linear programming problem acts like a knife that carves away a region from the space of possible solutions. We seek the point within the remaining region that maximizes (or minimizes) $f(X)$. By appropriately rotating the solution space, the optimal point can always be made to be the highest point in the region. The region (simplex) formed by the intersection of a set of linear constraints is convex, so unless we are at the top there is always a higher vertex neighboring any starting point. When we cannot find a higher neighbor to walk to, we have reached the optimal solution.

While the simplex algorithm is not too complex, there is considerable art to producing an efficient implementation capable of solving large linear programs. Large programs tend to be sparse (meaning that most inequalities use few variables), so sophisticated data structures must be used. There are issues of numerical stability and robustness, as well as choosing which neighbor we should walk to next (so-called *pivoting rules*). There also exist sophisticated *interior-point* methods, which cut through the interior of the simplex instead of walking along the outside, and beat simplex in many applications.

The bottom line on linear programming is this: you are much better off using an existing LP code than writing your own. Further, you are probably better off paying money than surfing the Web. Linear programming is an algorithmic problem of such economic importance that commercial implementations are far superior to free versions.

Issues that arise in linear programming include:

- *Do any variables have integrality constraints?* – It is impossible to send 6.54 airplanes from New York to Washington each business day, even if that value maximizes profit according to your model. Such variables often have natural integrality constraints. A linear program is called an *integer program* if all its variables have integrality constraints, or a *mixed integer program* if some of them do.

  Unfortunately, it is NP-complete to solve integer or mixed programs to optimality. However, there are integer programming techniques that work reasonably well in practice. *Cutting plane techniques* solve the problem first as a linear program, and then add extra constraints to enforce integrality around the optimal solution point before solving it again. After sufficiently many iterations, the optimum point of the resulting linear program matches that of the original integer program. As with most exponential-time algorithms,

run times for integer programming depend upon the difficulty of the problem instance and are unpredictable.

- *Do I have more variables or constraints?* – Any linear program with m variables and n inequalities can be written as an equivalent *dual* linear program with n variables and m inequalities. This is important to know, because the running time of a solver might be quite different on the two formulations. In general, linear programs (LPs) with much more variables than constraints should be solved directly. If there are many more constraints than variables, it is usually better to solve the dual LP or (equivalently) apply the dual simplex method to the primal LP.

- *What if my optimization function or constraints are not linear?* – In least-squares curve fitting, we seek the line that best approximates a set of points by minimizing the sum of squares of the distance between each point and the line. In formulating this as a mathematical program, the natural objective function is no longer linear, but quadratic. Although fast algorithms exist for least squares fitting, general *quadratic programming* is NP-complete.

  There are three possible courses of action when you must solve a nonlinear program. The best is if you can model it in some other way, as is the case with least-squares fitting. The second is to try to track down special codes for quadratic programming. Finally, you can model your problem as a constrained or unconstrained optimization problem and try to solve it with the codes discussed in Section 13.5 (page 407).

- *What if my model does not match the input format of my LP solver?* – Many linear programming implementations accept models only in so-called *standard form*, where all variables are constrained to be nonnegative, the object function must be minimized, and all constraints must be equalities (instead of inequalities).

  Do not fear. There exist standard transformations to map arbitrary LP models into standard form. To convert a maximization problem to a minimization one, simply multiply each coefficient of the objective function by -1. The remaining problems can be solved by adding *slack variables* to the model. See any textbook on linear programming for details. Modeling languages such as AMPC can provide a nice interface to your solver and deal with these issues for you.

Implementations: The USENET Frequently Asked Question (FAQ) list is a very useful resource on solving linear programs. In particular, it provides a list of available codes with descriptions of experiences. Check it out at *http://www-unix.mcs.anl.gov/otc/Guide/faq/*.

There are at least three reasonable choices in free LP-solvers. `Lp_solve`, written in ANSI C by Michel Berkelaar, can also handle integer and mixed-integer

problems. It is available at *http://lpsolve.sourceforge.net/5.5/*, and a substantial user community exists. The simplex solver `CLP` produced under the Computational Infrastructure for Operations Research is available (with other optimization software) at *http://www.coin-or.org/*. Finally, the GNU Linear Programming Kit (GLPK) is intended for solving large-scale linear programming, mixed integer programming (MIP), and other related problems. It is available at *http://www.gnu.org/software/glpk/*. In recent benchmarks among free codes (see *http://plato.asu.edu/bench.html*), `CLP` appeared to be fastest on on linear programming problems and `lp_solve` on mixed integer problems.

NEOS (Network-Enabled Optimization System) provides an opportunity to solve your problem on computers and software at Argonne National Laboratory. Linear programming and unconstrained optimization are both supported. This is worth checking out at *http://www.mcs.anl.gov/home/otc/Server/* if you need an answer instead of a program.

Algorithm 551 [Abd80] and Algorithm 552 [BR80] of the *Collected Algorithms of the ACM* are simplex-based codes for solving overdetermined systems of linear equations in Fortran. See Section 19.1.5 (page 659) for details.

Notes: The need for optimization via linear programming arose in logistics problems in World War II. The simplex algorithm was invented by George Danzig in 1947 [Dan63]. Klee and Minty [KM72] proved that the simplex algorithm is exponential in worst case, but it is very efficient in practice.

Smoothed analysis measures the complexity of algorithms assuming that their inputs are subject to small amounts of random noise. Carefully constructed worst-case instances for many problems break down under such perturbations. Spielman and Teng [ST04] used smoothed analysis to explain the efficiency of simplex in practice. Recently, Kelner and Spielman developed a randomized simplex algorithm running in polynomial time [KS05b].

Khachian's ellipsoid algorithm [Kha79] first proved that linear programming was polynomial in 1979. Karmarkar's algorithm [Kar84] is an interior-point method that has proven to be both a theoretical and practical improvement of the ellipsoid algorithm, as well as a challenge for the simplex method. Good expositions on the simplex and ellipsoid algorithms for linear programming include [Chv83, Gas03, MG06].

Semidefinite programming deals with optimization problems over symmetric positive semidefinite matrix variables, with linear cost function and linear constraints. Important special cases include linear programming and convex quadratic programming with convex quadratic constraints. Semidefinite programming and its applications to combinatorial optimization problems are surveyed in [Goe97, VB96].

Linear programming is P-complete under log-space reductions [DLR79]. This makes it unlikely to have an NC parallel algorithm, where a problem is in NC iff it can be solved on a PRAM in polylogarithmic time using a polynomial number of processors. Any problem that is P-complete under log-space reduction cannot be in NC unless P=NC. See [GHR95] for a thorough exposition of the theory of P-completeness, including an extensive list of P-complete problems.

Related Problems: Constrained and unconstrained optimization (see page 407), network flow (see page 509).

13.7 Random Number Generation

Input description: Nothing, or perhaps a seed.

Problem description: Generate a sequence of random integers.

Discussion: Random numbers have an enormous variety of interesting and important applications. They form the foundation of simulated annealing and related heuristic optimization techniques. Discrete event simulations run on streams of random numbers, and are used to model everything from transportation systems to casino poker. Passwords and cryptographic keys are typically generated randomly. Randomized algorithms for graph and geometric problems are revolutionizing these fields and establishing randomization as one of the fundamental ideas of computer science.

Unfortunately, generating random numbers looks a lot easier than it really is. Indeed, it is fundamentally impossible to produce truly random numbers on any deterministic device. Von Neumann [Neu63] said it best: "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin." The best we can hope for are *pseudorandom* numbers, a stream of numbers that appear as if they were generated randomly.

There can be serious consequences to using a bad random-number generator. In one famous case, a Web browser's encryption scheme was broken with the discovery that the seeds of its random-number generator employed too few random bits [GW96]. Simulation accuracy is regularly compromised or invalidated by poor random number generation. This is an area where people shouldn't mess around, but they do. Issues to think about include:

- *Should my program use the same "random" numbers each time it runs?* – A poker game that deals you the exact same hand each time you play quickly loses interest. One common solution is to use the lower-order bits of the machine clock as the *seed* or starting point for a stream of random numbers, so that each time the program runs it does something different.

Such methods are adequate for games, but not for serious simulations. There are liable to be periodicities in the distribution of random numbers whenever calls are made in a loop. Also, debugging is seriously complicated when program results are not repeatable. Should your program crash, you cannot go back and discover why. A possible compromise is to use a deterministic pseudorandom-number generator, but write the current seed to a file between runs. During debugging, this file can be overwritten with a fixed initial value of the seed.

- *How good is my compiler's built-in random number generator?* – If you need uniformly-generated random numbers, and won't be betting the farm on the accuracy of your simulation, my recommendation is simply to use what your compiler provides. Your best opportunity to mess things up is with a bad choice of starting seed, so read the manual for its recommendations.

  If you *are* going to bet the farm on the results of your simulation, you had better test your random number generator. Be aware that it is very difficult to eyeball the results and decide whether the output is really random. This is because people have very skewed ideas of how random sources should behave and often see patterns that don't really exist. Several different tests should be used to evaluate a random number generator, and the statistical significance of the results established. The National Institute of Standards and Technology (NIST) has developed a test suite for evaluating random number generators, discussed below.

- *What if I must implement my own random-number generator?* – The standard algorithm of choice is the *linear congruential generator*. It is fast, simple, and (if instantiated with the right constants) gives reasonable pseudorandom numbers. The nth random number R_n is a function of the $(n-1)$st random number:

  $$R_n = (aR_{n-1} + c) \bmod m$$

  In theory, linear congruential generators work the same way roulette wheels do. The long path of the ball around and around the wheel (captured by $aR_{n-1} + c$) ends in one of a relatively small number of bins, the choice of which is extremely sensitive to the length of the path (captured by the mod m-truncation).

  A substantial theory has been developed to select the constants a, c, m, and R_0. The period length is largely a function of the modulus m, which is typically constrained by the word length of the machine.

  Note that the stream of numbers produced by a linear congruential generator repeats the instant the first number repeats. Further, computers are fast enough to make 2^{32} calls to a random-number generator in a few minutes. Thus, any 32-bit linear congruential generator is in danger of cycling, motivating generators with significantly longer periods.

- *What if I don't want such large random numbers?* – The linear congruential generator produces a uniformly-distributed sequence of large integers that can be easily scaled to produce other uniform distributions. For uniformly distributed real numbers between 0 and 1, use R_i/m. Note that 1 cannot be realized this way, although 0 can. If you want uniformly distributed integers between l and h, use $\lfloor l + (h - l + 1)R_i/m \rfloor$.

- *What if I need nonuniformly distributed random numbers?* – Generating random numbers according to a given nonuniform distribution can be a tricky business. The most reliable way to do this correctly is the acceptance-rejection method. We can bound the desired geometric region to sample from by a box and then select a random point p from the box. This point can be generated by selecting the x and y coordinates independently at random. If it lies within the region of interest, we can return p as being selected at random. Otherwise we throw it away and repeat with another random point. Essentially, we throw darts at random and report those that hit the target.

  This method is correct, but it can be slow. If the volume of the region of interest is small relative to that of the box, most of our darts will miss the target. Efficient generators for Gaussian and other special distributions are described in the references and implementations below.

  Be cautious about inventing your own technique, however, since it can be tricky to obtain the right probability distribution. For example, an *incorrect* way to select points uniformly from a circle of radius r would be to generate polar coordinates by selecting an angle from 0 to 2π and a displacement between 0 and r—both uniformly at random. In such a scheme, half the generated points will lie within $r/2$ of the center, when only one-fourth of them should be! This is different enough to seriously skew the results, while being sufficiently subtle that it can easily escape detection.

- *How long should I run my Monte Carlo simulation to get the best results?* – The longer you run a simulation, the more accurately the results should approximate the limiting distribution, thus increasing accuracy. However, this is true only until you exceed the *period*, or cycle length, of your random-number generator. At that point, your sequence of random numbers repeats itself, and further runs generate no additional information.

  Instead of jacking up the length of a simulation run to the max, it is usually more informative to do many shorter runs (say 10 to 100) with different seeds and then consider the range of results you see. The variance provides a healthy measure of the degree to which your results are repeatable. This exercise corrects the natural tendency to see a simulation as giving "the" correct answer.

Implementations: See *http://random.mat.sbg.ac.at* for an excellent website on random-number generation and stochastic simulation. It includes pointers to papers and literally dozens of implementations of random-number generators.

Parallel simulations make special demands on random-number generators. How can we ensure that random streams are independent on each machine? L'Ecuyer et.al. [LSCK02] provide object-oriented generators with a period length of approximately 2^{191}. Implementations in C, C++, and Java are available at *http://www.iro.umontreal.ca/~lecuyer/myftp/streams00/*. Independent streams of random numbers are supported for parallel applications. Another possibility is the *Scalable Parallel Random Number Generators Library (SPRNG)* [MS00], available at *http://sprng.cs.fsu.edu/*.

Algorithms 488 [Bre74], 599 [AKD83], and 712 [Lev92] of the *Collected Algorithms of the ACM* are Fortran codes for generating non-uniform random numbers according to several probability distributions, including the normal, exponential, and Poisson distributions. They are available from Netlib (see Section 19.1.5 (page 659)).

The National Institute of Standards [RSN+01] has prepared an extensive statistical test suite to validate random number generators. Both the software and the report describing it are available at *http://csrc.nist.gov/rng/*.

True random-number generators extract random bits by observing physical processes. The website *http://www.random.org* makes available random numbers derived from atmospheric noise that passes the NIST statistical tests. This is an amusing solution if you need a small quantity of random numbers (say, to run a lottery) instead a random-number generator.

Notes: Knuth [Knu97b] provides a thorough treatment of random-number generation, which I heartily recommend. He presents the theory behind several methods, including the middle-square and shift-register methods we have not described here, as well as a detailed discussion of statistical tests for validating random-number generators.

That said, see [Gen04] for more recent developments in random number generation. The Mersenne twister [MN98] is a fast random number generator of period $2^{19937} - 1$. Other modern methods include [Den05, PLM06]. Methods for generating nonuniform random variates are surveyed in [HLD04]. Comparisons of different random-number generators in practice include [PM88].

Tables of random numbers appear in most mathematical handbooks as relics from the days before there was ready access to computers. Most notable is [RC55], which provides one million random digits.

The deep relationship between randomness, information, and compressibility is explored within the theory of Kolmogorov complexity, which measures the complexity of a string by its compressibility. Truly random strings are incompressible. The string of seemingly random digits of π cannot be random under this definition, since the entire sequence is defined by any program implementing a series expansion for π. Li and Vitáni [LV97] provide a thorough introduction to the theory of Kolmogorov complexity.

Related Problems: Constrained and unconstrained optimization (see page 407), generating permutations (see page 448), generating subsets (see page 452), generating partitions (see page 456).

INPUT	OUTPUT
833816926455584605284210207 1	179424673 2038074743 * 22801763489 833816926455584605284210207 1

13.8 Factoring and Primality Testing

Input description: An integer n.

Problem description: Is n a prime number, and if not what are its factors?

Discussion: The dual problems of integer factorization and primality testing have surprisingly many applications for a problem long suspected of being only of mathematical interest. The security of the RSA public-key cryptography system (see Section 18.6 (page 641)) is based on the computational intractability of factoring large integers. As a more modest application, hash table performance typically improves when the table size is a prime number. To get this benefit, an initialization routine must identify a prime near the desired table size. Finally, prime numbers are just interesting to play with. It is no coincidence that programs to generate large primes often reside in the games directory of UNIX systems.

Factoring and primality testing are clearly related problems, although they are quite different algorithmically. There exist algorithms that can demonstrate that an integer is *composite* (i.e. , not prime) without actually giving the factors. To convince yourself of the plausibility of this, note that you can demonstrate the compositeness of any nontrivial integer whose last digit is 0, 2, 4, 5, 6, or 8 without doing the actual division.

The simplest algorithm for both of these problems is brute-force trial division. To factor n, compute the remainder of n/i for all $1 < i \leq \sqrt{n}$. The prime factorization of n will contain at least one instance of every i such that $n/i = \lfloor n/i \rfloor$, unless n is prime. Make sure you handle the multiplicities correctly, and account for any primes larger than $\sqrt{n}$.

Such algorithms can be sped up by using a precomputed table of small primes to avoid testing all possible i. Surprisingly large numbers of primes can be represented in surprisingly little space by using bit vectors (see Section 12.5 (page 385)). A bit vector of all odd numbers less than 1,000,000 fits in under 64 kilobytes. Even tighter encodings become possible by eliminating all multiples of 3 and other small primes.

Although trial division runs in $O(\sqrt{n})$ time, it is *not* a polynomial-time algorithm. The reason is that it only takes $\lg_2 n$ bits to represent n, so trial division takes time exponential in the input size. Considerably faster (but still exponential time) factoring algorithms exist, whose correctness depends upon more substantial number theory. The fastest known algorithm, the *number field sieve*, uses randomness to construct a system of congruences—the solution of which usually gives a factor of the integer. Integers with as many as 200 digits (663 bits) have been factored using this method, although such feats require enormous amounts of computation.

Randomized algorithms make it much easier to test whether an integer is prime. Fermat's little theorem states that $a^{n-1} = 1 (\text{mod } n)$ for all a not divisible by n, provided n is prime. Suppose we pick a random value $1 \leq a < n$ and compute the residue of $a^{n-1} (\text{mod } n)$. If this residue is not 1, we have just proven that n cannot be prime. Such randomized primality tests are very efficient. PGP (see Section 18.6 (page 641)) finds 300+ digit primes using hundreds of these tests in minutes, for use as cryptographic keys.

Although the primes are scattered in a seemingly random way throughout the integers, there is some regularity to their distribution. The *prime number theorem* states that the number of primes less than n (commonly denoted by $\pi(n)$) is approximately $n/\ln n$. Further, there never are large gaps between primes, so in general, one would expect to examine about $\ln n$ integers if one wanted to find the first prime larger than n. This distribution and the fast randomized primality test explain how PGP can find such large primes so quickly.

Implementations: Several general systems for computational number theory are available. PARI is capable of handling complex number-theoretic problems on arbitrary-precision integers (to be precise, limited to 80,807,123 digits on 32-bit machines), as well as reals, rationals, complex numbers, polynomials, and matrices. It is written mainly in C, with assembly code for inner loops on major architectures, and includes more than 200 special predefined mathematical functions. PARI can be used as a library, but it also possesses a calculator mode that gives instant access to all the types and functions. PARI is available at *http://pari.math.u-bordeaux.fr/*

LiDIA (*http://www.cdc.informatik.tu-darmstadt.de/TI/LiDIA/*) is the acronym for the C++ *Library for Computational Number Theory*. It implements several of the modern integer factorization methods.

A Library for doing Number Theory (NTL) is a high-performance, portable C++ library providing data structures and algorithms for manipulating signed, arbitrary length integers, and for vectors, matrices, and polynomials over the integers and over finite fields. It is available at *http://www.shoup.net/ntl/*.

Finally, MIRACL (Multiprecision Integer and Rational Arithmetic C/C++ Library) implements six different integer factorization algorithms, including the quadratic sieve. It is available at *http://www.shamus.ie/*.

Notes: Expositions on modern algorithms for factoring and primality testing include Crandall and Pomerance [CP05] and Yan [Yan03]. More general surveys of computational number theory include Bach and Shallit [BS96] and Shoup [Sho05].

In 2002, Agrawal, Kayal, and Saxena [AKS04] solved a long-standing open problem by giving the first polynomial-time deterministic algorithm to test whether an integer is composite. Their algorithm, which is surprisingly elementary for such an important result, involves a careful analysis of techniques from earlier randomized algorithms. Its existence serves as somewhat of a rebuke to researchers (like me) who shy away from classical open problems due to fear. Dietzfelbinger [Die04] provides a self-contained treatment of this result.

The Miller-Rabin [Mil76, Rab80] randomized primality testing algorithm eliminates problems with Carmichael numbers, which are composite integers that always satisfy Fermat's theorem. The best algorithms for integer factorization include the quadratic-sieve [Pom84] and the elliptic-curve methods [Len87b].

Mechanical sieving devices provided the fastest way to factor integers surprisingly far into the computing era. See [SWM95] for a fascinating account of one such device, built during World War I. Hand-cranked, it proved the primality of $2^{31} - 1$ in 15 minutes of sieving time.

An important problem in computational complexity theory is whether P = NP $\cap$ co-NP. The decision problem "is n a composite number?" used to be the best candidate for a counterexample. By exhibiting the factors of n, it is trivially in NP. It can be shown to be in co-NP, since every prime has a short proof of its primality [Pra75]. The recent proof that composite numbers testing is in P shot down this line of reasoning. For more information on complexity classes, see [GJ79, Joh90].

The integer RSA-129 was factored in eight months using over 1,600 computers. This was particularly noteworthy because in the original RSA paper [RSA78] they had originally predicted such a factorization would take 40 quadrillion years using 1970s technology. Bahr, Boehm, Franke, and Kleinjung hold the current integer factorization record, with their successful attack on the 200-digit integer RSA-200 in May 2005. This required the equivalent of 55 years of computations on a single 2.2 GHz Opteron CPU.

Related Problems: Cryptography (see page 641), high precision arithmetic (see page 423).

INPUT	OUTPUT
$\frac{495782912874914951515089054258695 78}{7436743693123724272726335813880 4367}$	$\frac{2}{3}$

13.9 Arbitrary-Precision Arithmetic

Input description: Two very large integers, x and y.

Problem description: What is $x + y$, $x - y$, $x \times y$, and x/y?

Discussion: Any programming language rising above basic assembler supports single- and perhaps double-precision integer/real addition, subtraction, multiplication, and division. But what if we wanted to represent the national debt of the United States in pennies? One trillion dollars worth of pennies requires 15 decimal digits, which is far more than can fit into a 32-bit integer.

Other applications require *much* larger integers. The RSA algorithm for public-key cryptography recommends integer keys of at least 1000 digits to achieve adequate security. Experimenting with number-theoretic conjectures for fun or research requires playing with large numbers. I once solved a minor open problem [GKP89] by performing an exact computation on the integer $\binom{5906}{2953} \approx 9.93285 \times 10^{1775}$.

What should you do when you need large integers?

- *Am I solving a problem instance requiring large integers, or do I have an embedded application?* – If you just need the answer to a specific problem with large integers, such as in the number theory application above, you should consider using a computer algebra system like Maple or Mathematica. These provide arbitrary-precision arithmetic as a default and use nice Lisp-like programming languages as a front end—together often reducing your problem to a 5- to 10- line program.

  If you have an embedded application requiring high-precision arithmetic instead, you should use an existing arbitrary precision math library. You are likely to get additional functions beyond the four basic operations for computing things like greatest common divisor in the bargain. See the Implementations section for details.

- *Do I need high- or arbitrary-precision arithmetic?* – Is there an upper bound on how big your integers can get, or do you really need *arbitrary*-precision—i.e. , unbounded. This determines whether you can use a fixed-length array to represent your integers as opposed to a linked-list of digits. The array is likely to be simpler and will not prove a constraint in most applications.

- *What base should I do arithmetic in?* – It is perhaps simplest to implement your own high-precision arithmetic package in decimal, and thus represent each integer as a string of base-10 digits. However, it is far more efficient to use a higher base, ideally equal to the square root of the largest integer supported fully by hardware arithmetic.

  Why? The higher the base, the fewer digits we need to represent a number. Compare 64 decimal with 1000000 binary. Since hardware addition usually takes one clock cycle independent of value of the actual numbers, best performance is achieved using the highest supported base. The factor limiting us to base $b = \sqrt{maxint}$ is the desire to avoid overflow when multiplying two of these "digits" together.

  The primary complication of using a larger base is that integers must be converted to and from base-10 for input and output. The conversion is easily performed once all four high-precision arithmetical operations are supported.

- *How low-level are you willing to get for fast computation?* – Hardware addition is much faster than a subroutine call, so you take a significant hit on speed using high-precision arithmetic when low-precision arithmetic suffices. High-precision arithmetic is one of few problems in this book where inner loops in assembly language prove the right idea to speed things up. Similarly, using bit-level masking and shift operations instead of arithmetical operations can be a win if you really understand the machine integer representation.

The algorithm of choice for each of the five basic arithmetic operations is as follows:

- *Addition* – The basic schoolhouse method of lining up the decimal points and then adding the digits from right to left with "carries" runs to time linear in the number of digits. More sophisticated carry-look-ahead parallel algorithms are available for low-level hardware implementation. They are presumably used on your microprocessor for low-precision addition.

- *Subtraction* – By fooling with the sign bits of the numbers, subtraction can be a special considered case of addition: $(A - (-B)) = (A + B)$. The tricky part of subtraction is performing the "borrow." This can be simplified by always subtracting from the number with the larger absolute value and adjusting the signs afterwards, so we can be certain there will always be something to borrow from.

- *Multiplication* – Repeated addition will take exponential time on large integers, so stay away from it. The digit-by-digit schoolhouse method is reasonable to program and will work much better, presumably well enough for your application. On very large integers, Karatsuba's $O(n^{1.59})$ divide-and-conquer algorithm wins. Dan Grayson, author of Mathematica's arbitrary-precision arithmetic, found that the switch-over happened at well under 100 digits.

Even faster for very large integers is an algorithm based on Fourier transforms. Such algorithms are discussed in Section 13.11 (page 431).

- *Division* – Repeated subtraction will take exponential time, so the easiest reasonable algorithm to use is the long-division method you hated in school. This is fairly complicated, requiring arbitrary-precision multiplication and subtraction as subroutines, as well as trial-and-error, to determine the correct digit at each position of the quotient.

  In fact, integer division can be reduced to integer multiplication, although in a nontrivial way, so if you are implementing asymptotically fast multiplication, you can reuse that effort in long division. See the references below for details.

- *Exponentiation* – We can compute a^n using $n-1$ multiplications, by computing $a \times a \times \ldots \times a$. However, a much better divide-and-conquer algorithm is based on the observation that $n = \lfloor n/2 \rfloor + \lceil n/2 \rceil$. If n is even, then $a^n = (a^{n/2})^2$. If n is odd, then $a^n = a(a^{\lfloor n/2 \rfloor})^2$. In either case, we have halved the size of our exponent at the cost of at most two multiplications, so $O(\lg n)$ multiplications suffice to compute the final value:

```
function power(a, n)
    if (n = 0) return(1)
    x = power(a, ⌊n/2⌋)
    if (n is even) then return(x^2)
        else return(a × x^2)
```

High- but not arbitrary-precision arithmetic can be conveniently performed using the Chinese remainder theorem and modular arithmetic. The *Chinese remainder theorem* states that an integer between 1 and $P = \prod_{i=1}^{k} p_i$ is uniquely determined by its set of residues mod p_i, where each p_i, p_j are relatively prime integers. Addition, subtraction, and multiplication (but not division) can be supported using such residue systems, with the advantage that large integers can be manipulated without complicated data structures.

Many of these algorithms for computations on long integers can be directly applied to computations on polynomials. See the references for more details. A particularly useful algorithm is Horner's rule for fast polynomial evaluation. When $P(x) = \sum_{i=0}^{n} c_i \cdot x^i$ is blindly evaluated term by term, $O(n^2)$ multiplications will be performed. Much better is observing that $P(x) = c_0 + x(c_1 + x(c_2 + x(c_3 + \ldots)))$, the evaluation of which uses only a linear number of operations.

Implementations: All major commercial computer algebra systems incorporate high-precision arithmetic, including Maple, Mathematica, Axiom, and Macsyma. If you have access to one of these, this is your best option for a quick, nonembedded

application. The rest of this section focuses on source code available for embedded applications.

The premier C/C++ library for fast, arbitrary-precision is the GNU Multiple Precision Arithmetic Library (GMP), which operates on signed integers, rational numbers, and floating point numbers. It is widely used and well-supported, and available at *http://gmplib.org/*.

The `java.math BigInteger` class provides arbitrary-precision analogues to all of Java's primitive integer operators. `BigInteger` provides additional operations for modular arithmetic, GCD calculation, primality testing, prime generation, bit manipulation, and a few other miscellaneous operations.

A lower-performance, less-tested, but more personal implementation of high-precision arithmetic appears in the library from my book *Programming Challenges* [SR03]. See Section 19.1.10 (page 661) for details.

Several general systems for computational number theory are available. Each of these supports operations of arbitrary-precision integers. Information about the PARI, LiDIA, NTL and MIRACL number-theoretic libraries can be found in Section 13.8 (page 420).

ARPREC is a C++/Fortran-90 arbitrary precision package with an associated interactive calculator. MPFUN90 is a similar package written exclusively in Fortran-90. Both are available at *http://crd.lbl.gov/∼dhbailey/mpdist/*. Algorithm 693 [Smi91] of the *Collected Algorithms of the ACM* is a Fortran implementation of floating-point, multiple-precision arithmetic. See Section 19.1.5 (page 659).

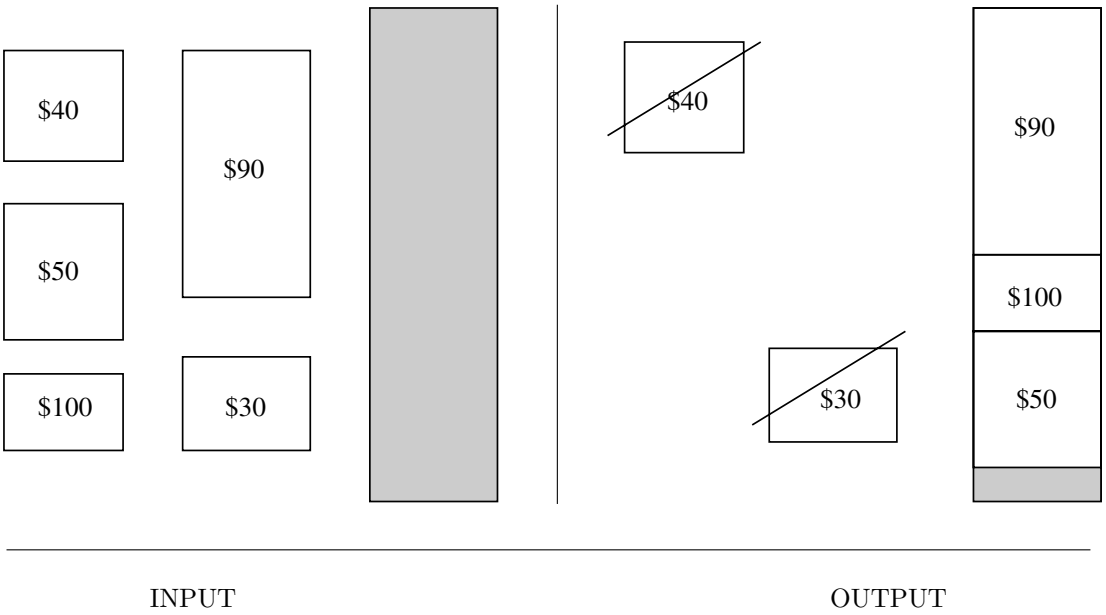
Notes: Knuth [Knu97b] is the primary reference on algorithms for all basic arithmetic operations, including implementations of them in the MIX assembly language. Bach and Shallit [BS96] and Shoup [Sho05] provide more recent treatments of computational number theory.

Expositions on the $O(n^{1.59})$-time divide-and-conquer algorithm for multiplication [KO63] include [AHU74, Man89]. An FFT-based algorithm multiplies two n-bit numbers in $O(n \lg n \lg \lg n)$ time and is due to Schönhage and Strassen [SS71]. Expositions include [AHU74, Knu97b]. The reduction between integer division and multiplication is presented in [AHU74, Knu97b]. Applications of fast multiplication to other arithmetic operations are presented by Bernstein [Ber04b]

Good expositions of algorithms for modular arithmetic and the Chinese remainder theorem include [AHU74, CLRS01]. A good exposition of circuit-level algorithms for elementary arithmetic algorithms is [CLRS01].

Euclid's algorithm for computing the greatest common divisor of two numbers is perhaps the oldest interesting algorithm. Expositions include [CLRS01, Man89].

Related Problems: Factoring integers (see page 420), cryptography (see page 641).



13.10 Knapsack Problem

Input description: A set of items $S = \{1, \ldots, n\}$, where item i has size s_i and value v_i. A knapsack capacity is C.

Problem description: Find the subset $S' \subset S$ that maximizes the value of $\sum_{i \in S'} v_i$, given that $\sum_{i \in S'} s_i \leq C$; i.e. , all the items fit in a knapsack of size C.

Discussion: The knapsack problem arises in resource allocation with financial constraints. How do you select what things to buy given a fixed budget? Everything has a cost and value, so we seek the most value for a given cost. The name *knapsack problem* invokes the image of the backpacker who is constrained by a fixed-size knapsack, and so must fill it only with the most useful and portable items.

The most common formulation is the *0/1 knapsack problem*, where each item must be put entirely in the knapsack or not included at all. Objects cannot be broken up arbitrarily, so it is not fair taking one can of Coke from a six-pack or opening a can to take just a sip. It is this 0/1 property that makes the knapsack problem hard, for a simple greedy algorithm finds the optimal selection when we are allowed to subdivide objects. We compute the "price per pound" for each item, and take the most expensive item or the biggest part thereof until the knapsack is full. Repeat with the next most expensive item. Unfortunately, the 0/1 constraint is usually inherent in most applications.

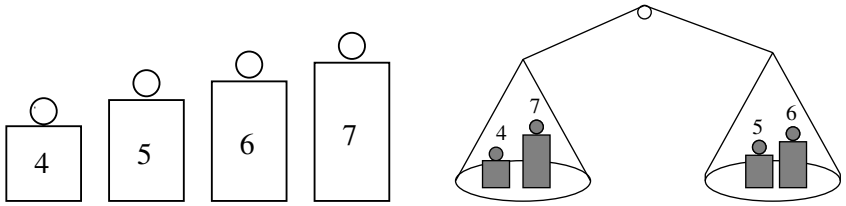


Figure 13.1: Integer partition is a special case of the knapsack problem

Issues that arise in selecting the best algorithm include:

- *Does every item have the same cost/value or the same size?* – When all items are worth exactly the same amount, we maximize our value by taking the greatest number of items. Therefore, the optimal solution is to sort the items in order of increasing size and insert them into the knapsack in this order until no more fit. The problem is similarly solved when each object has the same size but the costs are different. Sort by cost, and take the cheapest elements first. These are the easy cases of knapsack.

- *Does each item have the same "price per pound"?* – In this case, our problem is equivalent to ignoring the price and just trying to minimize the amount of empty space left in the knapsack. Unfortunately, even this restricted version is NP-complete, so we cannot expect an efficient algorithm that always solves the problem. Don't lose hope, however, because knapsack proves to be an "easy" hard problem, and one that can usually be handled with the algorithms described below.

  An important special case of a constant "price-per-pound" knapsack is the *integer partition* problem, presented in cartoon form in Figure 13.1. Here, we seek to partition the elements of S into two sets A and B such that $\sum_{a \in A} a = \sum_{b \in B}$, or alternately make the difference as small as possible. Integer partition can be thought of as bin packing into two equal-sized bins or knapsack with a capacity of half the total weight, so all three problems are closely related and NP-complete.

  The constant "price-per-pound" knapsack problem is often called the *subset sum* problem, because we seek a subset of items that adds up to a specific target number C; i.e. , the capacity of our knapsack.

- *Are all the sizes relatively small integers?* – When the sizes of the items and the knapsack capacity C are all integers, there exists an efficient dynamic programming algorithm to find the optimal solution in time $O(nC)$ and $O(C)$ space. Whether this works for you depends upon how big C is. It is great for $C \leq 1{,}000$, but not so great for $C \geq 10{,}000{,}000$.

The algorithm works as follows: Let S' be a set of items, and let $C[i, S']$ be true if and only if there is a subset of S' whose size adds up exactly to i. Thus, $C[i, \emptyset]$ is false for all $1 \leq i \leq C$. One by one we add a new item s_j to S' and update the affected values of $C[i, S']$. Observe that $C[i, S' \cup s_j] =$ true iff $C[i, S']$ or $C[i - s_j, S']$ is true, since we either use s_j in realizing the sum or we don't. We identify all sums that can be realized by performing n sweeps through all C elements—one for each s_j, $1 \leq j \leq n$—and so updating the array. The knapsack solution is given by the largest index of a true element of the largest realizable size. To reconstruct the winning subset, we must also store the name of the item number that turned $C[i]$ from false to true for each $1 \leq i \leq C$ and then scan backwards through the array.

This dynamic programming formulation ignores the values of the items. To generalize the algorithm, use each element of the array to store the value of the best subset to date summing up to i. We now update when the sum of the cost of $C[i - s_j, S']$ plus the cost of s_j is better than the previous cost of $C[i]$.

- *What if I have multiple knapsacks?* – When there are multiple knapsacks, your problem might be better thought of as a bin-packing problem. Check out Section 17.9 (page 595) for bin-packing/cutting-stock algorithms. That said, algorithms for optimizing over multiple knapsacks are provided in the Implementations section below.

Exact solutions for large capacity knapsacks can be found using integer programming or backtracking. A 0/1 integer variable x_i is used to denote whether item i is present in the optimal subset. We maximize $\sum_{i=1}^{n} x_i \cdot v_i$ given the constraint that $\sum_{i=1}^{n} x_i \cdot s_i \leq C$. Integer programming codes are discussed in Section 13.6 (page 411).

Heuristics must be used when exact solutions prove too costly to compute. The simple greedy heuristic inserts items according to the maximum "price per pound" rule described previously. Often this heuristic solution is close to optimal, but it can be arbitrarily bad depending upon the problem instance. The "price per pound" rule can also be used to reduce the problem size in exhaustive search-based algorithms by eliminating "cheap but heavy" objects from future consideration.

Another heuristic is based on *scaling*. Dynamic programming works well if the knapsack capacity is a reasonably small integer, say $\leq C_s$. But what if we have a problem with capacity $C > C_s$? We scale down the sizes of all items by a factor of C/C_s, round the size down to the nearest integer, and then use dynamic programming on the scaled items. Scaling works well in practice, especially when the range of sizes of items is not too large.

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 the associated book [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*. The strongest code is based on the dynamic programming algorithm of [MPT99].

Algorithm 632 [MT85] of the *Collected Algorithms of the ACM* is a Fortran code for the 0/1 knapsack problem, with the twist that it supports multiple knapsacks. See Section 19.1.5 (page 659).

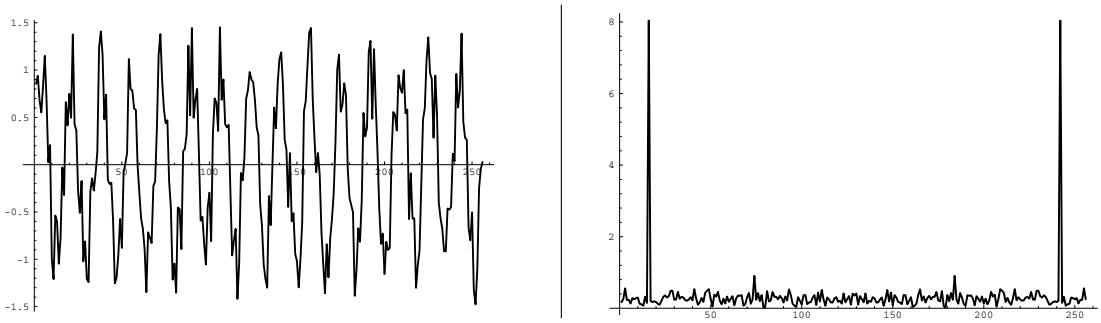
Notes: Keller, Pferschy, and Pisinger [KPP04] is the most current reference on the knapsack problem and variants. Martello and Toth's book [MT90a] and survey article [MT87] are standard references on the knapsack problem, including both theoretical and experimental results. An excellent exposition on integer programming approaches to knapsack problems appears in [SDK83]. See [MPT00] for a computational study of algorithms for 0-1 knapsack problems.

A polynomial-time approximation scheme is an algorithm that approximates the optimal solution of a problem in time polynomial in both its size and the approximation factor ϵ.

This very strong condition implies a smooth tradeoff between running time and approximation quality. Good expositions on polynomial-time approximation schemes [IK75] for knapsack and subset sum includes [BvG99, CLRS01, GJ79, Man89].

The first algorithm for generalized public key encryption by Merkle and Hellman [MH78] was based on the hardness of the knapsack problem. See [Sch96] for an exposition.

Related Problems: Bin packing (see page 595), integer programming (see page 411).



INPUT OUTPUT

13.11 Discrete Fourier Transform

Input description: A sequence of n real or complex values h_i, $0 \leq i \leq n-1$, sampled at uniform intervals from a function h.

Problem description: The discrete Fourier transform $H_m = \sum_{k=0}^{n-1} h_k e^{2\pi ikm/n}$ for $0 \leq m \leq n-1$.

Discussion: Although computer scientists tend to be fairly ignorant about Fourier transforms, electrical engineers and signal processors eat them for breakfast. Functionally, Fourier transforms provide a way to convert samples of a standard time-series into the *frequency domain*. This provides a dual representation of the function in which certain operations become easier than in the time domain. Applications of Fourier transforms include:

- *Filtering* – Taking the Fourier transform of a function is equivalent to representing it as the sum of sine functions. By eliminating undesirable high- and/or low-frequency components (i.e. , dropping some of the sine functions) and taking an inverse Fourier transform to get us back into the time domain, we can filter an image to remove noise and other artifacts. For example, the sharp spike in the figure above represents the period of the single sine function that closely models the input data. The rest is noise.
- *Image compression* – A smoothed, filtered image contains less information than the original, while retaining a similar appearance. By eliminating the coefficients of sine functions that contribute relatively little to the image, we can reduce the size of the image at little cost in image fidelity.
- *Convolution and deconvolution* – Fourier transforms can efficiently compute convolutions of two sequences. A *convolution* is the pairwise product of elements from two different sequences, such as in multiplying two n-variable

polynomials f and g or comparing two character strings. Implementing such products directly takes $O(n^2)$, while the fast Fourier transform led to a $O(n \lg n)$ algorithm.

Another example comes from image processing. Because a scanner measures the darkness of an image patch instead of a single point, the scanned input is always blurred. A reconstruction of the original signal can be obtained by deconvoluting the input signal with a Gaussian point-spread function.

- *Computing the correlation of functions* – The *correlation function* of two functions $f(t)$ and $g(t)$ is defined by

$$z(t) = \int_{-\infty}^{\infty} f(\tau)g(t+\tau)d\tau$$

and can be easily computed using Fourier transforms. When two functions are similar in shape but one is shifted relative to the other (such as $f(t) = \sin(t)$ and $g(t) = \cos(t)$), the value of $z(t_0)$ will be large at this shift offset t_0. As an application, suppose that we want to detect whether there are any funny periodicities in our random-number generator. We can generate a large series of random numbers, turn them into a time series (the ith number at time i), and take the Fourier transform of this series. Any funny spikes will correspond to potential periodicities.

The discrete Fourier transform takes as input n complex numbers h_k, $0 \leq k \leq n-1$, corresponding to equally spaced points in a time series, and outputs n complex numbers H_k, $0 \leq k \leq n-1$, each describing a sine function of given frequency. The discrete Fourier transform is defined by

$$H_m = \sum_{k=0}^{n-1} h_k e^{-2\pi ikm/n}$$

and the inverse Fourier transform is defined by

$$h_m = \frac{1}{n} \sum_{k=0}^{n-1} H_k e^{2\pi ikm/n}$$

which enables us move easily between h and H.

Since the output of the discrete Fourier transform consists of n numbers, each of which is computed using a formula on n numbers, they can be computed in $O(n^2)$ time. The fast Fourier transform (FFT) is an algorithm that computes the discrete Fourier transform in $O(n \log n)$. This is arguably the most important algorithm known, for it opened the door to modern signal processing. Several different algorithms call themselves FFTs, all of which are based on a divide-and-conquer approach. Essentially, the problem of computing the discrete Fourier transform on

n points is reduced to computing two transforms on $n/2$ points each, and is then applied recursively.

The FFT usually assumes that n is a power of two. If this is not the case, you are usually better off padding your data with zeros to create $n = 2^k$ elements rather than hunting for a more general code.

Many signal-processing systems have strong real-time constraints, so FFTs are often implemented in hardware, or at least in assembly language tuned to the particular machine. Be aware of this possibility if the codes prove too slow.

Implementations: FFTW is a C subroutine library for computing the discrete Fourier transform in one or more dimensions, with arbitrary input size, and supporting both real and complex data. It is the clear choice among freely available FFT codes. Extensive benchmarking proves it to be the "Fastest Fourier Transform in the West." Interfaces to Fortran and C++ are provided. FFTW received the 1999 J. H. Wilkinson Prize for Numerical Software. It is available at *http://www.fftw.org/*.

FFTPACK is a package of Fortran subprograms for the fast Fourier transform of periodic and other symmetric sequences, written by P. Swartzrauber. It includes complex, real, sine, cosine, and quarter-wave transforms. FFTPACK resides on Netlib (see Section 19.1.5 (page 659)) at *http://www.netlib.org/fftpack*. The GNU Scientific Library for C/C++ provides a reimplementation of FFTPACK. See *http://www.gnu.org/software/gsl/*.

Algorithm 545 [Fra79] of the *Collected Algorithms of the ACM* is a Fortran implementation of the fast Fourier transform optimizing virtual memory performance. See Section 19.1.5 (page 659) for further information.

Notes: Bracewell [Bra99] and Brigham [Bri88] are excellent introductions to Fourier transforms and the FFT. See also the exposition in [PFTV07]. Credit for inventing the fast Fourier transform is usually given to Cooley and Tukey [CT65], but see [Bri88] for a complete history.

A cache-oblivious algorithm for the fast Fourier transform is given in [FLPR99]. This paper first introduced the notion of cache-oblivious algorithms. The FFTW is based on this algorithm. See [FJ05] for more on the design of the FFTW.

An interesting divide-and-conquer algorithm for polynomial multiplication [KO63] does the job in $O(n^{1.59})$ time and is discussed in [AHU74, Man89]. An FFT-based algorithm that multiplies two n-bit numbers in $O(n \lg n \lg \lg n)$ time is due to Schönhage and Strassen [SS71] and is presented in [AHU74].

It is an open question of whether complex variables are really fundamental to fast algorithms for convolution. Fortunately, fast convolution can be used as a black box in most applications. Many variants of string matching are based on fast convolution [Ind98].

In recent years, wavelets have been proposed to replace Fourier transforms in filtering. See [Wal99] for an introduction to wavelets.

Related Problems: Data compression (see page 637), high-precision arithmetic (see page 423).

14

Combinatorial Problems

We now consider several algorithmic problems of a purely combinatorial nature. These include sorting and permutation generations, both of which were among the first non-numerical problems arising on electronic computers. Sorting can be viewed as identifying or imposing a total order on the keys, while searching and selection involve identifying specific keys based on their position in this total order.

The rest of this section deals with other combinatorial objects, such as permutations, partitions, subsets, calendars, and schedules. We are particularly interested in algorithms that *rank* and *unrank* combinatorial objects—i.e. , that map each distinct object to and from a unique integer. Rank and unrank operations make many other tasks simple, such as generating random objects (pick a random number and unrank) or listing all objects in order (iterate from 1 to n and unrank).

We conclude with the problem of generating graphs. Graph algorithms are more fully presented in subsequent sections of the catalog.

Books on general combinatorial algorithms, in this restricted sense, include:

- *Nijenhuis and Wilf* [NW78] – This book specializes in algorithms for constructing basic combinatorial objects such as permutations, subsets, and partitions. Such algorithms are often very short but hard to locate and usually are surprisingly subtle. Fortran programs for all of the algorithms are provided, as well as a discussion of the theory behind each of them. See Section 19.1.10 for details.

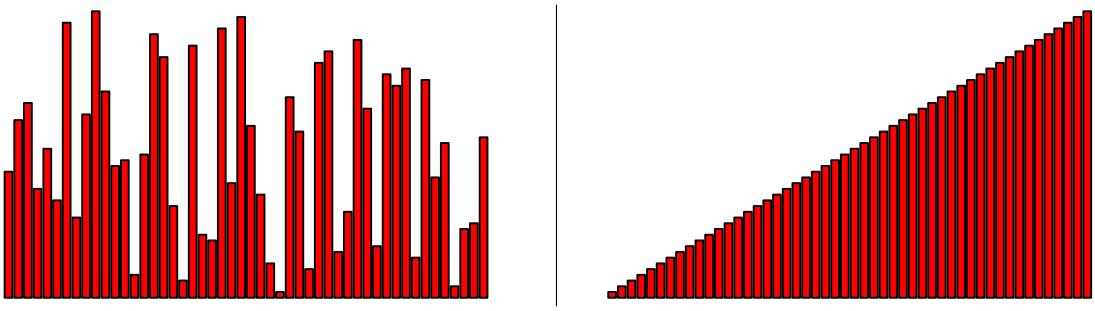
- *Kreher and Stinson* [KS99] – The most recent book on combinatorial generation algorithms, with additional particular focus on algebraic problems such as isomorphism and dealing with symmetry.

S.S. Skiena, *The Algorithm Design Manual*, 2nd ed., DOI: 10.1007/978-1-84800-070-4_14,

- *Knuth* [Knu97a, Knu98] – The standard reference on searching and sorting, with significant material on combinatorial objects such as permutations. New material on the generation of permutations [Knu05a], subsets and partitions [Knu05b], and trees [Knu06] has been released on "fasciles,"—short paperback chunks of what is slated to be the mythical Volume 4.

- *Stanton and White* [SW86a] – An undergraduate combinatorics text with algorithms for generating permutations, subsets, and set partitions. It contains relevant programs in Pascal.

- *Pemmaraju and Skiena* [PS03] – This description of *Combinatorica*, a library of over 400 Mathematica functions for generating combinatorial objects and graph theory (see Section 19.1.9), provides a distinctive view of how different algorithms can fit together. Its second author is considered qualified to write a manual on algorithm design.



INPUT OUTPUT

14.1 Sorting

Input description: A set of n items.

Problem description: Arrange the items in increasing (or decreasing) order.

Discussion: Sorting is the most fundamental algorithmic problem in computer science. Learning the different sorting algorithms is like learning scales for a musician. Sorting is the first step in solving a host of other algorithm problems, as shown in Section 4.2 (page 107). Indeed, *"when in doubt, sort"* is one of the first rules of algorithm design.

Sorting also illustrates all the standard paradigms of algorithm design. The result is that most programmers are familiar with many different sorting algorithms, which sows confusion as to which should be used for a given application. The following criteria can help you decide:

- *How many keys will you be sorting?* – For small amounts of data (say $n \leq 100$), it really doesn't matter much which of the quadratic-time algorithms you use. Insertion sort is faster, simpler, and less likely to be buggy than bubblesort. Shellsort is closely related to, but much faster than, insertion sort, but it involves looking up the right insert sequences in Knuth [Knu98].

  When you have more than 100 items to sort, it is important to use an $O(n \lg n)$-time algorithm like heapsort, quicksort, or mergesort. There are various partisans who favor one of these algorithms over the others, but since it can be hard to tell which is fastest, it usually doesn't matter.

  Once you get past (say) 5,000,000 items, it is important to start thinking about external-memory sorting algorithms that minimize disk access. Both types of algorithm are discussed below.

- *Will there be duplicate keys in the data?* – The sorted order is completely defined if all items have distinct keys. However, when two items share the same key, something else must determine which one comes first. In many applications it doesn't matter, so any sorting algorithm suffices. Ties are often broken by sorting on a secondary key, like the first name or initial when the family names collide.

  Occasionally, ties need to be broken by their initial position in the data set. Suppose the 5th and 27th items of the initial data set share the same key. This means the 5th item must appear before the 27th in the final order. A *stable* sorting algorithm preserves the original ordering in case of ties. Most of the quadratic-time sorting algorithms are stable, while many of the $O(n \lg n)$ algorithms are not. If it is important that your sort be stable, it is probably better to explicitly use the initial position as a secondary key in your comparison function rather than trust the stability of your implementation.

- *What do you know about your data?* – You can often exploit special knowledge about your data to get it sorted faster or more easily. Of course, general sorting is a fast $O(n \lg n)$ operation, so if the time spent sorting is really the bottleneck in your application, you are a fortunate person indeed.

  - *Has the data already been partially sorted?* If so, certain algorithms like insertion sort perform better than they otherwise would.
  - *Do you know the distribution of the keys?* If the keys are randomly or uniformly distributed, a *bucket* or *distribution sort* makes sense. Throw the keys into bins based on their first letter, and recur until each bin is small enough to sort by brute force. This is very efficient when the keys get evenly distributed into buckets. However, bucket sort would perform very badly sorting names on the membership roster of the "Smith Society."
  - *Are your keys very long or hard to compare?* If your keys are long text strings, it might pay to use a relatively short prefix (say ten characters) of each key for an initial sort, and then resolve ties using the full key. This is particularly important in external sorting (see below), since you don't want to waste your fast memory on the dead weight of irrelevant detail.

    Another idea might be to use radix sort. This always takes time linear in the number of characters in the file, instead of $O(n \lg n)$ times the cost of comparing two keys.
  - *Is the range of possible keys very small?* If you want to sort a subset of, say, $n/2$ distinct integers, each with a value from 1 to n, the fastest algorithm would be to initialize an n-element bit vector, turn on the bits corresponding to keys, then scan from left to right and report the positions with true bits.

- *Do I have to worry about disk accesses?* – In massive sorting problems, it may not be possible to keep all data in memory simultaneously. Such a problem is called *external sorting*, because one must use an external storage device. Traditionally, this meant tape drives, and Knuth [Knu98] describes a variety of intricate algorithms for efficiently merging data from different tapes. Today, it usually means virtual memory and swapping. Any sorting algorithm will run using virtual memory, but most will spend all their time swapping.

  The simplest approach to external sorting loads the data into a B-tree (see Section 12.1 (page 367)) and then does an in-order traversal of the tree to read the keys off in sorted order. Real high-performance sorting algorithms are based on multiway-mergesort. Files containing portions of the data are sorted into runs using a fast internal sort, and then files with these sorted runs are merged in stages using 2- or k-way merging. Complicated merging patterns and buffer management based on the properties of the external storage device can be used to optimize performance.

- *How much time do you have to write and debug your routine?* – If I had under an hour to deliver a working routine, I would probably just implement a simple selection sort. If I had an afternoon to build an efficient sort routine, I would probably use heapsort, for it delivers reliable performance without tuning.

The best general-purpose internal sorting algorithm is quicksort (see Section 4.2 (page 107)), although it requires tuning effort to achieve maximum performance. Indeed, you are much better off using a library function instead of coding it yourself. A poorly written quicksort will likely run more slowly than a poorly written heapsort.

If you are determined to implement your own quicksort, use the following heuristics, which make a big difference in practice:

- *Use randomization* – By randomly permuting (see Section 14.4 (page 448)) the keys before sorting, you can eliminate the potential embarrassment of quadratic-time behavior on nearly-sorted data.

- *Median of three* – For your pivot element, use the median of the first, last, and middle elements of the array to increase the likelihood of partitioning the array into roughly equal pieces. Some experiments suggest using a larger sample on big subarrays and a smaller sample on small ones.

- *Leave small subarrays for insertion sort* – Terminating the quicksort recursion and switching to insertion sort makes sense when the subarrays get small, say fewer than 20 elements. You should experiment to determine the best switchpoint for your implementation.

- *Do the smaller partition first* – Assuming that your compiler is smart enough to remove tail recursion, you can minimize run-time memory by processing

the smaller partition before the larger one. Since successive stored calls are at most half as large as the previous one, only $O(\lg n)$ stack space is needed.

Before you get started, see Bentley's article on building a faster quicksort [Ben92b].

Implementations: The best freely available sort program is presumably GNU sort, part of the GNU core utilities library. See *http://www.gnu.org/software/coreutils/*. Be aware that there are also commercial vendors of high-performance external sorting programs. These include Cosort (*www.cosort.com*), Syncsort (*www.syncsort.com*) and Ordinal Technology (*www.ordinal.com*).

Modern programming languages provide libraries offering complete and efficient container implementations. Thus, you should never need to implement your own sort routine. The C standard library contains `qsort`, a generic implementation of (presumably) quicksort. The C++ *Standard Template Library* (STL) provides both `sort` and `stable_sort` methods. It is available with documentation at *http://www.sgi.com/tech/stl/*. See Josuttis [Jos99], Meyers [Mey01] and Musser [MDS01] for more detailed guides to using STL and the C++ standard library.

Java Collections (JC), a small library of data structures, is included in the `java.util` package of Java standard edition (*http://java.sun.com/javase/*). In particular, `SortedMap` and `SortedSet` classes are provided.

For a C++ implementation of an cache-oblivious algorithm (*funnelsort*), check out *http://kristoffer.vinther.name/projects/funnelsort/*. Benchmarks attest to its excellent performance.

There are a variety of websites that provide applets/animations of all the basic sorting algorithms, including bubblesort, heapsort, mergesort, quicksort, radix sort, and shellsort. Many of these are quite interesting to watch. Indeed, sorting is the canonical problem for algorithm animation. Representative examples include Harrison (*http://www.cs.ubc.ca/spider/harrison/Java/sorting-demo.html*) and Bentley [Ben99] (*http://www.cs.bell-labs.com/cm/cs/pearls/sortanim.html*).

Notes: Knuth [Knu98] is the best book that has been written on sorting and indeed is the best book that will ever be written on sorting. It is now over thirty years old, but remains fascinating reading. One area that has developed since Knuth is sorting under presortedness measures, surveyed in [ECW92]. A noteworthy reference on sorting is [GBY91], which includes pointers to algorithms for partially sorted data and includes implementations in C and Pascal for all of the fundamental algorithms.

Expositions on the basic internal sorting algorithms appear in every algorithms text. Heapsort was first invented by Williams [Wil64]. Quicksort was invented by Hoare [Hoa62], with careful analysis and implementation by Sedgewick [Sed78]. Von Neumann is credited with having produced the first implementation of mergesort on the EDVAC in 1945. See Knuth for a full discussion of the history of sorting, dating back to the days of punch-card tabulating machines.

The primary competitive forum for high-performance sorting is an annual competition initiated by the late Jim Gray. See *http://research.microsoft.com/barc/SortBenchmark/* for current and previous results, which are are either inspiring or depressing depending

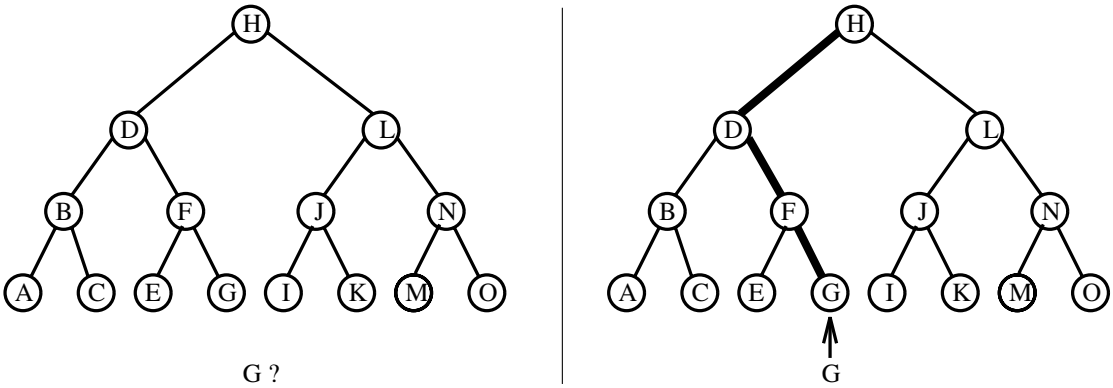
upon how you look at it. The magnitude of progress is inspiring (the million-record instances of the original benchmarks are now too small to bother with) but it is depressing (to me) the extent that systems/memory management issues thoroughly trump the combinatorial/algorithmic aspects of sorting.

Modern attempts to engineer high-performance sort programs include work on both cache-conscious [LL99] and cache-oblivious [BFV07] sorting.

Sorting has a well-known $\Omega(n \lg n)$ lower bound under the algebraic decision tree model [BO83]. Determining the exact number of comparisons required for sorting n elements, for small values of n, has generated considerable study. See [Aig88, Raw92] for expositions and Peczarski [Pec04, Pec07] for the latest results.

This lower-bound does not hold under different models of computation. Fredman and Willard [FW93] present an $O(n\sqrt{\lg n})$ algorithm for sorting under a model of computation that permits arithmetic operations on keys. See Andersson [And05] for a survey of algorithms for fast sorting on such nonstandard models of computation.

Related Problems: Dictionaries (see page 367), searching (see page 441), topological sorting (see page 481).



INPUT OUTPUT

14.2 Searching

Input description: A set of n keys S, and a query key q.

Problem description: Where is q in S?

Discussion: "Searching" is a word that means different things to different people. Searching for the global maximum or minimum of a function is the problem of *unconstrained optimization* and is discussed in Section 13.5 (page 407). Chess-playing programs select the best move to make via an exhaustive search of possible moves using a variation of backtracking (see Section 7.1 (page 231)).

Here we consider the task of searching for a key in a list, array, or tree. Dictionary data structures maintain efficient access to sets of keys under insertion and deletion and are discussed in Section 12.1 (page 367). Typical dictionaries include binary trees and hash tables.

We treat searching as a problem distinct from dictionaries because simpler and more efficient solutions emerge when our primary interest is static searching without insertion/deletion. These little data structures can yield large performance improvements when properly employed in an innermost loop. Also, ideas such as binary search and self-organization apply to other problems and well justify our attention.

Our two basic approaches are sequential search and binary search. Both are simple, yet have interesting and subtle variations. In *sequential search*, we start from the front of our list/array of keys and compare each successive item against the key until we find a match or reach the end. In *binary search*, we start with a sorted array of keys. To search for key q, we compare q to the middle key $S_{n/2}$. If q is before $S_{n/2}$, it must reside in the top half of our set; if not, it must reside in the

bottom half of our set. By repeating this process on the correct half, we find the key using $\lceil \lg n \rceil$ comparisons. This is a big win over the $n/2$ comparisons we expect with sequential search. See Section 4.9 (page 132) for more on binary search.

A sequential search is the simplest algorithm, and likely to be fastest on up to about 20 elements. Beyond (say) 100 elements, binary search will clearly be more efficient than sequential search, easily justifying the cost of sorting if there will be multiple queries. Other issues come into play, however, in identifying the proper variant of the algorithm:

- *How much time can you spend programming?* – A binary search is a notoriously tricky algorithm to program correctly. It took seventeen years after its invention until the first *correct* version of a binary search was published! Don't be afraid to start from one of the implementations described below. Test it completely by writing a driver that searches for every key in the set S as well as between the keys.

- *Are certain items accessed more often than other ones?* – Certain English words (such as "the") are much more likely to occur than others (such as "defenestrate"). We can reduce the number of comparisons in a sequential search by putting the most popular words on the top of the list and the least popular ones at the bottom. Nonuniform access is usually the rule, not the exception. Many real-world distributions are governed by *power laws*. A classic example is word use in English, which is fairly accurately modeled by *Zipf's law*. Under *Zipf's law*, the ith most frequently accessed key is selected with probability $(i-1)/i$ times the probability of the $(i-1)$st most popular key, for all $1 \leq i \leq n$.

  Knowledge of access frequencies is easy to exploit with sequential search. But the issue is more complicated with binary trees. We want popular keys close to the root (so we hit them quickly) but not at the expense of losing balance and degenerating into sequential search. The answer is to employ a dynamic programming algorithm to find the *optimal binary search tree*. The key observation is that each possible root node i partitions the space of keys into those to the left of i and those to the right; each of which should be represented by an optimal binary search tree on a smaller subrange of keys. The root of the optimal tree is selected to minimize the expected search costs of the resulting partition.

- *Might access frequencies change over time?* – Preordering a list or tree to exploit a skewed access pattern requires knowing the access pattern in advance. For many applications, it can be difficult to obtain such information. Better are *self-organizing lists*, where the order of the keys changes in response to the queries. The best self-organizing scheme is move-to-front; that is, we move the most recently searched-for key from its current position to the front of the list. Popular keys keep getting boosted to the front, while unsearched-for

keys drift towards the back of the list. There is no need to keep track of the frequency of access; just move the keys on demand. Self-organizing lists also exploit *locality of reference*, since accesses to a given key are likely to occur in clusters. A hot key will be maintained near the top of the list during a cluster of accesses, even if other keys have proven more popular in the past.

Self-organization can extend the useful size range of sequential search. However, you should switch to binary search beyond 100 elements. But consider using *splay trees*. which are self-organizing binary search trees that rotate each searched-for node to the root. They offer excellent amortized performance guarantees.

- *Is the key close by?* – Suppose we know that the target key is to the right of position p, and we think it is nearby. A sequential search is fast if we are correct, but we will be punished severely when we guess wrong. A better idea is to test repeatedly at larger intervals ($p+1$, $p+2$, $p+4$, $p+8$, $p+16$, ...) to the right until we find a key to the right of our target. Now we have a window containing the target and we can proceed with binary search.

  Such a *one-sided binary search* finds the target at position $p+l$ using at most $2\lceil \lg l \rceil$ comparisons, so it is faster than binary search when $l << n$, yet it can never be much worse. One-sided binary search is particularly useful in unbounded search problems, such as in numerical root finding.

- *Is my data structure sitting on external memory?* – Once the number of keys grows *too* large, a binary search loses its status as the best search technique. A binary search jumps wildly around the set of keys looking for midpoints to compare, and so each comparison requires reading a new page in from external memory. Much better are data structures such as B-trees (see Section 12.1 (page 367)) or Emde Boas trees (see notes below), which cluster the keys into pages to minimize the number of disk accesses per search.

- *Can I guess where the key should be?* – In *interpolation search*, we exploit our understanding of the distribution of keys to guess where to look next. An interpolation search is probably a more accurate description of how we use a telephone book than binary search. Suppose we are searching for *Washington, George* in a sorted telephone book. We would be safe making our first comparison three-fourths of the way down the list, essentially doing two comparisons for the price of one.

  Although an interpolation search is an appealing idea, we caution against it for three reasons: First, you have to work very hard to optimize your search algorithm before you can hope for a speedup over binary search. Second, even if you do beat a binary search, it is unlikely to be by enough to have justified the exercise. Finally, your program will be much less robust and efficient when the distribution changes, such as when your application gets ported to work on French words instead of English.

Implementations: The basic sequential and binary search algorithms are simple enough that you may consider implementing them yourself. That said, the C standard library contains `bsearch`, a generic implementation of (presumably) a binary search. The C++ *Standard Template Library* (STL) provides `find` (sequential search) and `binary_search` iterators. *Java Collections* (JC), provides `binarySearch` in the `java.util` package of Java standard edition (*http://java.sun.com/javase/*).

Many data structure textbooks provide extensive and illustrative implementations. Sedgewick (*http://www.cs.princeton.edu/∼rs/*) [Sed98] and Weiss (*http://www.cs.fiu.edu/∼weiss/*) [Wei06] provide implementation of splay trees and other search structures in both C++ and Java.

Notes: *The Handbook of Data Structures and Applications* [MS05] provides up-to-date surveys on all aspects of dictionary data structures. Other surveys include Mehlhorn and Tsakalidis [MT90b] and Gonnet and Baeza-Yates [GBY91]. Knuth [Knu97a] provides a detailed analysis and exposition on all fundamental search algorithms and dictionary data structures, but omits such modern data structures as red-black and splay trees.

The next position probed in linear interpolation search on an array of sorted numbers is given by

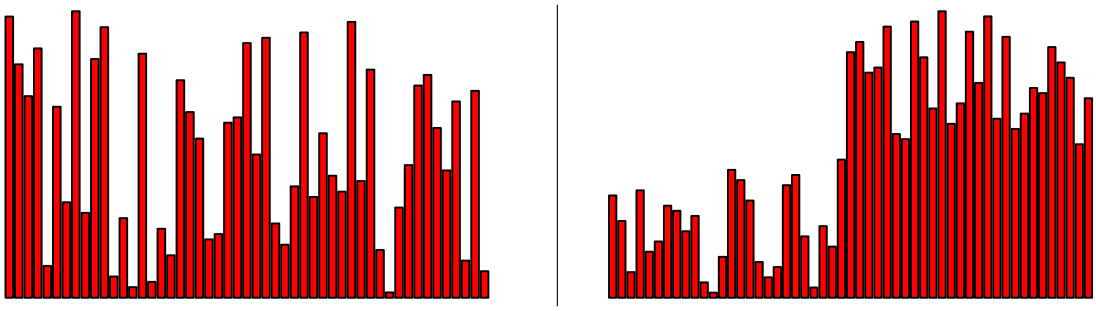
$$next = (low - 1) + \lceil \frac{q - S[low - 1]}{S[high + 1] - S[low - 1]} \times (high - low + 1) \rceil$$

where q is the query numerical key and S the sorted numerical array. If the keys are drawn independently from a uniform distribution, the expected search time is $O(\lg \lg n)$ [DJP04, PIA78].

Nonuniform access patterns can be exploited in binary search trees by structuring them so that popular keys are located near the root, thus minimizing search time. Dynamic programming can be used to construct such optimal search trees in $O(n \lg n)$ time [Knu98]. Stout and Warren [SW86b] provide a slick algorithm to efficiently transform a binary tree to a minimum height (optimally balanced) tree using rotations.

The Van Emde Boas layout of a binary tree (or sorted array) offers better external memory performance than conventional binary search, at a cost of greater implementation complexity. See the survey of Arge, et al. [ABF05] for more on this and other cache-oblivious data structures.

Related Problems: Dictionaries (see page 367), sorting (see page 436).



INPUT OUTPUT

14.3 Median and Selection

Input description: A set of n numbers or keys, and an integer k.

Problem description: Find the key smaller than exactly k of the n keys.

Discussion: Median finding is an essential problem in statistics, where it provides a more robust notion of average than the *mean*. The mean wealth of people who have published research papers on sorting is significantly affected by the presence of one William Gates [GP79], although his effect on the *median* wealth is merely to cancel out one starving graduate student.

Median finding is a special case of the more general *selection* problem, which asks for the kth element in sorted order. Selection arises in several applications:

- *Filtering outlying elements* – In dealing with noisy data, it is usually a good idea to throw out (say) the 10% largest and smallest values. Selection can be used to identify the items defining the 10^{th} and 90^{th} percentiles, and the outliers then filtered out by comparing each item to the two selected bounds.
- *Identifying the most promising candidates* – In a computer chess program, we might quickly evaluate all possible next moves, and then decide to study the top 25% more carefully. Selection followed by filtering is the way to go.
- *Deciles and related divisions* – A useful way to present income distribution in a population is to chart the salary of the people ranked at regular intervals, say exactly at the 10th percentile, 20th percentile, etc. Computing these values is simply selection on the appropriate position ranks.
- *Order statistics* – Particularly interesting special cases of selection include finding the smallest element ($k = 1$), the largest element ($k = n$), and the median element ($k = n/2$).

The mean of n numbers can be computed in linear time by summing the elements and dividing by n. However, finding the median is a more difficult problem. Algorithms that compute the median can readily be generalized to arbitrary selection. Issues in median finding and selection include:

- *How fast does it have to be?* – The most elementary median-finding algorithm sorts the items in $O(n \lg n)$ time and then returns the item occupying the $(n/2)$nd position. The good thing is that this gives much more information than just the median, enabling you to select the kth element (for any $1 \leq k \leq n$) in constant time after the sort. However, there are faster algorithms if all you want is the median.

  In particular, there is an $O(n)$ *expected*-time algorithm based on quicksort. Select a random element in the data set as a pivot, and use it to partition the data into sets of elements less than and greater than the pivot. From the sizes of these sets, we know the position of the pivot in the total order, and hence whether the median lies to the left or right of this point. Now we recur on the appropriate subset until it converges on the median. This takes (on average) $O(\lg n)$ iterations, with the cost of each iteration being roughly half that of the previous one. This defines a geometric series that converges to a linear-time algorithm, although if you are very unlucky it takes the same time as quicksort, $O(n^2)$.

  More complicated algorithms can find the median in worst-case linear time. However, the expected-time algorithm will likely win in practice. Just make sure to select random pivots to avoid the worst case.

- *What if you only get to see each element once?* – Selection and median finding become expensive on large datasets because they typically require several passes through external memory. In data-streaming applications, the volume of data is often too large to store, making repeated consideration (and thus exact median finding) impossible. Much better is computing a small summary of the data for future analysis, say approximate deciles or frequency moments (where the kth moment of stream x is defined as $F_k = \sum_i x_i^k$).

  One solution to such a problem is random sampling. Flip a coin for each value to decide whether to store it, with the probability of heads set low enough that you won't overflow your buffer. Likely the median of your samples will be close to that of the underlying data set. Alternately, you can devote some fraction of memory to retaining (say) decile values of large blocks, and then combine these decile distributions to yield more refined decile bounds.

- *How fast can you find the mode?* – Beyond mean and median lies a third notion of average. The *mode* is defined to be the element that occurs the greatest number of times in the data set. The best way to compute the mode sorts the set in $O(n \lg n)$ time, which places all identical elements next to each other. By doing a linear sweep from left to right on this sorted set, we can

count the length of the longest run of identical elements and hence compute the mode in a total of $O(n \lg n)$ time.

In fact, there is no faster worst-case algorithm possible to compute the mode, since the problem of testing whether there exist two identical elements in a set (called element uniqueness) can be shown to have an $\Omega(n \log n)$ lower bound. Element uniqueness is equivalent to asking whether the mode occurs more than once. Possibilities exist, at least theoretically, for improvements when the mode is large by using fast median computations.

Implementations: The C++ *Standard Template Library* (STL) provides a general selection method (`nth_element`) implemented using the linear expected-time algorithm. It is available with documentation at *http://www.sgi.com/tech/stl/*. See Josuttis [Jos99], Meyers [Mey01] and Musser [MDS01] for more detailed guides to using STL and the C++ standard library.

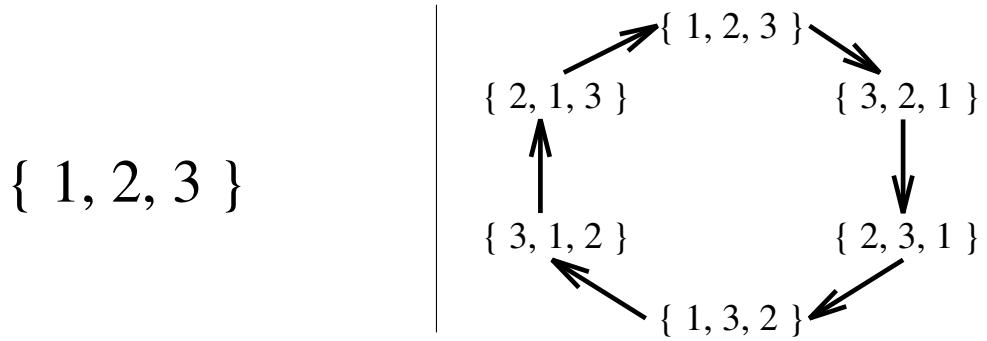
Notes: The linear expected-time algorithm for median and selection is due to Hoare [Hoa61]. Floyd and Rivest [FR75] provide an algorithm that uses fewer comparisons on average. Good expositions on linear-time selection include [AHU74, BvG99, CLRS01, Raw92], with [Raw92] being particularly enlightening.

Streaming algorithms have extensive applications to large data sets, and are well surveyed by Muthukrishnan [Mut05].

A sport of considerable theoretical interest is determining *exactly* how many comparisons are sufficient to find the median of n items. The linear-time algorithm of Blum et al. [BFP+72] proves that $c \cdot n$ suffice, but we want to know what c is. Dor and Zwick [DZ99] proved that $2.95n$ comparisons suffice to find the median. These algorithms attempt to minimize the number of element comparisons but not the total number of operations, and hence do not lead to faster algorithms in practice. They also hold the current best lower bower bound of $(2 + \epsilon)$ comparisons for median finding [DZ01].

Tight combinatorial bounds for selection problems are presented in [Aig88]. An optimal algorithm for computing the mode is given by [DM80].

Related Problems: Priority queues (see page 373), sorting (see page 436).



INPUT OUTPUT

14.4 Generating Permutations

Input description: An integer n.

Problem description: Generate (1) all, or (2) a random, or (3) the next permutation of length n.

Discussion: A permutation describes an arrangement or ordering of items. Many algorithmic problems in this catalog seek the best way to order a set of objects, including *traveling salesman* (the least-cost order to visit n cities), *bandwidth* (order the vertices of a graph on a line so as to minimize the length of the longest edge), and *graph isomorphism* (order the vertices of one graph so that it is identical to another). Any algorithm for solving such problems exactly must construct a series of permutations along the way.

There are $n!$ permutations of n items. This grows so quickly that you can't really expect to generate all permutations for $n > 12$, since $12! = 479{,}001{,}600$. Numbers like these should cool the ardor of anyone interested in exhaustive search and help explain the importance of generating random permutations.

Fundamental to any permutation-generation algorithm is a notion of order, the sequence in which the permutations are constructed from first to last. The most natural generation order is *lexicographic*, the sequence they would appear if they were sorted numerically. Lexicographic order for $n = 3$ is $\{1, 2, 3\}$, $\{1, 3, 2\}$, $\{2, 1, 3\}$, $\{2, 3, 1\}$, $\{3, 1, 2\}$, and finally $\{3, 2, 1\}$. Although lexicographic order is aesthetically pleasing, there is often no particular advantage to using it. For example, if you are searching through a collection of files, it does not matter whether the filenames are encountered in sorted order, so long as you eventually search through all of them. Indeed, nonlexicographic orders lead to faster and simpler permutation generation algorithms.

There are two different paradigms for constructing permutations: ranking/unranking and incremental change methods. The latter are more efficient, but ranking and unranking can be applied to solve a much wider class of problems. The key is to define the functions *rank* and *unrank* on all permutations p and integers n, m, where $|p| = n$ and $0 \leq m \leq n!$.

- *Rank(p)* – What is the position of p in the given generation order? A typical ranking function is recursive, such as basis case $Rank(\{1\}) = 0$ with

  $$Rank(p) = (p_1 - 1) \cdot (|p| - 1)! + Rank(p_2, \ldots, p_{|p|})$$

  Getting this right means relabeling the elements of the smaller permutation to reflect the deleted first element. Thus

  $$Rank(\{2, 1, 3\}) = 1 \cdot 2! + Rank(\{1, 2\}) = 2 + 0 \cdot 1! + Rank(\{1\}) = 2$$

- *Unrank(m,n)* – Which permutation is in position m of the $n!$ permutations of n items? A typical unranking function finds the number of times $(n-1)!$ goes into m and proceeds recursively. $Unrank(2, 3)$ tells us that the first element of the permutation must be '2', since $(2-1) \cdot (3-1)! \leq 2$ but $(3-1) \cdot (3-1)! > 2$. Deleting $(2-1) \cdot (3-1)!$ from m leaves the smaller problem $Unrank(0, 2)$. The ranking of 0 corresponds to the total order. The total order on the two remaining elements (since 2 has been used) is $\{1, 3\}$, so $Unrank(2, 3) = \{2, 1, 3\}$.

What the actual rank and unrank functions are does not matter as much as the fact that they must be inverses of each other. In other words, $p = Unrank(Rank(p), n)$ for all permutations p. Once you define ranking and unranking functions for permutations, you can solve a host of related problems:

- *Sequencing permutations* – To determine the *next* permutation that occurs in order after p, we can $Rank(p)$, add 1, and then $Unrank(p)$. Similarly, the permutation right before p in order is $Unrank(Rank(p) - 1, |p|)$. Counting through the integers from 0 to $n! - 1$ and unranking them is equivalent to generating all permutations.

- *Generating random permutations* – Selecting a random integer from 0 to $n!-1$ and then unranking it yields a truly random permutation.

- *Keep track of a set of permutations* – Suppose we want to construct random permutations and act only when we encounter one we have not seen before. We can set up a bit vector (see Section 12.5 (page 385)) with $n!$ bits, and set bit i to 1 if permutation *Unrank(i,n)* has been seen. A similar technique was employed with k-subsets in the Lotto application of Section 1.6 (page 23).

This rank/unrank method is best suited for small values of n, since $n!$ quickly exceeds the capacity of machine integers unless arbitrary-precision arithmetic is available (see Section 13.9 (page 423)). The incremental change methods work by defining the *next* and *previous* operations to transform one permutation into another, typically by swapping two elements. The tricky part is to schedule the swaps so that permutations do not repeat until all of them have been generated. The output picture above gives an ordering of the six permutations of $\{1, 2, 3\}$ using a single swap between successive permutations.

Incremental change algorithms for sequencing permutations are tricky, but they are so concise that they can be expressed in a dozen-line program. See the implementation section for pointers to code. Because the incremental change is only a single swap, these algorithms can be extremely fast—on average, constant time—which is independent of the size of the permutation! The secret is to represent the permutation using an n-element array to facilitate the swap. In certain applications, only the change between permutations is important. For example, in a brute-force program to search for the optimal TSP tour, the cost of the tour associated with the new permutation will be that of the previous permutation, with the addition and deletion of four edges.

Throughout this discussion, we have assumed that the items we are permuting are all distinguishable. However, if there are duplicates (meaning our set is a *multi-set*), you can save considerable time and effort by avoiding identical permutations. For example, there are only ten distinct permutations of $\{1, 1, 2, 2, 2\}$, instead of 120. To avoid duplicates, use backtracking and generate the permutations in lexicographic order.

Generating random permutations is an important little problem that people often stumble across, and often botch up. The right way is to use the following two-line, linear-time algorithm. We assume that *Random[i,n]* generates a random integer between i and n, inclusive.

for $i = 1$ to n do $a[i] = i$;
for $i = 1$ to $n - 1$ do $swap[a[i], a[Random[i, n]]$;

That this algorithm generates all permutations uniformly at random is not obvious. If you think so, convincingly explain why the following algorithm *does not* generate permutations uniformly:

for $i = 1$ to n do $a[i] = i$;
for $i = 1$ to $n - 1$ do $swap[a[i], a[Random[1, n]]$;

Such subtleties demonstrate why you must be very careful with random generation algorithms. Indeed, we recommend that you try some reasonably extensive experiments with *any* random generator before really believing it. For example, generate 10,000 random permutations of length 4 and see whether all 24 distinct permutations occur approximately the same number of times. If you know how to measure statistical significance, you are in even better shape.

Implementations: The C++ *Standard Template Library* (STL) provides two functions (`next_permutation` and `prev_permutation`) for sequencing permutations in lexicographic order. Kreher and Stinson [KS99] provide implementations of minimum change and lexicographic permutation generation in C at *http://www.math.mtu.edu/~kreher/cages/Src.html*.

The *Combinatorial Object Server* (*http://theory.cs.uvic.ca/*) developed by Frank Ruskey of the University of Victoria is a unique resource for generating permutations, subsets, partitions, graphs, and other objects. An interactive interface enables you to specify which objects you would like returned to you. Implementations in C, Pascal, and Java are available for certain types of objects.

C++ routines for generating an astonishing variety of combinatorial objects, including permutations and cyclic permutations, are available at *http://www.jjj.de/fxt/*.

Nijenhuis and Wilf [NW78] is a venerable but still excellent source on generating combinatorial objects. They provide efficient Fortran implementations of algorithms to construct random permutations and to sequence permutations in minimum-change order. Also included are routines to extract the cycle structure of a permutation. See Section 19.1.10 (page 661) for details.

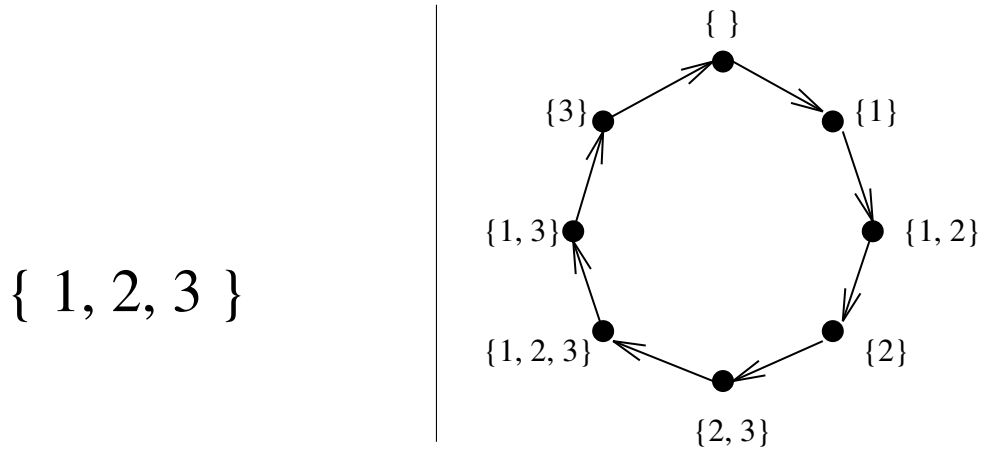
Combinatorica [PS03] provides Mathematica implementations of algorithms that construct random permutations and sequence permutations in minimum change and lexicographic orders. It also provides a backtracking routine to construct all distinct permutations of a multiset, and it supports various permutation group operations. See Section 19.1.9 (page 661).

Notes: The best recent reference on permutation generation is Knuth [Knu05a]. Sedgewick's excellent survey on the topic is older [Sed77], but this is not a fast moving area. Good expositions include [KS99, NW78, Rus03].

Fast permutation generation methods make only a single swap between successive permutations. The Johnson-Trotter algorithm [Joh63, Tro62] satisfies an even stronger condition, namely that the two elements being swapped are always adjacent. Simple linear-time ranking and unranking functions for permutations are given by Myrvold and Ruskey [MR01].

In the days before ready access to computers, books with tables of random permutations [MO63] were used instead of algorithms. The swap-based random permutation algorithm presented above was first described in [MO63].

Related Problems: Random-number generation (see page 415), generating subsets (see page 452), generating partitions (see page 456).



INPUT OUTPUT

14.5 Generating Subsets

Input description: An integer n.

Problem description: Generate (1) all, or (2) a random, or (3) the next subset of the integers $\{1, \ldots, n\}$.

Discussion: A subset describes a selection of objects, where the order among them does not matter. Many important algorithmic problems seek the best subset of a group of things: *vertex cover* seeks the smallest subset of vertices to touch each edge in a graph; *knapsack* seeks the most profitable subset of items of bounded total size; while *set packing* seeks the smallest subset of subsets that together cover each item exactly once.

There are 2^n distinct subsets of an n-element set, including the empty set as well as the set itself. This grows exponentially, but at a considerably slower rate than the $n!$ permutations of n items. Indeed, since $2^{20} = 1{,}048{,}576$, a brute-force search through all subsets of 20 elements is easily manageable. Since $2^{30} = 1{,}073{,}741{,}824$, you will certainly hit limits for slightly larger values of n.

By definition, the relative order among the elements does not distinguish different subsets. Thus, $\{1, 2, 5\}$ is the same as $\{2, 1, 5\}$. However, it is a very good idea to maintain your subsets in a sorted or *canonical* order to speed up such operations as testing whether or not two subsets are identical.

As with permutations (see Section 14.4 (page 448)), the key to subset generation problems is establishing a numerical sequence among all 2^n subsets. There are three primary alternatives:

- *Lexicographic order* – Lexicographic order means sorted order, and is often the most natural way to generate combinatorial objects. The eight subsets of $\{1,2,3\}$ in lexicographic order are $\{\}$, $\{1\}$, $\{1,2\}$, $\{1,2,3\}$, $\{1,3\}$, $\{2\}$, $\{2,3\}$, and $\{3\}$. But it is surprisingly difficult to generate subsets in lexicographic order. Unless you have a compelling reason to do so, don't bother.

- *Gray Code* – A particularly interesting and useful subset sequence is the minimum change order, wherein adjacent subsets differ by the insertion or deletion of exactly one element. Such an ordering, called a *Gray code*, appears in the output picture above.

  Generating subsets in Gray code order can be very fast, because there is a nice recursive construction. Construct a Gray code of $n-1$ elements G_{n-1}. Reverse a second copy of G_{n-1} and add n to each subset in this copy. Then concatenate them together to create G_n. Study the output example for clarification.

  Further, since only one element changes between subsets, exhaustive search algorithms built on Gray codes can be quite efficient. A set cover program would only have to update the change in coverage by the addition or deletion of one subset. See the implementation section below for Gray code subset-generation programs.

- *Binary counting* – The simplest approach to subset-generation problems is based on the observation that any subset S' is defined by the items of that S are in S'. We can represent S' by a binary string of n bits, where bit i is 1 iff the ith element of S is in S'. This defines a bijection between the 2^n binary strings of length n, and the 2^n subsets of n items. For $n = 3$, binary counting generates subsets in the following order: {}, {3}, {2}, {2,3}, {1}, {1,3}, {1,2}, {1,2,3}.

  This binary representation is the key to solving all subset generation problems. To generate all subsets in order, simply count from 0 to $2^n - 1$. For each integer, successively mask off each of the bits and compose a subset of exactly the items corresponding to 1 bits. To generate the *next* or *previous* subset, increment or decrement the integer by one. *Unranking* a subset is exactly the masking procedure, while *ranking* constructs a binary number with 1's corresponding to items in S and then converts this binary number to an integer.

  To generate a random subset, you might generate a random integer from 0 to $2^n - 1$ and unrank, although how your random number generator rounds things off might mean that certain subsets can never occur. Much better is to flip a coin n times, with the ith flip deciding whether to include element i in the subset. A coin flip can be robustly simulated by generating a random real or large integer and testing whether it is bigger or smaller than half its range. A Boolean array of n items can thus be used to represent subsets as a sort of premasked integer.

Generation problems for two closely related problems arise often in practice:

- *K-subsets* – Instead of constructing all subsets, we may only be interested in the subsets containing exactly k elements. There are $\binom{n}{k}$ such subsets, which is substantially less than 2^n, particularly for small values of k.

  The best way to construct all k-subsets is in lexicographic order. The ranking function is based on the observation that there are $\binom{n-f}{k-1}$ k-subsets whose smallest element is f. Using this, it is possible to determine the smallest element in the mth k-subset of n items. We then proceed recursively for subsequent elements of the subset. See the implementations below for details.

- *Strings* – Generating all subsets is equivalent to generating all 2^n strings of true and false. The same basic techniques apply to generate all or random strings on alphabets of size α, except there will be α^n strings in total.

Implementations: Kreher and Stinson [KS99] provide generators for both subsets and k-subsets, including lexicographic and Gray code orders. These implementations in C are available at *http://www.math.mtu.edu/~kreher/cages/Src.html*.

The *Combinatorial Object Server* (*http://theory.cs.uvic.ca/*), developed by Frank Ruskey of the University of Victoria, is a unique resource for generating permutations, subsets, partitions, graphs, and other objects. An interactive interface enables you to specify which objects you would like returned to you. Implementations in C, Pascal, and Java are available for certain types of objects.

C++ routines for generating an astonishing variety of combinatorial objects, including subsets and k-subsets (combinations), are available in the combinatorics package at *http://www.jjj.de/fxt/*.

Nijenhuis and Wilf [NW78] is a venerable but still excellent source on generating combinatorial objects. They provide efficient Fortran implementations of algorithms to construct random subsets and to sequence subsets in Gray code and lexicographic order. They also provide routines to construct random k-subsets and sequence them in lexicographic order. See Section 19.1.10 (page 661) for details on ftp-ing these programs. Algorithm 515 [BL77] of the *Collected Algorithms of the ACM* is another Fortran implementation of lexicographic k-subsets, available from Netlib (see Section 19.1.5 (page 659)).

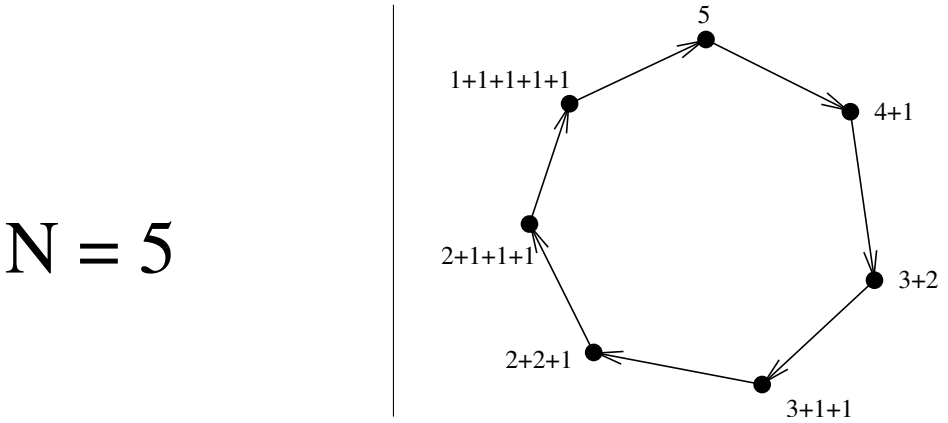
Combinatorica [PS03] provides Mathematica implementations of algorithms to construct random subsets and sequence subsets in Gray code, binary, and lexicographic order. They also provide routines to construct random k-subsets and strings, and sequence them lexicographically. See Section 19.1.9 (page 661) for further information on Combinatorica.

Notes: The best reference on subset generation is Knuth [Knu05b]. Good expositions include [KS99, NW78, Rus03]. Wilf [Wil89] provides an update of [NW78], including a thorough discussion of modern Gray code generation problems.

Gray codes were first developed [Gra53] to transmit digital information in a robust manner over an analog channel. By assigning the code words in Gray code order, the ith word differs only slightly from the $(i+1)$st, so minor fluctuations in analog signal strength corrupts only a few bits. Gray codes have a particularly nice correspondence to Hamiltonian cycles on the hypercube. Savage [Sav97] gives an excellent survey of Gray codes (minimum change orderings) for a large class of combinatorial objects, including subsets.

The popular puzzle *Spinout*, manufactured by ThinkFun (formerly Binary Arts Corporation), is solved using ideas from Gray codes.

Related Problems: Generating permutations (see page 448), generating partitions (see page 456).



INPUT OUTPUT

14.6 Generating Partitions

Input description: An integer n.

Problem description: Generate (1) all, or (2) a random, or (3) the next integer or set partitions of length n.

Discussion: There are two different types of combinatorial objects denoted by the word "partition," namely integer partitions and set partitions. They are quite different beasts, but it is a good idea to make both a part of your vocabulary:

- *Integer partitions* are multisets of nonzero integers that add up exactly to n. For example, the seven distinct integer partitions of 5 are {5}, {4,1}, {3,2}, {3,1,1}, {2,2,1}, {2,1,1,1}, and {1,1,1,1,1}. An interesting application I encountered that required generating integer partitions was in a simulation of nuclear fission. When an atom is smashed, the nucleus of protons and neutrons is broken into a set of smaller clusters. The sum of the particles in the set of clusters must equal the original size of the nucleus. As such, the integer partitions of this original size represent all the possible ways to smash an atom.

- *Set partitions* divide the elements $1, \ldots, n$ into nonempty subsets. There are 15 distinct set partitions of $n = 4$: {1234}, {123,4}, {124,3}, {12,34}, {12,3,4}, {134,2}, {13,24}, {13,2,4}, {14,23}, {1,234}, {1,23,4}, {14,2,3}, {1,24,3}, {1,2,34}, and {1,2,3,4}. Several algorithm problems return set partitions as results, including *vertex/edge coloring* and *connected components*.

Although the number of integer partitions grows exponentially with n, they do so at a refreshingly slow rate. There are only 627 partitions of $n = 20$. It is even possible to enumerate all integer partitions of $n = 100$, since there there are only 190,569,292 of them.

The easiest way to generate integer partitions is to construct them in lexicographically decreasing order. The first partition is $\{n\}$ itself. The general rule is to subtract 1 from the smallest part that is > 1 and then collect all the 1's so as to match the new smallest part > 1. For example, the partition following $\{4, 3, 3, 3, 1, 1, 1, 1\}$ is $\{4, 3, 3, 2, 2, 2, 1\}$, since the five 1's left after $3 - 1 = 2$ becomes the smallest part are best packaged as 2,2,1. When the partition is all 1's, we have completed one pass through all the partitions.

This algorithm is sufficiently intricate to program that you should consider using one of the implementations below. In either case, test it to make sure that you get exactly 627 distinct partitions for $n = 20$.

Generating integer partitions uniformly at random is a trickier business than generating random permutations or subsets. This is because selecting the first (i.e. largest) element of the partition has a dramatic effect on the number of possible partitions that can be generated. Observe that no matter how large n is, there is only one partition of n whose largest part is 1. The number of partitions of n with largest part at most k is given by the recurrence

$$P_{n,k} = P_{n-k,k} + P_{n,k-1}$$

with the two boundary conditions $P_{x-y,x} = P_{x-y,x-y}$ and $P_{n,1} = 1$. This function can be used to select the largest part of your random partition with the correct probabilities and, by proceeding recursively, to eventually construct the entire random partition. Implementations are cited below.

Random partitions tend to have large numbers of fairly small parts, best visualized by a Ferrers diagram as in Figure 14.1. Each row of the diagram corresponds to one part of the partition, with the size of each part represented by that many dots.

Set partitions can be generated using techniques akin to integer partitions. Each set partition is encoded as a *restricted growth function*, $a_1, \ldots, a_n$, where $a_1 = 0$ and $a_i \leq 1 + \max(a_1, \ldots, a_i)$, for $i = 2, \ldots, n$. Each distinct digit identifies a subset, or *block*, of the partition, while the growth condition ensures that the blocks are sorted into a canonical order based on the smallest element in each block. For example, the restricted growth function $0, 1, 1, 2, 0, 3, 1$ defines the set partition $\{\{1, 5\}, \{2, 3, 7\}, \{4\}, \{6\}\}$.

Since there is a one-to-one equivalence between set partitions and restricted growth functions, we can use lexicographic order on the restricted growth functions to order the set partitions. Indeed, the fifteen partitions of $\{1, 2, 3, 4\}$ listed above are sequenced according to the lexicographic order of their restricted growth function (check it out).

We can use a similar counting strategy to generate random set partitions as we did with integer partitions. The Stirling numbers of the second kind $\{{n \atop k}\}$ count the

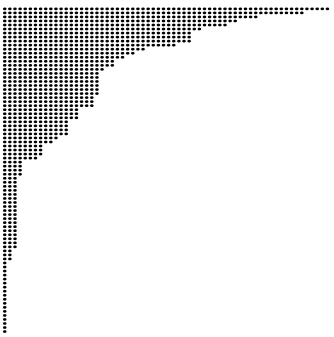


Figure 14.1: The Ferrers diagram of a random partition of $n = 1000$

number of partitions of $\{1, \ldots, n\}$ with exactly k blocks. They are computed using the recurrence

$$\left\{ {n \atop k} \right\} = \left\{ {n-1 \atop k-1} \right\} + k \left\{ {n-1 \atop k} \right\}$$

with the boundary conditions $\left\{ {n \atop n} \right\} = \left\{ {n \atop 1} \right\} = 1$. The reader is referred to the Notes and Implementations section for more details.

Implementations: Kreher and Stinson [KS99] generate both integer and set partitions in lexicographic order, including ranking/unranking functions. These implementations in C are available at *http://www.math.mtu.edu/~kreher/cages/Src.html*.

The *Combinatorial Object Server* (*http://theory.cs.uvic.ca/*) developed by Frank Ruskey of the University of Victoria is a unique resource for generating permutations, subsets, partitions, graphs, and other objects. An interactive interface enables you to specify which objects you would like returned. Implementations in C, Pascal, and Java are available for certain types of objects.

C++ routines for generating an astonishing variety of combinatorial objects, including integer partitions and compositions, are available in the combinatorics package at *http://www.jjj.de/fxt/*.

Nijenhuis and Wilf [NW78] is a venerable but still excellent source on generating combinatorial objects. They provide efficient Fortran implementations of algorithms to construct random and sequential integer partitions, set partitions, compositions, and Young tableaux. See Section 19.1.10 (page 661) for details.

Combinatorica [PS03] provides Mathematica implementations of algorithms to construct random and sequential integer partitions, compositions, strings, and

Young tableaux, as well as to count and manipulate these objects. See Section 19.1.9 (page 661).

Notes: The best reference on algorithms for generating both integer and set partitions is Knuth [Knu05b]. Good expositions include [KS99, NW78, Rus03, PS03]. Andrews [And98] is the primary reference on integer partitions and related topics, with [AE04] his more accessible introduction.

Integer and set partitions are both special cases of *multiset partitions*, or set partitions of nonnecessarily distinct numbers. In particular, the distinct set partitions on the multiset $\{1, 1, 1, \ldots, 1\}$ correspond exactly to integer partitions. Multiset partitions are discussed in Knuth [Knu05b].

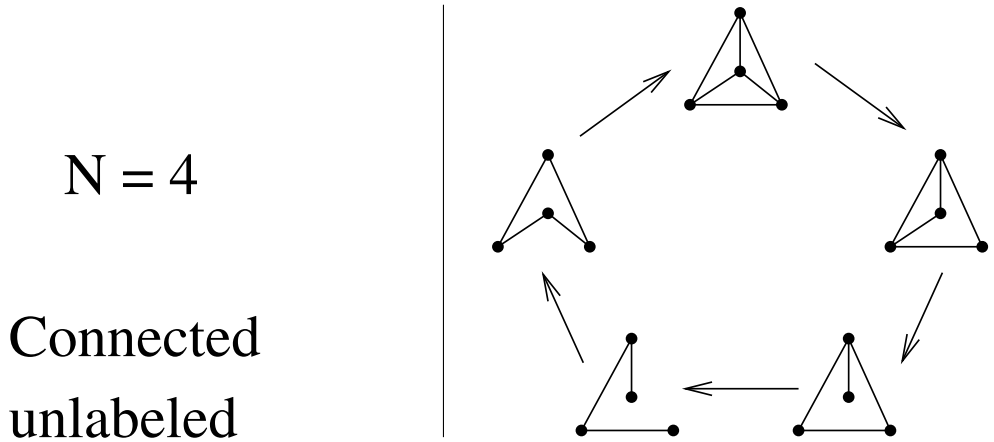
The (long) history of combinatorial object generation is detailed by Knuth [Knu06]. Particularly interesting are connections between set partitions and a Japanese incense burning game, and the naming of all 52 set partitions for $n = 5$ with distinct chapters from the oldest novel known, *The Tale of Genji*.

Two related combinatorial objects are Young tableaux and integer compositions, although they are less likely to emerge in applications. Generation algorithms for both are presented in [NW78, Rus03, PS03].

Young tableaux are two-dimensional configurations of integers $\{1, \ldots, n\}$ where the number of elements in each row is defined by an integer partition of n. Further, the elements of each row and column are sorted in increasing order, and the rows are left-justified. This notion of shape captures a wide variety of structures as special cases. They have many interesting properties, including the existence of a bijection between pairs of tableaux and permutations.

Compositions represent the possible assignments of a set of n indistinguishable balls to k distinguishable boxes. For example, we can place three balls into two boxes as {3,0}, {2,1}, {1,2}, or {0,3}. Compositions are most easily constructed sequentially in lexicographic order. To construct them randomly, pick a random $(k-1)$-subset of $n+k-1$ items using the algorithm of Section 14.5 (page 452), and count the number of unselected items between the selected ones. For example, if $k = 5$ and $n = 10$, the $(5-1)$ subset {1,3,7,14} of $1, \ldots, (n+k-1) = 14$ defines the composition {0,1,3,6,0}, since there are no items to the left of element 1 nor right of element 14.

Related Problems: Generating permutations (see page 448), generating subsets (see page 452).



INPUT OUTPUT

14.7 Generating Graphs

Input description: Parameters describing the desired graph, including the number of vertices n, and the number of edges m or edge probability p.

Problem description: Generate (1) all, or (2) a random, or (3) the next graph satisfying the parameters.

Discussion: Graph generation typically arises in constructing test data for programs. Perhaps you have two different programs that solve the same problem, and you want to see which one is faster or make sure that they always give the same answer. Another application is experimental graph theory, verifying whether a particular property is true for all graphs or how often it is true. It is much easier to believe the four-color theorem after you have demonstrated four colorings for all planar graphs on 15 vertices.

A different application of graph generation arises in network design. Suppose you need to design a network linking ten machines using as few cables as possible, such that this network can survive up to two vertex failures. One approach is to test all the networks with a given number of edges until you find one that will work. For larger graphs, heuristic approaches like simulated annealing will likely be necessary.

Many factors complicate the problem of generating graphs. First, make sure you know exactly what types of graphs you want to generate. Figure 5.2 on page 147 illustrates several important properties of graphs. For purposes of generation, the most important questions are:

- *Do I want labeled or unlabeled graphs?* – The issue here is whether the names of the vertices matter in deciding whether two graphs are the same. In generating *labeled graphs*, we seek to construct all possible labelings of all possible graph topologies. In generating *unlabeled graphs*, we seek only one representative for each topology and ignore labelings. For example, there are only two connected unlabeled graphs on three vertices—a triangle and a simple path. However, there are four connected labeled graphs on three vertices—one triangle and three 3-vertex paths, each distinguished by the name of their central vertex. In general, labeled graphs are much easier to generate. However, there are so many more of them that you quickly get swamped with isomorphic copies of the same few graphs.

- *Do I want directed or undirected graphs?* – Most natural generation algorithms generate undirected graphs. These can be turned into directed graphs by flipping coins to orient the edges. Any graph can be oriented to be directed and acyclic (i.e. , a DAG) by randomly permuting the vertices on a line and aiming each edge from left to right. With all such ideas, careful thought must be given to decide whether you are generating all graphs uniformly at random, and how much this matters to you.

You also must define what you mean by random. There are three primary models of random graphs, all of which generate graphs according to different probability distributions:

- *Random edge generation* – The first model is parameterized by a given edge probability p. Typically, $p = 0.5$, although smaller values can be used to construct sparser random graphs. In this model, a coin is flipped for each pair of vertices x and y to decide whether to add an edge (x, y). All *labeled* graphs will be generated with equal probability when $p = 1/2$.

- *Random edge selection* – The second model is parameterized by the desired number of edges m. It selects m distinct edges uniformly at random. One way to do this is by drawing random (x, y)-pairs and creating an edge if that pair is not already in the graph. An alternative approach to computing the same things constructs the set of $\binom{n}{2}$ possible edges and selects a random m-subset of them, as discussed in Section 14.5 (page 452).

- *Preferential attachment* – Under a rich-get-richer model, newly created edges are likely to point to high-degree vertices than low-degree ones. Consider new links (edges) being added to the graph of webpages. Under any realistic web generation model, it is much more likely the next link will be to Google than *http://www.cs.sunysb.edu/∼algorith.*[1] Selecting the next neighbor with

[1]Please link to us from your homepage to correct for this travesty.

probability proportional to its degree yields graphs with *power law* properties encountered in many real networks.

Which of these options best models your application? Probably none of them. Random graphs have very little structure by definition. But graphs are used to model relationships, which are often highly structured. Experiments conducted on random graphs, although interesting and easy to perform, often fail to capture what you are looking for.

An alternative to random graphs is "organic" graphs—graphs that reflect the relationships among real-world objects. The Stanford GraphBase, discussed below, is an outstanding source of organic graphs. Many raw sources of relationships are available on the web that can be turned into interesting organic graphs with a little programming and imagination. Consider the graph defined by a set of web-pages, with any hyperlink between two pages defining an edge. Or, what about the graph implicit in railroad, subway, or airline networks, with vertices being stations and edges between two stations connected by direct service? As a final example, every large computer program defines a call graph, where the vertices represent subroutines, and there is an edge (x, y) if x calls y.

Two classes of graphs have particularly interesting generation algorithms:

- *Trees* – Prüfer codes provide a simple way to rank and unrank *labeled* trees and thus solve all standard generation problems (see Section 14.4 (page 448)). There are exactly n^{n-2} labeled trees on n vertices, and exactly that many strings of length $n-2$ on the alphabet $\{1, 2, \ldots, n\}$.

  The key to Prüfer's bijection is the observation that every tree has at least two vertices of degree 1. Thus, in any labeled tree the vertex v incident on the leaf with lowest label is well defined. We take v to be S_1, the first character in the code. We then delete the associated leaf and repeat the procedure until only two vertices are left. This defines a unique code S for any given labeled tree that can be used to rank the tree. To go from code to tree, observe that the degree of vertex v in the tree is one more than the number of times v occurs in S. The lowest-labeled leaf will be the smallest integer missing from S, which when paired with S_1 determines the first edge of the tree. The entire tree follows by induction.

  Algorithms for efficiently generating unlabeled rooted trees are discussed in the Implementation section.

- *Fixed degree sequence graphs* – The *degree sequence* of a graph G is an integer partition $p = (p_1, \ldots, p_n)$, where p_i is the degree of the ith highest-degree vertex of G. Since each edge contributes to the degree of two vertices, p is an integer partition of $2m$, where m is the number of edges in G.

  Not all partitions correspond to degree sequences of graphs. However, there is a recursive construction that constructs a graph with a given degree sequence if one exists. If a partition is realizable, the highest-degree vertex v_1 can be

connected to the next p_1 highest-degree vertices in G, or the vertices corresponding to parts $p_2, \ldots, p_{p_1+1}$. Deleting p_1 and decrementing $p_2, \ldots, p_{p_1+1}$ yields a smaller partition, which we recur on. If we terminate without ever creating negative numbers, the partition was realizable. Since we always connect the highest-degree vertex to other high-degree vertices, it is important to reorder the parts of the partition by size after each iteration.

Although this construction is deterministic, a semi-random collection of graphs realizing this degree sequence can be generated from G using *edge-flipping* operations. Suppose edges (x, y) and (w, z) are in G, but (x, w) and (y, z) are not. Exchanging these pairs of edges creates a different (not necessarily connected) graph without changing the degrees of any vertex.

Implementations: The Stanford GraphBase [Knu94] is perhaps most useful as an instance generator for constructing graphs to serve as test data for other programs. It incorporates graphs derived from interactions of characters in famous novels, Roget's Thesaurus, the Mona Lisa, expander graphs, and the economy of the United States. It also contains routines for generating binary trees, graph products, line graphs, and other operations on basic graphs. Finally, because of its machine-independent, random-number generators, it provides a way to construct random graphs such that they can be reconstructed elsewhere, thus making them perfect for experimental comparisons of algorithms. See Section 19.1.8 (page 660) for additional information.

Combinatorica [PS03] provides Mathematica generators for such graphs as stars, wheels, complete graphs, random graphs and trees, and graphs with a given degree sequence. Further, it includes operations to construct more interesting graphs from these, including join, product, and line graph.

The *Combinatorial Object Server* (*http://theory.cs.uvic.ca/*) developed by Frank Ruskey of the University of Victoria provides routines for generating both free and rooted trees.

Viger [VL05] has made available a C++ implementation of his algorithm to generate simple connected graphs with a prescribed degree sequence. See *http://www.liafa.jussieu.fr/~fabien/generation/*.

The graph isomorphism testing program Nauty (see Section 16.9 (page 550)) includes a suite of programs for generating nonisomorphic graphs, plus special generators for bipartite graphs, digraphs, and multigraphs. They are available at *http://cs.anu.edu.au/~bdm/nauty/*.

The mathematicians Brendan McKay (*http://cs.anu.edu.au/~bdm/data/*) and Gordon Royle (*http://people.csse.uwa.edu.au/gordon/data.html*) provide exhaustive catalogs of several families of graphs and trees up to the largest reasonable number of vertices.

Nijenhuis and Wilf [NW78] provide efficient Fortran routines to enumerate all labeled trees via Prüfer codes and to construct random unlabeled rooted trees. See Section 19.1.10 (page 661). Kreher and Stinson [KS99] generate labeled trees in C,

with implementations available at *http://www.math.mtu.edu/~kreher/cages/Src.html*.

Notes: Extensive literature exists on generating graphs uniformly at random. Surveys include [Gol93, Tin90]. Closely related to the problem of generating classes of graphs is counting them. Harary and Palmer [HP73] survey results in graphical enumeration.

Knuth [Knu06] is the best recent reference on generating trees. The bijection between $n-2$ strings and labeled trees is due to Prüfer [Prü18].

Random graph theory is concerned with the properties of random graphs. Threshold laws in random graph theory define the edge density at which properties such as connectedness become highly likely to occur. Expositions on random graph theory include [Bol01, JLR00].

The preferential attachment model of graphical evolution has emerged relatively recently in the study of networks. See [Bar03, Wat04] for introductions to this exciting field.

An integer partition is *graphic* if there exists a simple graph with that degree sequence. Erdős and Gallai [EG60] proved that a degree sequence is graphic if and only if the sequence observes the following condition for each integer $r < n$:

$$\sum_{i=1}^{r} d_i \leq r(r-1) + \sum_{i=r+1}^{n} \min(r, d_i)$$

Related Problems: Generating permutations (see page 448), graph isomorphism (see page 550).

INPUT	OUTPUT
December 21, 2012 ? (Gregorian)	5773 Teveth 8 (Hebrew) 1434 Safar 7 (Islamic) 1934 Agrahayana 30 (Indian Civil) 13.0.0.0 (Mayan Long Count)

14.8 Calendrical Calculations

Input description: A particular calendar date d, specified by month, day, and year.

Problem description: Which day of the week did d fall on according to the given calendar system?

Discussion: Many business applications need to perform calendrical calculations. Perhaps we want to display a calendar of a specified month and year. Maybe we need to compute what day of the week or year some event occurs, as in figuring out the date on which a 180-day futures contract comes due. The importance of correct calendrical calculations was perhaps best revealed by the furor over the "Millennium bug"—the year 2000 crisis in legacy programs that allocated only two digits for storing the year.

More complicated questions arise in international applications, because different nations and ethnic groups use different calendar systems. Some of these, like the Gregorian calendar used in most of the world, are based on the Sun, while others, like the Hebrew calendar, are lunar calendars. How would you tell today's date according to the Chinese or Islamic calendar?

Calendrical calculations differ from other problems in this book because calendars are historical objects, not mathematical ones. The algorithmic issues revolve around specifying the rules of the calendrical system and implementing them correctly, rather than designing efficient shortcuts for the computation.

The basic approach underlying calendar systems is to start with a particular reference date (called the *epoch*) and count up from there. The particular rules for wrapping the count around into months and years is what distinguishes one system from another. Implementing a calendar requires two functions, one that, given a date, returns the integer number of days that have elapsed since the epoch, the other of which takes an integer n and returns the calendar date exactly n days

from epoch. These are exactly analogous to the ranking and unranking rules for combinatorial objects such as permutations (see Section 14.4 (page 448)).

That the solar year is not an integer number of days long is the major source of complications in calendar systems. To keep a calendar's annual dates in sync with the seasons, leap days must be added at both regular and irregular intervals. Since a solar year is 365 days and 5:49:12 hours long, adding a leap day every four years leaves an extra 10 minutes and 48 seconds unaccounted for each year.

The original Julian calendar (from Julius Caesar) ignored these extra minutes, which had accumulated to ten days by 1582. Pope Gregory XIII then proposed the Gregorian calendar used today, by deleting the ten days and eliminating leap days in years that are multiples of 100 but not 400. Supposedly, riots ensued because the masses feared their lives were being shortened by ten days. Outside the Catholic church, resistance to change slowed the reforms. The deletion of days did not occur in England and America until September 1752, and not until 1927 in Turkey.

The rules for most calendrical systems are sufficiently complicated and pointless that you should lift code from a reliable place rather than attempt to write your own. We identify suitable implementations next.

There are a variety of "impress your friends" algorithms that enable you to compute in your head what day of the week a particular date occurred. Such algorithms often fail to work reliably outside the given century and certainly should be avoided for computer implementation.

Implementations: Readily available calendar libraries exist in both C++ and Java. The Boost time-data library provides a reliable implementation of the Gregorian calendar in C++. See *http://www.boost.org/doc/html/date_time.html*. The Calendar class in `java.util.Calendar` implements the Gregorian calendar in Java. Either of these will likely suffice for most applications.

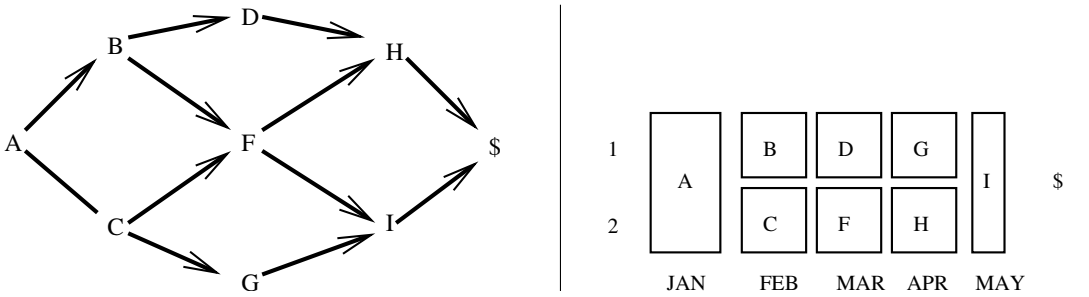
Dershowitz and Reingold provide a uniform algorithmic presentation [RD01] for a variety of different calendar systems, including the Gregorian, ISO, Chinese, Hindu, Islamic, and Hebrew calendars, as well as other calendars of historical interest. *Calendrical* is an implementation of these calendars in Common Lisp, Java, and Mathematica, with routines to convert dates between calendars, day of the week computations, and the determination of secular and religious holidays. *Calendrical* is likely to be the most comprehensive and reliable calendrical routines available. See their website at *http://calendarists.com*.

C and Java implementations of international calendars of unknown reliability are readily available at SourceForge (*http://sourceforge.net*). Search for "Gregorian calendar" to avoid the mass of datebook implementations.

Notes: A comprehensive discussion of calendrical computation algorithms appear in the papers of Dershowitz and Reingold [DR90, RDC93], which have been superseded by their book [RD01] that outlines algorithms for no less than 25 international and historical calendars Three hundred years of calendars representing tabulations for all dates from 1900 to 2200 appear in [DR02].

Concern exists in certain quarters whether December 21, 2012 represents the end of the world. The argument rests on it being the date the Mayan calendar spins over to 13.0.0.0.0 after a 5,125 year cycle. The reader should rest assured, since I would never have devoted so much effort to writing this book were the world to be ending so imminently. The Mayan calendar is authoritatively described in [RD01].

Related Problems: Arbitrary-precision arithmetic (see page 423), generating permutations (see page 448).



INPUT OUTPUT

14.9 Job Scheduling

Input description: A directed acyclic graph $G = (V, E)$, where vertices represent jobs and edge (u, v) implies that task u must be completed before task v.

Problem description: What schedule of tasks completes the job using the minimum amount of time or processors?

Discussion: Devising a proper schedule to satisfy a set of constraints is fundamental to many applications. Mapping tasks to processors is a critical aspect of any parallel-processing system. Poor scheduling can leave all other machines sitting idle while one bottleneck task is performed. Assigning people-to-jobs, meetings-to-rooms, or courses-to-exam periods are all different examples of scheduling problems.

Scheduling problems differ widely in the nature of the constraints that must be satisfied and the type of schedule desired. Several other catalog problems have application to various kinds of scheduling:

- Topological sorting can construct a schedule consistent with the precedence constraints. See Section 15.2 (page 481).
- Bipartite matching can assign a set of jobs to people who have the appropriate skills for them. See Section 15.6 (page 498).
- Vertex and edge coloring can assign a set of jobs to time slots such that no two interfering jobs are assigned the same time slot. See Sections 16.7 and 16.8.
- Traveling salesman can schedule select the most efficient route for a delivery person to visit a given set of locations. See Section 16.4 (page 533).

- Eulerian cycle can construct the most efficient route for a snowplow or mailman to completely traverse a given set of edges. See Section 15.7 (page 502).

Here we focus on precedence-constrained scheduling problems for directed acyclic graphs. Suppose you have broken a big job into a large number of smaller tasks. For each task, you know how long it should take (or perhaps an upper bound on how long it might take). Further, for each pair of tasks you know whether it is essential that one task be performed before the other. The fewer constraints we have to enforce, the better our schedule can be. These constraints must define a directed acyclic graph—acyclic because a cycle in the precedence constraints represents a Catch-22 situation that can never be resolved.

We are interested in several problems on these networks:

- *Critical path* – The longest path from the start vertex to the completion vertex defines the *critical path*. This can be important to know, for the only way to shorten the minimum total completion time is to reduce the time of one of the tasks on each critical path. The tasks on the critical paths can be determined in $O(n + m)$ time using the dynamic programming presented in Section 15.4 (page 489).

- *Minimum completion time* – What is the fastest we can get this job completed while respecting precedence constraints, assuming that we have an unlimited number of workers. If there were *no* precedence constraints, each task could be worked on by its own worker, and the total time would be that of the longest single task. If there are such strict precedence constraints that each task must follow the completion of its immediate predecessor, the minimum completion time would be obtained by summing up the times for each task.

  The minimum completion time for a DAG can be computed in $O(n+m)$ time. The completion time is defined by the critical path. To get such a schedule, consider the jobs in topological order, and start each job on a new processor the moment its latest prerequisite completes.

- *What is the tradeoff between the number of workers and completion time?* – What we really are interested in is how best to complete the schedule with a given number of workers. Unfortunately, this and most similar problems are NP-complete.

Any real scheduling application is likely to present combinations of constraints that will be difficult or impossible to model using these techniques. There are two reasonable ways to deal with such problems. First, we can ignore constraints until the problem reduces to one of the types that we have described here, solve it, and then see how bad it is with respect to the other constraints. Perhaps the schedule can be easily modified by hand to satisfy other esoteric constraints, like keeping Joe and Bob apart so they won't kill each other. Another approach is to formulate

your scheduling problem via linear-integer programming (see Section 13.6 (page 411)) in all its complexity. I always recommend starting out with something simple and see how it works before trying something complicated.

Another fundamental scheduling problem takes a set of jobs without precedence constraints and assigns them to identical machines to minimize the total elapsed time. Consider a copy shop with k Xerox machines and a stack of jobs to finish by the end of the day. Such tasks are called *job-shop scheduling*. They can be modeled as bin-packing (see Section 17.9 (page 595)), where each job is assigned a number equal to the number of hours it will take to complete, and each machine is represented by a bin with space equal to the number of hours in a day.

More sophisticated variations of job-shop scheduling provide each task with allowable start and required finishing times. Effective heuristics are known, based on sorting the tasks by size and finishing time. We refer the reader to the references for more information. Note that these scheduling problems become hard only when the tasks cannot be broken up onto multiple machines or interrupted (preempted) and then rescheduled. If your application allows these degrees of freedom, you should exploit them.

Implementations: JOBSHOP is a collection of C programs for job-shop scheduling created in the course of a computational study by Applegate and Cook [AC91]. They are available at *http://www2.isye.gatech.edu/∼wcook/jobshop/*.

Tablix (*http://tablix.org*) is an open source program for solving timetabling problems faced by real schools. Support for parallel/cluster computation is provided.

LEKIN is a flexible job-shop scheduling system designed for educational use [Pin02]. It supports single machine, parallel machines, flow-shop, flexible flow-shop, job-shop, and flexible job-shop scheduling, and is available at *http://www.stern.nyu.edu/om/software/lekin*.

For commercial scheduling applications, ILOG CP is reflective of the state-of-the-art. For more, see *http://www.ilog.com/products/cp/*.

Algorithm 520 [WBCS77] of the *Collected Algorithms of the ACM* is a Fortran code for multiple-resource network scheduling. It is available from Netlib (see Section 19.1.5 (page 659)).

Notes: The literature on scheduling algorithms is a vast one. Brucker [Bru07] and Pinedo [Pin02] provide comprehensive overviews of the field. The *Handbook of Scheduling* [LA04] provides an up-to-date collection of surveys on all aspects of scheduling.

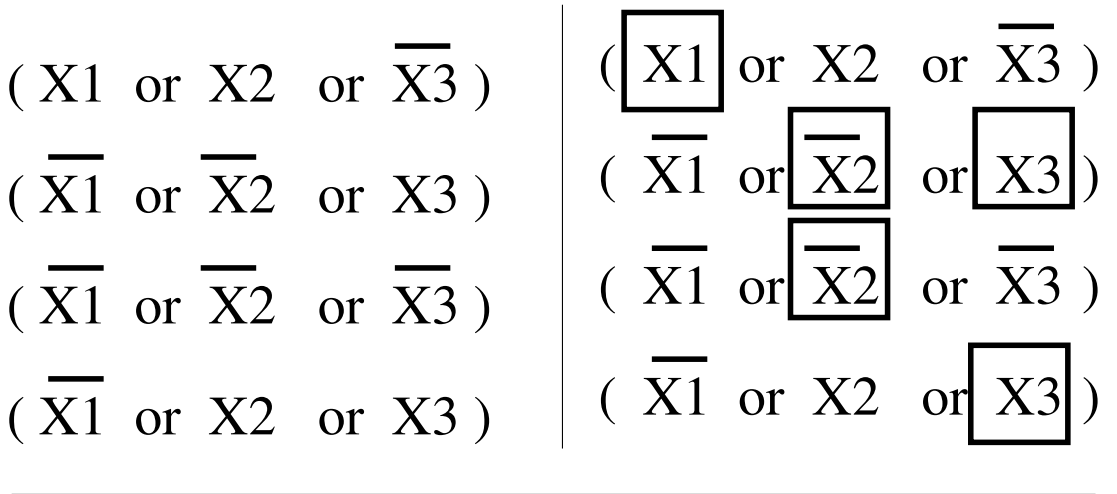
A well-defined taxonomy exists covering thousands of job-shop scheduling variants, which classifies each problem $\alpha|\beta|\gamma$ according to (α) the machine environment, (β) details of processing characteristics and constraints, and (γ) the objectives to be minimized. Surveys of results include [Bru07, CPW98, LLK83, Pin02].

Gantt charts provide visual representations of job-shop scheduling solutions, where the x-axis represents time and rows represent distinct machines. The output figure above illustrates a Gantt chart. Each scheduled job is represented as a horizontal block, thus identifying its start-time, duration, and server. Project precedence-constrained scheduling

techniques are often called PERT/CPM, for *Program Evaluation and Review Technique/Critical Path Method*. Both Gantt charts and PERT/CPM appear in most textbooks on operations research, including [Pin02].

Timetabling is a term often used in discussion of classroom and related scheduling problems. PATAT (for *Practice and Theory of Automated Timetabling*) is a biannual conference reporting new results in the field. Its proceedings are available from *http://www.asap.cs.nott.ac.uk/patat/patat-index.shtml*.

Related Problems: Topological sorting (see page 481), matching (see page 498), vertex coloring (see page 544), edge coloring (see page 548), bin packing (see page 595).



14.10 Satisfiability

Input description: A set of clauses in conjunctive normal form.

Problem description: Is there a truth assignment to the Boolean variables such that every clause is simultaneously satisfied?

Discussion: Satisfiability arises whenever we seek a configuration or object that must be consistent with (i.e. , satisfy) a set of logical constraints. A primary application is in verifying that a hardware or software system design works correctly on all inputs. Suppose a given logical formula $S(\bar{X})$ denotes the specified result on input variables $\bar{X} = X_1, \ldots, X_n$, while a different formula $C(\bar{X})$ denotes the Boolean logic of a proposed circuit for computing $S(\bar{X})$. This circuit is correct unless there exists an $\bar{X}$ such that $S(\bar{X}) \neq C(\bar{X})$.

Satisfiability (or SAT) is *the* original NP-complete problem. Despite its applications to constraint satisfaction, logic, and automatic theorem proving, it is perhaps most important theoretically as the root problem from which all other NP-completeness proofs originate. So much engineering has gone into today's best SAT solvers that they represent a reasonable starting point if one really needs to solve an NP-complete problem *exactly*. That said, employing heuristics that give good but nonoptimal solutions is usually the better approach in practice.

Issues in satisfiability testing include:

- *Is your formula the AND of ORs (CNF) or the OR of ANDs (DNF)?* – In satisfiability, constraints are specified as a logical formula. There are two

primary ways of expressing logical formulas—conjunctive normal form (CNF) and disjunctive normal form (DNF). In CNF formulas, we must satisfy all clauses, where each clause is constructed by and-ing or's of literals together, such as

$$(v_1 \text{ or } \bar{v_2}) \text{ and } (v_2 \text{ or } v_3)$$

In DNF formulas, we must satisfy any one clause, where each clause is constructed by or-ing ands of literals together. The formula above can be written in DNF as

$$(\bar{v_1} \text{ and } \bar{v_2} \text{ and } v_3) \text{ or } (\bar{v_1} \text{ and } v_2 \text{ and } \bar{v_3}) \text{ or } (\bar{v_1} \text{ and } v_2 \text{ and } v_3) \text{ or } (v_1 \text{ and } \bar{v_2} \text{ and } v_3)$$

Solving DNF-satisfiability is trivial, since any DNF formula can be satisfied unless *every* clause contains both a literal and its complement (negation). However, CNF-satisfiability is NP-complete. This seems paradoxical, since we can use De Morgan's laws to convert CNF-formulae into equivalent DNF-formulae, and vice versa. The catch is that an exponential number of terms might be constructed in the course of translation, so that the translation itself does not run in polynomial time.

- *How big are your clauses?* – k-SAT is a special case of satisfiability when each clause contains at most k literals. The problem of 1-SAT is trivial, since we must set true any literal appearing in any clause. The problem of 2-SAT is not trivial, but can still be solved in linear time. This is interesting, because certain problems can be modeled as 2-SAT using a little cleverness. The good times end as soon as clauses contain three literals each (i.e. , 3-SAT) for 3-SAT is NP-complete.

- *Does it suffice to satisfy* most *of the clauses?* – If you must solve it exactly, there is not much you can do to solve satisfiability except by backtracking algorithms such as the Davis-Putnam procedure. In the worst case, there are 2^m truth assignments to be tested, but fortunately there are many ways to prune the search. Although satisfiability is NP-complete, how hard it is in practice depends on how the instances are generated. Naturally defined "random" instances are often surprisingly easy to solve, and in fact it is nontrivial to generate instances of the problem that are truly hard.

  Still, we can benefit by relaxing the problem so that the goal becomes satisfying as many clauses as possible. Optimization techniques such as simulated annealing can then be put to work to refine random or heuristic solutions. Indeed, any random truth assignment to the variables will satisfy each k-SAT clause with probability $1 - (1/2)^k$, so our first attempt is likely to satisfy most of the clauses. Finishing off the job is the hard part. Finding an assignment that satisfies the maximum number of clauses is NP-complete even for nonsatisfiable instances.

When faced with a problem of unknown complexity, proving it NP-complete can be an important first step. If you think your problem might be hard, skim through Garey and Johnson [GJ79] looking for your problem. If you don't find it, I recommend that you put the book away and try to prove hardness from first principles, using the basic problems of 3-SAT, vertex cover, independent set, integer partition, clique, and Hamiltonian cycle. I find it much easier to start from these than some complicated problem in the book, and more insightful too, since the reason for the hardness is not obscured by the hidden hardness proof for the complicated problem. Chapter 9 focuses on strategies for proving hardness.

Implementations: Recent years have seen tremendous progress in the performance of SAT solvers. An annual SAT competition identifies the top performing solvers in each of three categories of instances (drawn from industrial, handmade, and random problem instances, respectively).

The top three programs of the SAT 2007 industrial competition were Rsat (*http://reasoning.cs.ucla.edu/rsat/*), PicoSAT (*http://fmv.jku.at/picosat/*) and MiniSAT (*http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/*). The source code for all these solvers and more are available from the competition webpage (*http://www.satcompetition.org/*).

SAT *Live!* (*http://www.satlive.org/*) is the most up-to-date source for papers, programs, and test sets for satisfiability and related logic optimization problems.

Notes: The most comprehensive overview of satisfiability testing is Kautz, et al. [KSBD07]. The Davis-Putnam-Logemann-Loveland (DPLL) algorithm is a backtracking algorithm introduced in 1962 for solving satisfiability problems. Local search techniques work better on certain classes of problems that are difficult for DPLL solvers. *Chaff* [MMZ$^+$01] is a particularly influential solver, available at *http://www.princeton.edu/∼chaff/*. See [KS07] for a survey of recent progress in the field of satisfiability testing.

An algorithm for solving 3-SAT in worst-case $O^*(1.4802^n)$ appears in [DGH$^+$02]. Efficient (but nonpolynomial) algorithms for NP-complete problems are surveyed in [Woe03].

The primary reference on NP-completeness is [GJ79], featuring a list of roughly 400 NP-complete problems. The book remains an extremely useful reference; it is perhaps the book I reach for most often. An occasional column by David Johnson in the *Journal of Algorithms* and (now) *ACM Transactions on Algorithms* provides updates.

Good expositions of Cook's theorem [Coo71], where satisfiability is proven hard, include [CLRS01, GJ79, KT06]. The importance of Cook's result became clear in Karp's paper [Kar72], showing the hardness of over 20 different combinatorial problems.

A linear-time algorithm for 2-SAT appears in [APT79]. See [WW95] for an interesting application of 2-SAT to map labeling. The best heuristic known approximates maximum 2-SAT to within a factor of 1.0741 [FG95].

Related Problems: Constrained optimization (see page 407), traveling salesman problem (see page 533).

15

Graph Problems: Polynomial-Time

Algorithmic graph problems constitute approximately one third of the problems in this catalog. Problems from other sections could have been formulated equally well in terms of graphs, such as bandwidth minimization and finite-state automata optimization. Identifying the name of a graph-theoretic invariant or problem is one of the primary skills of a good algorist. Indeed, the catalog will tell you exactly how to proceed as soon as you figure out your particular problem's name.

In this section, we deal only with problems for which there are efficient algorithms to solve them. As there is often more than one way to model a given application, it makes sense to look here before proceeding on to the harder formulations.

The algorithms presented here have running times that grow polynomially with the size of the graph. We adopt throughout the convention that n refers to the number of vertices in a graph, while m is the number of edges.

Graphs are often best understood as drawings. Many interesting graph properties follow from the nature of a particular type of drawing, such as planar graphs. Thus, we also discuss algorithms for drawing graphs, trees, and planar graphs.

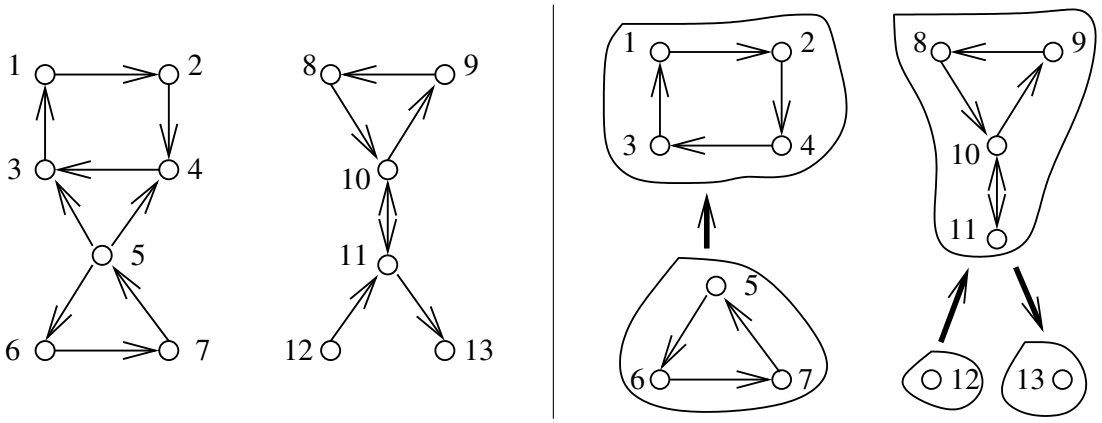
Most advanced graph algorithms are difficult to program. However, good implementations are available if you know where to look. The best general sources include LEDA [MN99] and the Boost Graph Library [SLL02]. However, better special-purpose codes exist for many problems.

See the *Handbook of Graph Algorithms* [TNX08] for up-to-date surveys on all areas of graph algorithms. Other excellent surveys include van Leeuwen [vL90a], and several chapters in Atallah [Ata98]. Books specializing in graph algorithms include:

- *Sedgewick* [Sed98] – The graph algorithms volume of this algorithms text provides a comprehensive but gentle introduction to the field.

S.S. Skiena, *The Algorithm Design Manual*, 2nd ed., DOI: 10.1007/978-1-84800-070-4_15,

- *Ahuja, Magnanti, and Orlin* [AMO93] – While purporting to be a book on network flows, it covers the gamut of graph algorithms with an emphasis on operations research. Strongly recommended.
- *Gibbons* [Gib85] – A good book on graph algorithms, including planarity testing, matching, and Eulerian/Hamiltonian cycles, as well as more elementary topics.
- *Even* [Eve79a] – An older but still respected advanced text on graph algorithms, with a particularly thorough treatment of planarity-testing algorithms.



INPUT OUTPUT

15.1 Connected Components

Input description: A directed or undirected graph G.

Problem description: Identify the different pieces or components of G, where vertices x and y are members of different components if no path exists from x to y in G.

Discussion: The connected components of a graph represent, in grossest terms, the pieces of the graph. Two vertices are in the same component of G if and only if there exists some path between them.

Finding connected components is at the heart of many graph applications. For example, consider the problem of identifying natural clusters in a set of items. We represent each item by a vertex and add an edge between each pair of items deemed "similar." The connected components of this graph correspond to different classes of items.

Testing whether a graph is connected is an essential preprocessing step for every graph algorithm. Subtle, hard-to-detect bugs often result when an algorithm is run only on one component of a disconnected graph. Connectivity tests are so quick and easy that you should always verify the integrity of your input graph, even when you know for certain that it *has* to be connected.

Testing the connectivity of any undirected graph is a job for either depth-first or breadth-first search, as discussed in Section 5 (page 145). Which one you choose doesn't really matter. Both traversals initialize the *component-number* field for each vertex to 0, and then start the search for component 1 from vertex v_1. As each vertex is discovered, the value of this field is set to the current component

number. When the initial traversal ends, the component number is incremented, and the search begins again from the first vertex whose *component-number* remains 0. Properly implemented using adjacency lists (as is done in Section 5.7.1 (page 166)) this runs in $O(n+m)$ time.

Other notions of connectivity also arise in practice:

- *What if my graph is directed?* – There are two distinct notions of connected components for directed graphs. A directed graph is *weakly connected* if it would be connected by ignoring the direction of edges. Thus, a weakly connected graph consists of a single piece. A directed graph is *strongly connected* if there is a directed path between every pair of vertices. This distinction is best made clear by considering the network of one- and two-way streets in any city. The network is strongly connected if it is possible to drive legally between every two positions. The network is weakly connected when it is possible to drive legally or *illegally* between every two positions. The network is disconnected if there is no possible way to drive from a to b.

  Weakly and strongly connected components define unique partitions of the vertices. The output figure at the beginning of this section illustrates a directed graph consisting of two weakly or five strongly-connected components (also called *blocks* of G).

  Testing whether a directed graph is weakly connected can be done easily in linear time. Simply turn all edges of G into undirected edges and use the DFS-based connected components algorithm described previously. Tests for strong connectivity are somewhat more complicated. The simplest linear-time algorithm performs a search from some vertex v to demonstrate that the entire graph is reachable from v. We then construct a graph G' where we reverse all the edges of G. A traversal of G' from v suffices to decide whether all vertices of G can reach v. Graph G is strongly connected iff all vertices can reach, and are reachable, from v.

  All the strongly connected components of G can be extracted in linear time using more sophisticated DFS-based algorithms. A generalization of the above "two-DFS" idea is deceptively easy to program but somewhat subtle to understand exactly why it works:

  1. Perform a DFS, starting from an arbitrary vertex in G, and labeling each vertex in order of its completion (not discovery).
  2. Reverse the direction of each edge in G, yielding G'.
  3. Perform a DFS of G', starting from the highest numbered vertex in G. If this search does not completely traverse G', continue with the highest numbered unvisited vertex.
  4. Each DFS tree created in Step 3 is a strongly connected component.

My implementation of a single-pass algorithm appears in Section 5.10.2 (page 181). In either case, it is probably easier to start from an existing implementation than a textbook description.

- *What is the weakest point in my graph/network?* – A chain is only as strong as its weakest link. Losing one or more internal links causes a chain to become disconnected. The *connectivity* of graphs measures their strength—how many edges or vertices must be removed to disconnect it. Connectivity is an essential invariant for network design and other structural problems.

  Algorithmic connectivity problems are discussed in Section 15.8 (page 505). In particular, *biconnected components* are pieces of the graph that result from cutting the edges incident on a single vertex. All biconnected components can be found in linear time using DFS. See Section 5.9.2 (page 173) for an implementation of this algorithm. Vertices whose deletion disconnects the graph belong to more than one biconnected component, whose edges are uniquely partitioned by components.

- *Is the graph a tree? How can I find a cycle if one exists?* – The problem of cycle identification often arises, particularly with respect to directed graphs. For example, testing if a sequence of conditions can deadlock often reduces to cycle detection. If I am waiting for Fred, and Fred is waiting for Mary, and Mary is waiting for me, there is a cycle and we are all deadlocked.

  For undirected graphs, the analogous problem is tree identification. A tree is, by definition, an undirected, connected graph without any cycles. As described above, a depth-first search can be used to test whether it is connected. If the graph is connected and has $n - 1$ edges for n vertices, it is a tree.

  Depth-first search can be used to find cycles in both directed and undirected graphs. Whenever we encounter a back edge in our DFS—i.e. , an edge to an ancestor vertex in the DFS tree—the back edge and the tree together define a directed cycle. No other such cycle can exist in a directed graph. Directed graphs without cycles are called DAGs (directed acyclic graphs). Topological sorting (see Section 15.2 (page 481)) is the fundamental operation on DAGs.

Implementations: The graph data structure implementations of Section 12.4 (page 381) all include implementations of BFS/DFS, and hence connectivity testing to at least some degree. The C++ Boost Graph Library [SLL02] (*http://www.boost.org/libs/graph/doc*) provides implementations of connected components and strongly connected components. LEDA (see Section 19.1.1 (page 658)) provides these plus biconnected and triconnected components, breadth-first and depth-first search, connected components and strongly connected components, all in C++.

With respect to Java, *JUNG* (*http://jung.sourceforge.net/*) also provides biconnected component algorithms, while JGraphT (*http://jgrapht.sourceforge.net/*) does strongly connected components.

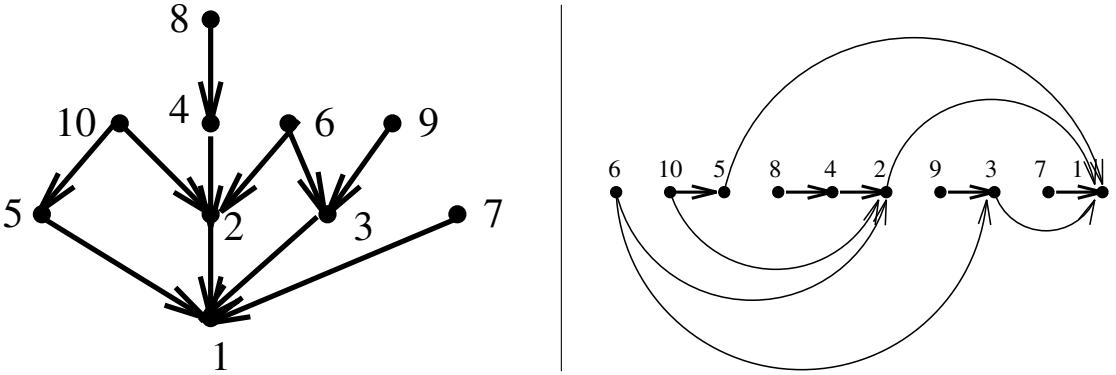
My (biased) preference for C language implementations of all basic graph connectivity algorithms. including strongly connected components and biconnected components, is the library associated with this book. See Section 19.1.10 (page 661) for details.

Notes: Depth-first search was first used to find paths out of mazes, and dates back to the nineteenth century [Luc91, Tar95]. Breadth-first search was first reported to find the shortest path by Moore in 1957 [Moo59].

Hopcroft and Tarjan [HT73b, Tar72] established depth-first search as a fundamental technique for efficient graph algorithms. Expositions on depth-first and breadth-first search appear in every book discussing graph algorithms, with [CLRS01] perhaps the most thorough description available.

The first linear-time algorithm for strongly connected components is due to Tarjan [Tar72], with expositions including [BvG99, Eve79a, Man89]. Another algorithm—simpler to program and slicker—for finding strongly connected components is due to Sharir and Kosaraju. Good expositions of this algorithm appear in [AHU83, CLRS01]. Cheriyan and Mehlhorn [CM96] propose improved algorithms for certain problems on dense graphs, including strongly connected components.

Related Problems: Edge-vertex connectivity (see page 505), shortest path (see page 489).



INPUT OUTPUT

15.2 Topological Sorting

Input description: A directed acyclic graph $G = (V, E)$, also known as a *partial order* or *poset*.

Problem description: Find a linear ordering of the vertices of V such that for each edge $(i, j) \in E$, vertex i is to the left of vertex j.

Discussion: Topological sorting arises as a subproblem in most algorithms on directed acyclic graphs. Topological sorting orders the vertices and edges of a DAG in a simple and consistent way and hence plays the same role for DAGs that a depth-first search does for general graphs.

Topological sorting can be used to schedule tasks under precedence constraints. Suppose we have a set of tasks to do, but certain tasks have to be performed before other tasks. These precedence constraints form a directed acyclic graph, and any topological sort (also known as a *linear extension*) defines an order to do these tasks such that each is performed only after all of its constraints are satisfied.

Three important facts about topological sorting are

1. *Only* DAGs can be topologically sorted, since any directed cycle provides an inherent contradiction to a linear order of tasks.

2. *Every* DAG can be topologically sorted, so there must always be at least one schedule for any reasonable precedence constraints among jobs.

3. DAGs can often be topologically sorted in many different ways, especially when there are few constraints. Consider n unconstrained jobs. Any of the $n!$ permutations of the jobs constitutes a valid topological ordering.

The conceptually simplest linear-time algorithm for topological sorting performs a depth-first search of the DAG to identify the complete set of *source vertices*, where source vertices are vertices of in-degree zero. At least one such source must exist in any DAG. Source vertices can appear at the start of any schedule without violating any constraints. Deleting all the outgoing edges of these source vertices will create new source vertices, which can then sit comfortably to the immediate right of the first set. We repeat until all vertices are accounted for. A modest amount of care with data structures (adjacency lists and queues) is sufficient to make this run in $O(n + m)$ time.

An alternate algorithm makes use of the observation that ordering the vertices in terms of decreasing DFS finishing time yields a linear extension. An implementation of this algorithm with an argument for correctness is given in Section 5.10.1 (page 179).

Two special considerations with respect to topological sorting are:

- *What if I need all the linear extensions, instead of just one of them?* – In certain applications, it is important to construct *all* linear extensions of a DAG. Beware, because the number of linear extensions can grow exponentially in the size of the graph. Even the problem of counting the number of linear extensions is NP-hard.

  Algorithms for listing all linear extensions in a DAG are based on backtracking. They build all possible orderings from left to right, where each of the in-degree zero vertices are candidates for the next vertex. The outgoing edges from the selected vertex are deleted before moving on. An optimal algorithm for listing (or counting) linear extensions is discussed in the notes.

  Algorithms for constructing random linear extensions start from an arbitrary linear extension. We then repeatedly sample pairs of vertices. These are exchanged if the resulting permutation remains a topological ordering. This results in a random linear extension given enough random samples. See the Notes section for details.

- *What if your graph is not acyclic?* – When a set of constraints contains inherent contradictions, the natural problem becomes removing the smallest set of items that eliminates all inconsistencies. The sets of offending jobs (vertices) or constraints (edges) whose deletion leaves a DAG are known as the *feedback vertex set* and the *feedback arc set*, respectively. They are discussed in Section 16.11 (page 559). Unfortunately, both problems are NP-complete.

  Since the topological sorting algorithm gets stuck as soon as it identifies a vertex on a directed cycle, we can delete the offending edge or vertex and continue. This quick-and-dirty heuristic will eventually leave a DAG, but might delete more things than necessary. Section 9.10.3 (page 348) describes a better approach to the problem.

Implementations: Essentially all the graph data structure implementations of Section 12.4 (page 381) include implementations of topological sorting. This means the Boost Graph Library [SLL02] (*http://www.boost.org/libs/graph/doc*) and LEDA (see Section 19.1.1 (page 658)) for C++. For Java, the *Data Structures Library in Java* (JDSL) (*http://www.jdsl.org/*) includes a special routine to compute a unit-weighted topological numbering. Also check out JGraphT (*http://jgrapht.sourceforge.net/*).

The *Combinatorial Object Server* (*http://theory.cs.uvic.ca/*) provides C language programs to generate linear extensions in both lexicographic and Gray code orders, as well as count them. An interactive interface is also provided.

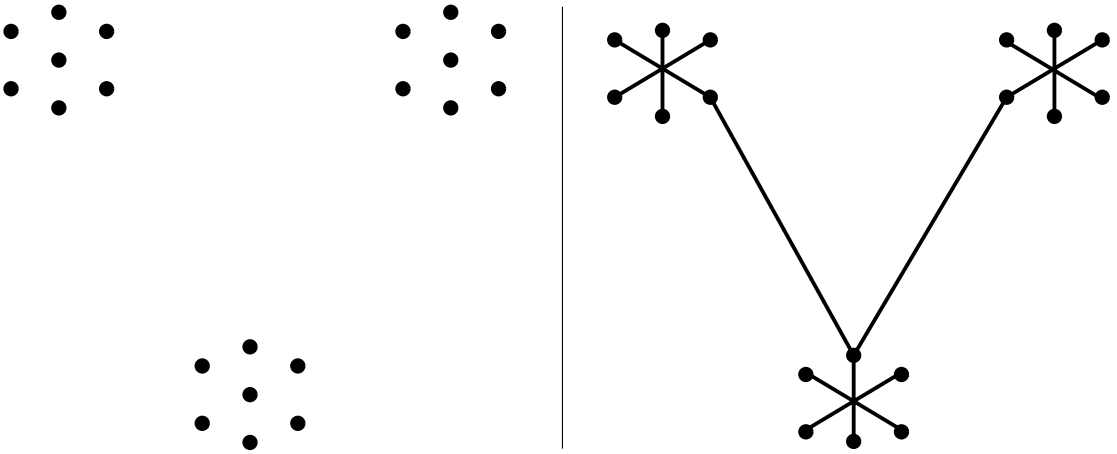
My (biased) preference for C language implementations of all basic graph algorithms, including topological sorting, is the library associated with this book. See Section 19.1.10 (page 661) for details.

Notes: Good expositions on topological sorting include [CLRS01, Man89]. Brightwell and Winkler [BW91] prove that it is #P-complete to count the number of linear extensions of a partial order. The complexity class #P includes NP, so any #P-complete problem is at least NP-hard.

Pruesse and Ruskey [PR86] give an algorithm that generates linear extensions of a DAG in constant amortized time. Further, each extension differs from its predecessor by either one or two adjacent transpositions. This algorithm can be used to count the number of linear extensions $e(G)$ of an n-vertex DAG G in $O(n^2 + e(G))$. Alternately, the reverse search technique of Avis and Fukuda [AF96] can be employed to list linear extensions. A backtracking program to generate all linear extensions is described in [KS74].

Huber [Hub06] gives an algorithm to sample linear extensions uniformly at random from an arbitrary partial order in expected $O(n^3 \lg n)$ time, improving the result of [BD99].

Related Problems: Sorting (see page 436), feedback edge/vertex set (see page 559).



INPUT OUTPUT

15.3 Minimum Spanning Tree

Input description: A graph $G = (V, E)$ with weighted edges.

Problem description: The minimum weight subset of edges $E' \subset E$ that form a tree on V.

Discussion: The minimum spanning tree (MST) of a graph defines the cheapest subset of edges that keeps the graph in one connected component. Telephone companies are interested in minimum spanning trees, because the MST of a set of locations defines the wiring scheme that connects the sites using as little wire as possible. MST is the mother of all network design problems.

Minimum spanning trees prove important for several reasons:

- They can be computed quickly and easily, and create a sparse subgraph that reflects a lot about the original graph.
- They provide a way to identify clusters in sets of points. Deleting the long edges from an MST leaves connected components that define natural clusters in the data set, as shown in the output figure above.
- They can be used to give approximate solutions to hard problems such as Steiner tree and traveling salesman.
- As an educational tool, MST algorithms provide graphic evidence that greedy algorithms can give provably optimal solutions.

Three classical algorithms efficiently construct MSTs. Detailed implementations of two of them (Prim's and Kruskal's) are given with correctness arguments in Section 6.1 (page 192). The third somehow manages to be less well known despite being invented first and (arguably) being both easier to implement and more efficient.

The contenders are

- *Kruskal's algorithm* – Each vertex starts as a separate tree and these trees merges together by repeatedly adding the lowest cost edge that spans two distinct subtrees (i.e. , does not create a cycle).

```
Kruskal(G)
    Sort the edges in order of increasing weight
    count = 0
    while (count < n − 1) do
        get next edge (v, w)
        if (component (v) ≠ component(w))
            add to T
            component(v) = component(w)
```

  The "which component?" tests can be efficiently implemented using the union-find data structure (Section 12.5 (page 385)) to yield an $O(m \lg m)$ algorithm.

- *Prim's algorithm* – Starts with an arbitrary vertex v and "grows" a tree from it, repeatedly finding the lowest-cost edge that links some new vertex into this tree. During execution, we label each vertex as either in the tree, in the *fringe* (meaning there exists an edge from a tree vertex), or *unseen* (meaning the vertex is still more than one edge away from the tree).

```
Prim(G)
    Select an arbitrary vertex to start
    While (there are fringe vertices)
        select minimum-weight edge between tree and fringe
        add the selected edge and vertex to the tree
        update the cost to all affected fringe vertices
```

  This creates a spanning tree for any connected graph, since no cycle can be introduced via edges between tree and fringe vertices. That it is in fact a tree of minimum weight can be proven by contradiction. With simple data structures, Prim's algorithm can be implemented in $O(n^2)$ time.

- *Boruvka's algorithm* – Rests on the observation that the lowest-weight edge incident on each vertex must be in the minimum spanning tree. The union of these edges will result in a spanning forest of at most $n/2$ trees. Now for each of these trees T, select the edge (x, y) of lowest weight such that $x \in T$ and

$y \notin T$. Each of these edges must again be in an MST, and the union again results in a spanning forest with at most half as many trees as before:

```
Boruvka(G)
    Initialize spanning forest F to n single-vertex trees
        While (F has more than one tree)
            for each T in F, find the smallest edge from T to G − T
            add all selected edges to F, thus merging pairs of trees
```

The number of trees are at least halved in each round, so we get the MST after at most $\lg n$ iterations, each of which takes linear time. This gives an $O(m \log n)$ algorithm without using any fancy data structures.

MST is only one of several spanning tree problems that arise in practice. The following questions will help you sort your way through them:

- *Are the weights of all edges of your graph identical?* – Every spanning tree on n points contains exactly $n-1$ edges. Thus, if your graph is unweighted, *any* spanning tree will be a minimum spanning tree. Either breadth-first or depth-first search can be used to find a rooted spanning tree in linear time. DFS trees tend to be long and thin, while BFS trees better reflect the distance structure of the graph, as discussed in Section 5 (page 145).

- *Should I use Prim's or Kruskal's algorithm?* – As implemented in Section 6.1 (page 192), Prim's algorithm runs in $O(n^2)$, while Kruskal's algorithm takes $O(m \log m)$ time. Thus Prim's algorithm is faster on dense graphs, while Kruskal's is faster on sparse graphs.

  That said, Prim's algorithm can be implemented in $O(m + n \lg n)$ time using more advanced data structures, and a Prim's implementation using pairing heaps would be the fastest practical choice for both sparse and dense graphs.

- *What if my input is points in the plane, instead of a graph?* – Geometric instances, comprising n points in d-dimensions, can be solved by constructing the complete distance graph in $O(n^2)$ and then finding the MST of this complete graph. However, for points in two dimensions, it is more efficient to solve the geometric version of the problem directly. To find the minimum spanning tree of n points, first construct the Delaunay triangulation of these points (see Sections 17.3 and 17.4). In two dimensions, this gives a graph with $O(n)$ edges that contains all the edges of the minimum spanning tree of the point set. Running Kruskal's algorithm on this sparse graph finishes the job in $O(n \lg n)$ time.

- *How do I find a spanning tree that avoids vertices of high degree?* – Another common goal of spanning tree problems is to minimize the maximum degree,

typically to minimize the fan out in an interconnection network. Unfortunately, finding a spanning tree of maximum degree 2 is NP-complete, since this is identical to the Hamiltonian path problem. However, efficient algorithms are known that construct spanning trees whose maximum degree is at most one more than required. This is likely to suffice in practice. See the references below.

Implementations: All the graph data structure implementations of Section 12.4 (page 381) *should* include implementations of Prim's and/or Kruskal's algorithms. This includes the Boost Graph Library [SLL02] (*http://www.boost.org/libs/graph/doc*) and LEDA (see Section 19.1.1 (page 658)) for C++. For some reason it does not seem to include the Java graph libraries oriented around social networks, but Prim and Kruskal are present in the *Data Structures Library in Java* (JDSL) (*http://www.jdsl.org/*).

Timing experiments on MST algorithms produce contradicting results, suggesting the stakes are really too low to matter. Pascal implementations of Prim's, Kruskal's, and the Cheriton-Tarjan algorithm are provided in [MS91], along with extensive empirical analysis proving that Prim's algorithm with the appropriate priority queue is fastest on most graphs. The programs in [MS91] are available by anonymous FTP from *cs.unm.edu* in directory */pub/moret_shapiro*. Kruskal's algorithm proved the fastest of four different MST algorithms in the Stanford GraphBase (see Section 19.1.8 (page 660)).

Combinatorica [PS03] provides Mathematica implementations of Kruskal's MST algorithm and quickly counting the number of spanning trees of a graph. See Section 19.1.9 (page 661).

My (biased) preference for C language implementations of all basic graph algorithms, including minimum spanning trees, is the library associated with this book. See Section 19.1.10 (page 661) for details.

Notes: The MST problem dates back to Boruvka's algorithm in 1926. Prim's [Pri57] and Kruskal's [Kru56] algorithms did not appear until the mid-1950's. Prim's algorithm was then rediscovered by Dijkstra [Dij59]. See [GH85] for more on the interesting history of MST algorithms. Wu and Chao [WC04b] have written a monograph on MSTs and related problems.

The fastest implementations of Prim's and Kruskal's algorithms use Fibonacci heaps [FT87]. However, pairing heaps have been proposed to realize the same bounds with less overhead. Experiments with pairing heaps are reported in [SV87].

A simple combination of Boruvka's algorithm with Prim's algorithm yields an $O(m \lg \lg n)$ algorithm. Run Borukva's algorithm for $\lg \lg n$ iterations, yielding a forest of at most $n/\lg n$ trees. Now create a graph G' with one vertex for each tree in this forest, with the weight of the edge between trees T_i and T_j set to the lightest edge (x, y), where $x \in T_i$ and $y \in T_j$. The MST of G' coupled with the edges selected by Boruvka's algorithm yields the MST of G. Prim's algorithm (implemented with Fibonacci heaps) will take $O(n + m)$ time on this $n/\lg n$ vertex, m edge graph.

The best theoretical bounds on finding MSTs tell a complicated story. Karger, Klein, and Tarjan [KKT95] give a linear-time randomized algorithm for MSTs, based again on Borukva's algorithm. Chazelle [Cha00] gave a deterministic $O(n\alpha(m,n))$ algorithm, where $\alpha(m,n)$ is the inverse Ackerman function. Pettie and Ramachandran [PR02] give an provably optimal algorithm whose exact running time is (paradoxically) unknown, but lies between $\Omega(n+m)$ and $O(n\alpha(m,n))$.

A *spanner* $S(G)$ of a given graph G is a subgraph that offers an effective compromise between two competing network objectives. To be precise, they have total weight close to the MST of G, while guaranteeing that the shortest path between vertices x and y in $S(G)$ approaches the shortest path in the full graph G. The monograph of Narasimhan and Smid [NS07] provides a complete, up-to-date survey on spanner networks.

The $O(n \log n)$ algorithm for Euclidean MSTs is due to Shamos, and discussed in computational geometry texts such as [dBvKOS00, PS85].

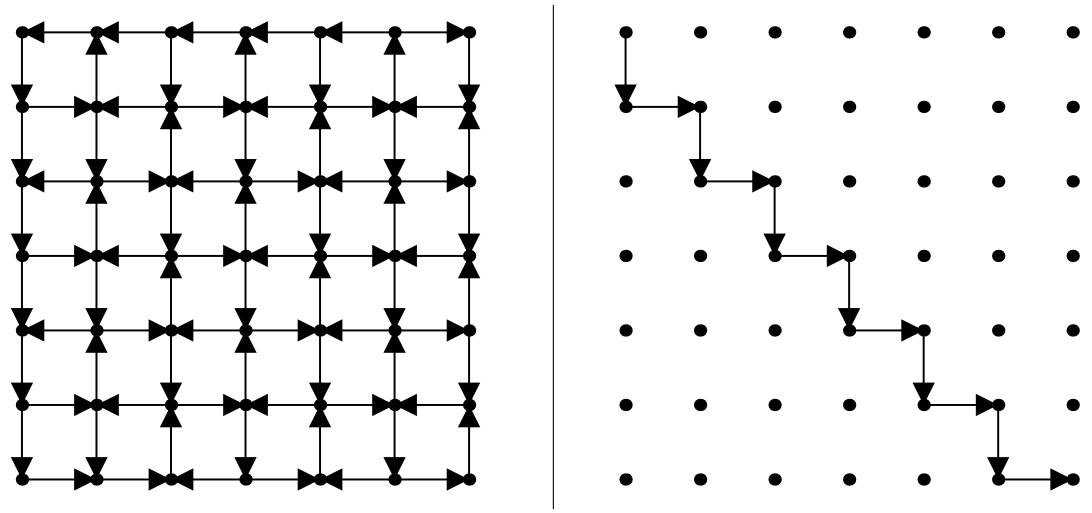
Fürer and Raghavachari [FR94] give an algorithm that constructs a spanning tree whose maximum degree is almost minimized—indeed is at most one more than the lowest-degree spanning tree. The situation is analogous to Vizing's theorem for edge coloring, which also gives an approximation algorithm to within additive factor one. A recent generalization [SL07] gives a polynomial-time algorithm for finding a spanning tree of maximum degree $\leq k+1$ whose cost is no more than that of the optimal minimum spanning tree of maximum degree $\leq k$.

Minimum spanning tree algorithms have an interpretation in terms of *matroids*, which are systems of subsets closed under inclusion. The maximum weighted independent set in matroids can be found using a greedy algorithm. The connection between greedy algorithms and matroids was established by Edmonds [Edm71]. Expositions on the theory of matroids include [Law76, PS98].

Dynamic graph algorithms seek to maintain an graph invariant (such as the MST) efficiently under edge insertion or deletion operations. Holm et al. [HdlT01] gives an efficient, deterministic algorithm to maintain MSTs (and several other invariants) in amortized polylogarithmic time per update.

Algorithms for generating spanning trees in order from minimum to maximum weight are presented in [Gab77]. The complete set of spanning trees of an unweighted graph can be generated in constant amortized time. See Ruskey [Rus03] for an overview of algorithms for generating, ranking, and unranking spanning trees.

Related Problems: Steiner tree (see page 555), traveling salesman (see page 533).



INPUT OUTPUT

15.4 Shortest Path

Input description: An edge-weighted graph G, with vertices s and t.

Problem description: Find the shortest path from s to t in G.

Discussion: The problem of finding shortest paths in a graph has several applications, some quite surprising:

- The most obvious applications arise in transportation or communications, such as finding the best route to drive between Chicago and Phoenix or figuring how to direct packets to a destination across a network.
- Consider the task of partitioning a digitized image into regions containing distinct objects—a problem known as *image segmentation*. Separating lines are needed to carve the space into regions, but what path should these lines take through the grid? We may want a line that relatively directly goes from x to y, but avoids cutting through object pixels as much as possible. This grid of pixels can be modeled as a graph, with the cost of an edge reflecting the color transitions between neighboring pixels. The shortest path from x to y in this weighted graph defines the best separating line.
- *Homophones* are words that sound alike, such as *to*, *two*, and *too*. Distinguishing between homophones is a major problem in speech recognition systems.

The key is to bring some notion of grammatical constraints into the interpretation. We map each string of phonemes (recognized sounds) into words they might possibly match. We construct a graph whose vertices correspond to these possible word interpretations, with edges between neighboring word-interpretations. If we set the weight of each edge to reflect the likelihood of transition, the shortest path across this graph defines the best interpretation of the sentence. See Section 6.4 (page 212) for a more detailed account of a similar application.

- Suppose we want to draw an informative visualization of a graph. The "center" of the graph should appear near the center of the page. A good definition of the graph center is the vertex that minimizes the maximum distance to any other vertex in the graph. Identifying this center point requires knowing the distance (i.e. , shortest path) between all pairs of vertices.

The primary algorithm for finding shortest paths is *Dijkstra's algorithm*, which efficiently computes the shortest path from a given starting vertex x to all $n-1$ other vertices. In each iteration, it identifies a new vertex v for which the shortest path from x to v is known. We maintain a set of vertices S to which we currently know the shortest path from v, and this set grows by one vertex in each iteration. In each iteration, we identify the edge (u, v) where $u, u' \in S$ and $v, v' \in V - S$ such that

$$dist(x, u) + weight(u, v) = \min_{(u', v') \in E} dist(x, u') + weight(u', v')$$

This edge (u, v) gets added to a *shortest path tree*, whose root is x and describes all the shortest paths from x.

An $O(n^2)$ implementation of Dijkstra's algorithm appears in in Section 6.3.1 (page 206). Theoretically faster times can be achieved using significantly more complicated data structures, as described below. If we just need to know the shortest path from x to y, terminate the algorithm as soon as y enters S.

Dijkstra's algorithm is the right choice for single-source shortest path on positively weighted graphs. However, special circumstances dictate different choices:

- *Is your graph weighted or unweighted?* – If your graph is unweighted, a simple breadth-first search starting from the source vertex will find the shortest path to all other vertices in linear time. Only when edges have different weights do you need more sophisticated algorithms. A breadth-first search is both simpler and faster than Dijkstra's algorithm.

- *Does your graph have negative cost weights?* – Dijkstra's algorithm assumes that all edges have positive cost. For graphs with edges of negative weight, you must use the more general, but less efficient, Bellman-Ford algorithm. Graphs with negative cost cycles are an even bigger problem. Observe that the shortest x to y path in such a graph is not defined because we can detour

from x to the negative cost cycle and repeatedly loop around it, making the total cost arbitrarily small.

Note that adding a fixed amount of weight to make each edge positive *does not* solve the problem. Dijkstra's algorithm will then favor paths using a fewer number of edges, even if those were not the shortest weighted paths in the original graph.

- *Is your input a set of geometric obstacles instead of a graph?* – Many applications seek the shortest path between two points in a geometric setting, such as an obstacle-filled room. The most straightforward solution is to convert your problem into a graph of distances to feed to Dijkstra's algorithm. Vertices will correspond the vertices on the boundaries of the obstacles, with edges defined only between pairs of vertices that "see" each other.

  Alternately, there are more efficient geometric algorithms that compute the shortest path directly from the arrangement of obstacles. See Section 17.14 (page 610) on motion planning and the Notes section for pointers to such geometric algorithms.

- *Is your graph acyclic—i.e. , a DAG?* – Shortest paths in directed acyclic graphs can be found in linear time. Perform a topological sort to order the vertices such that all edges go from left to right starting from source s. The distance from s to itself, $d(s, s)$, clearly equals 0. We now process the vertices from left to right. Observe that

$$d(s, j) = \min_{(x,i) \in E} d(s, i) + w(i, j)$$

  since we already know the shortest path $d(s, i)$ for all vertices to the left of j. Indeed, most dynamic programming problems can be formulated as shortest paths on specific DAGs. Note that the same algorithm (replacing min with max) also suffices to find the *longest path* in a DAG, which is useful in many applications like scheduling (see Section 14.9 (page 468)).

- *Do you need the shortest path between all pairs of points?* – If you are interested in the shortest path between all pairs of vertices, one solution is to run Dijkstra n times, once with each vertex as the source. The Floyd-Warshall algorithm is a slick $O(n^3)$ dynamic programming algorithm for all-pairs shortest path, which is faster and easier to program than Dijkstra. It works with negative cost edges but not cycles, and is presented with an implementation in Section 6.3.2 (page 210). Let M denote the distance matrix, where $M_{ij} = \infty$ if there is no edge (i, j):

$$D^0 = M$$
$$\text{for } k = 1 \text{ to } n \text{ do}$$
$$\quad \text{for } i = 1 \text{ to } n \text{ do}$$

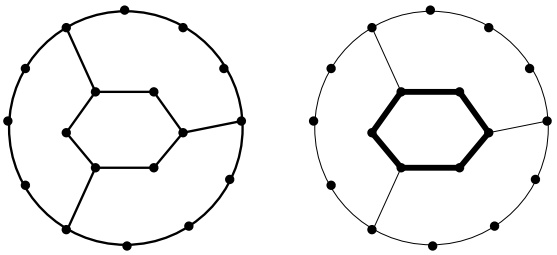


Figure 15.1: The girth, or shortest cycle, in a graph

---

for $j = 1$ to n do
$D_{ij}^k = \min(D_{ij}^{k-1}, D_{ik}^{k-1} + D_{kj}^{k-1})$
Return D^n

The key to understanding Floyd's algorithm is that D_{ij}^k denotes "the length of the shortest path from i to j that goes through vertices $1, \ldots, k$ as possible intermediate vertices." Note that $O(n^2)$ space suffices, since we only must keep D^k and D^{k-1} around at time k.

- *How do I find the shortest cycle in a graph?* – One application of all-pairs shortest path is to find the shortest cycle in a graph, called its *girth*. Floyd's algorithm can be used to compute d_{ii} for $1 \leq i \leq n$, which is the length of the shortest way to get from vertex i to i—in other words, the shortest cycle through i.

  This *might* be what you want. The shortest cycle through x is likely to go from x to y back to x, using the same edge twice. A *simple* cycle is one that visits no edge or vertex twice. To find the shortest simple cycle, the easiest approach is to compute the lengths of the shortest paths from i to all other vertices, and then explicitly check whether there is an acceptable edge from each vertex back to i.

  Finding the *longest* cycle in a graph includes Hamiltonian cycle as a special case (see Section 16.5), so it is NP-complete.

The all-pairs shortest path matrix can be used to compute several useful invariants related to the center of graph G. The *eccentricity* of vertex v in a graph is the shortest-path distance to the farthest vertex from v. From the eccentricity come other graph invariants. The *radius* of a graph is the smallest eccentricity of any vertex, while the *center* is the set of vertices whose eccentricity is the radius. The *diameter* of a graph is the maximum eccentricity of any vertex.

Implementations: The highest performance shortest path codes are due to Andrew Goldberg and his collaborators, at *http://www.avglab.com/andrew/soft.html*.

In particular, MLB is a C++ short path implementation for non-negative, integer-weighted edges. See [Gol01] for details of the algorithm and its implementation. Its running time is typically only 4-5 times that of a breadth-first search, and it is capable of handling graphs with millions of vertices. High-performance C implementations of both Dijkstra and Bellman-Ford are also available [CGR99].

All the C++ and Java graph libraries discussed in Section 12.4 (page 381) include at least an implementation of Dijkstra's algorithm. The C++ Boost Graph Library [SLL02] (*http://www.boost.org/libs/graph/doc*) has a particularly broad collection, including Bellman-Ford and Johnson's all-pairs shortest-path algorithm. LEDA (see Section 19.1.1 (page 658)) provides good implementations in C++ for all of the shortest-path algorithms we have discussed, including Dijkstra, Bellman-Ford, and Floyd's algorithms. JGraphT (*http://jgrapht.sourceforge.net/*) provides both Dijkstra and Bellman-Ford in Java. C language implementations of Dijkstra and Floyd's algorithms are provided in the library from this book. See Section 19.1.10 (page 661) for details.

Shortest-path algorithms was the subject of the 9th DIMACS Implementation Challenge, held in October 2006. Implementations of efficient algorithms for several aspects of finding shortest paths were discussed. The papers, instances, and implementations are available at *http://dimacs.rutgers.edu/Challenges/*.

Notes: Good expositions on Dijkstra's algorithm [Dij59], the Bellman-Ford algorithm [Bel58, FF62], and Floyd's all-pairs-shortest-path algorithm [Flo62] include [CLRS01]. Zwick [Zwi01] provides an up-to-date survey on shortest path algorithms. Geometric shortest-path algorithms are surveyed by Mitchell [PN04].

The fastest algorithm known for single-source shortest-path for positive edge weight graphs is Dijkstra's algorithm with Fibonacci heaps, running in $O(m + n \log n)$ time [FT87]. Experimental studies of shortest-path algorithms include [DF79, DGKK79]. However, these experiments were done before Fibonacci heaps were developed. See [CGR99] for a more recent study. Heuristics can be used to enhance the performance of Dijkstra's algorithm in practice. Holzer, et al. [HSWW05] provide a careful experimental study of how four such heuristics interact together.

Online services like Mapquest quickly find at least an approximate shortest path between two points in enormous road networks. This problem differs somewhat from the shortest-path problems here in that (1) preprocessing costs can be amortized over many point-to-point queries, (2) the backbone of high-speed, long-distance highways can reduce the path problem to identifying the best place to get on and off this backbone, and (3) approximate or heuristic solutions suffice in practice.

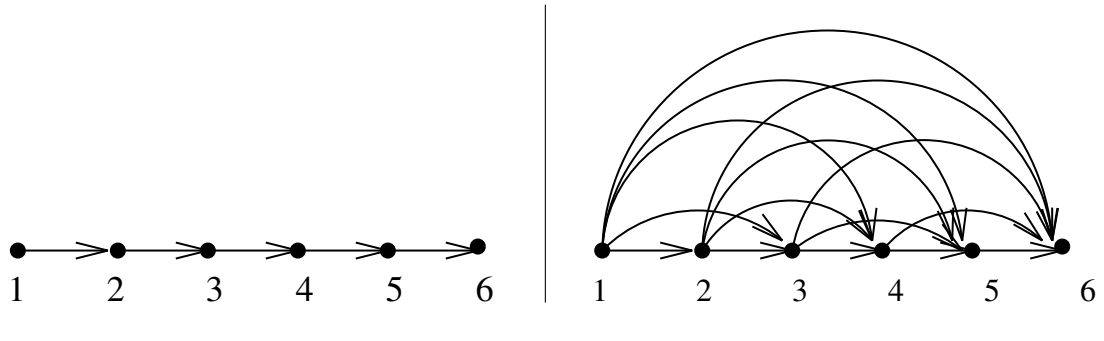
The A^*-algorithm performs a best-first search for the shortest path coupled with a lower-bound analysis to establish when the best path we have seen is indeed the shortest-path in the graph. Goldberg, Kaplan, and Werneck [GKW06, GKW07] describe an implementation of A^* capable of answering point-to-point queries in one millisecond on national-scale road networks after two hours of preprocessing.

Many applications demand multiple alternative short paths in addition to the optimal path. This motivates the problem of finding the k shortest paths. Variants exist depending upon whether the paths must be simple, or can contain cycles. Eppstein [Epp98] generates an implicit representation of these paths in $O(m+n \log n+k)$ time, from which each path

can be reconstructed in $O(n)$ time. Hershberger, et al. [HMS03] present a new algorithm and experimental results.

Fast algorithms for computing the girth are known for both general [IR78] and planar graphs [Dji00].

Related Problems: Network flow (see page 509), motion planning (see page 610).



15.5 Transitive Closure and Reduction

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

Problem description: For *transitive closure*, construct a graph $G' = (V, E')$ with edge $(i, j) \in E'$ iff there is a directed path from i to j in G. For *transitive reduction*, construct a small graph $G' = (V, E')$ with a directed path from i to j in G' iff there is a directed path from i to j in G.

Discussion: Transitive closure can be thought of as establishing a data structure that makes it possible to solve reachability questions (can I get to x from y?) efficiently. After constructing the transitive closure, all reachability queries can be answered in constant time by simply reporting the appropriate matrix entry.

Transitive closure is fundamental in propagating the consequences of modified attributes of a graph G. For example, consider the graph underlying any spreadsheet model whose vertices are cells and have an edge from cell i to cell j if the result of cell j depends on cell i. When the value of a given cell is modified, the values of all reachable cells must also be updated. The identity of these cells is revealed by the transitive closure of G. Many database problems reduce to computing transitive closures, for analogous reasons.

There are three basic algorithms for computing transitive closure:

- The simplest algorithm just performs a breadth-first or depth-first search from each vertex and keeps track of all vertices encountered. Doing n such traversals gives an $O(n(n+m))$ algorithm, which degenerates to cubic time if the graph is dense. This algorithm is easily implemented, runs well on sparse graphs, and is likely the right answer for your application.

- Warshall's algorithm constructs transitive closures in $O(n^3)$ using a simple, slick algorithm that is identical to Floyd's all-pairs shortest-path algorithm

of Section 15.4 (page 489). If we are interested only in the transitive closure, and not the length of the resulting paths, we can reduce storage by retaining only one bit for each matrix element. Thus, $D_{ij}^k = 1$ iff j is reachable from i using only vertices $1, \ldots, k$ as intermediates.

- Matrix multiplication can also be used to solve transitive closure. Let M^1 be the adjacency matrix of graph G. The non-zero matrix entries of $M^2 = M \times M$ identify all length-2 paths in G. Observe that $M^2[i,j] = \sum_x M[i,x] \cdot M[x,j]$, so path (i,x,j) contributes to $M^2[i,j]$. Thus, the union $\cup_i^n M^i$ yields the transitive closure T. Furthermore, this union can be computed using only $O(\lg n)$ matrix operations using the fast exponentiation algorithm in Section 13.9 (page 423).

  You might conceivably win for large n by using Strassen's fast matrix multiplication algorithm, although I for one wouldn't bother trying. Since transitive closure is provably as hard as matrix multiplication, there is little hope for a significantly faster algorithm.

The running time of all three of these procedures can be substantially improved on many graphs. Recall that a strongly connected component is a set of vertices for which all pairs are mutually reachable. For example, any cycle defines a strongly connected subgraph. All the vertices in any strongly connected component must reach exactly the same subset of G. Thus, we can reduce our problem finding the transitive closure on a graph of strongly connected components that should have considerably fewer edges and vertices than G. The strongly connected components of G can be computed in linear time (see Section 15.1 (page 477)).

Transitive reduction (also known as *minimum equivalent digraph*) is the inverse operation of transitive closure, namely reducing the number of edges while maintaining identical reachability properties. The transitive closure of G is identical to the transitive closure of the transitive reduction of G. The primary application of transitive reduction is space minimization, by eliminating redundant edges from G that do not effect reachability. Transitive reduction also arises in graph drawing, where it is important to eliminate as many unnecessary edges as possible to reduce the visual clutter.

Although the transitive closure of G is uniquely defined, a graph may have many different transitive reductions, including G itself. We want the smallest such reduction, but there are multiple formulations of the problem:

- A linear-time, quick-and-dirty transitive reduction algorithm identifies the strongly connected components of G, replaces each by a simple directed cycle, and adds these edges to those bridging the different components. Although this reduction is not provably minimal, it is likely to be pretty close on typical graphs.

  One catch with this heuristic is that it might add edges to the transitive reduction of G that are not in G. This may or may not be a problem depending on your application.

- If all edges of our transitive reduction must exist in G, we have to abandon hope of finding the minimum size reduction. To see why, consider a directed graph consisting of one strongly connected component so that every vertex can reach every other vertex. The smallest possible transitive reduction will be a simple directed cycle, consisting of exactly n edges. This is possible if and only if G is Hamiltonian, thus proving that finding the smallest subset of edges is NP-complete.

  A heuristic for finding such a transitive reduction is to consider each edge successively and delete it if its removal does not change the transitive reduction. Implementing this efficiently means minimizing the time spent on reachability tests. Observe that a directed edge (i, j) can be eliminated whenever there is another path from i to j avoiding this edge.

- The minimum size reduction where we are allowed arbitrary pairs of vertices as edges can be found in $O(n^3)$ time. See the references below for details. However, the quick-and-dirty heuristic above will likely suffice for most applications, being easier to program as well as more efficient.

Implementations: The Boost implementation of transitive closure appears particularly well engineered, and relies on algorithms from [Nuu95]. LEDA (see Section 19.1.1 (page 658)) provides implementations of both transitive closure and reduction in C++ [MN99].

None of our usual Java libraries appear to contain implementations of either transitive closure or reduction. However, *Graphlib* contains a Java `Transitivity` library with both of them. See *http://www-verimag.imag.fr/∼cotton/* for details.

Combinatorica [PS03] provides Mathematica implementations of transitive closure and reduction, as well as the display of partial orders requiring transitive reduction. See Section 19.1.9 (page 661).

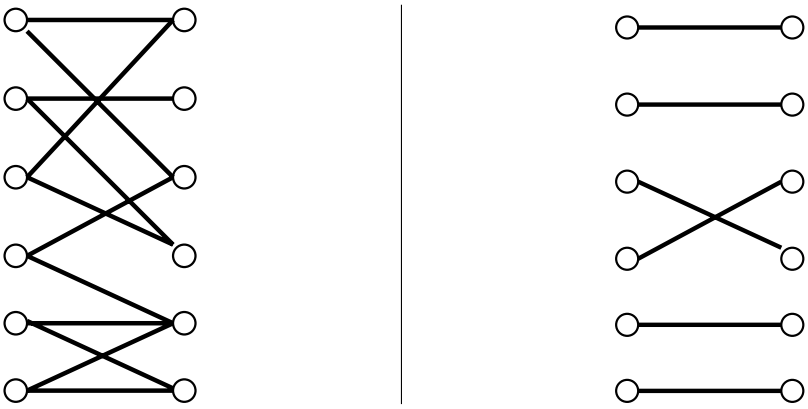
Notes: Van Leeuwen [vL90a] provides an excellent survey on transitive closure and reduction. The equivalence between matrix multiplication and transitive closure was proven by Fischer and Meyer [FM71], with expositions including [AHU74].

There is a surprising amount of recent activity on transitive closure, much of it captured by Nuutila [Nuu95]. Penner and Prasanna [PP06] improved the performance of Warshall's algorithm [War62] by roughly a factor of two through a cache-friendly implementation.

The equivalence between transitive closure and reduction, as well as the $O(n^3)$ reduction algorithm, was established in [AGU72]. Empirical studies of transitive closure algorithms include [Nuu95, PP06, SD75].

Estimating the size of the transitive closure is important in database query optimization. A linear-time algorithm for estimating the size of the closure is given by Cohen [Coh94].

Related Problems: Connected components (see page 477), shortest path (see page 489).



INPUT OUTPUT

15.6 Matching

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

Problem description: Find the largest set of edges E' from E such that each vertex in V is incident to at most one edge of E'.

Discussion: Suppose we manage a group of workers, each of whom is capable of performing a subset of the tasks needed to complete a job. Construct a graph with vertices representing both the set of workers and the set of tasks. Edges link workers to the tasks they can perform. We must assign each task to a different worker so that no worker is overloaded. The desired assignment is the largest possible set of edges where no employee or job is repeated—i.e. , a matching.

Matching is a very powerful piece of algorithmic magic, so powerful that it is surprising that optimal matchings can be found efficiently. Applications arise often once you know to look for them.

Marrying off a set of boys to a set of girls such that each couple is happy is another bipartite matching problem, on a graph with an edge between any compatible boy and girl. For a synthetic biology application [MPC+06], I need to shuffle the characters in a string S to maximize the number of characters that move. For example, *aaabc* can be shuffled to *bcaaa* so that only one character stays fixed. This is yet another bipartite matching problem, where the boys represent the multiset of alphabet symbols and the girls are the positions in the string (1 to $|S|$). Edges link symbols to all the string positions that originally contained a different symbol.

This basic matching framework can be enhanced in several ways, while remaining essentially the same *assignment* problem:

- *Is your graph bipartite?* – Most matching problems involve bipartite graphs, as in the classic assignment problem of jobs to workers. This is fortunate because faster and simpler algorithms exist for bipartite matching.

- *What if certain employees can be given multiple jobs?* – Natural generalizations of the assignment problem include assigning certain employees more than one task to do, or (equivalently) seeking multiple workers for a given job. Here, we do not seek a matching so much as a covering with small "stars." Such desires can be modeled by replicating an employee vertex by as many times as we want her to be matched. Indeed, we employed this trick in the string permutation example above.

- *Is your graph weighted or unweighted?* – Many matching applications are based on unweighted graphs. Perhaps we seek to maximize the total number of tasks performed, where one task is as good as another. Here we seek a maximum *cardinality* matching—ideally a *perfect* matching where every vertex is matched to another in the matching.

  For other applications, however, we need to augment each edge with a weight, perhaps reflecting the suitability of an employee for a given task. The problem now becomes constructing a maximum *weight* matching—i.e. , the set of independent edges of maximum total cost.

Efficient algorithms for constructing matchings work by constructing *augmenting paths* in graphs. Given a (partial) matching M in a graph G, an augmenting path is a path of edges P that alternate (out-of-M, in-M, ..., out-of-M). We can enlarge the matching by one edge given such an augmenting path, replacing the even-numbered edges of P from M with the odd-numbered edges of P. Berge's theorem states that a matching is maximum if and only if it does not contain any augmenting path. Therefore, we can construct maximum-cardinality matchings by searching for augmenting paths and stopping when none exist.

General graphs prove trickier because it is possible to have augmenting paths that are odd-length cycles (i.e. , the first and last vertices are the same). Such cycles (or blossoms) are impossible in bipartite graphs, which by definition do not contain odd-length cycles.

The standard algorithms for bipartite matching are based on network flow, using a simple transformation to convert a bipartite graph into an equivalent flow graph. Indeed, an implementation of this is given in Section 6.5 (page 217).

Be warned that different approaches are needed to solve weighted matching problems, most notably the matrix-oriented "Hungarian algorithm."

Implementations: High-performance codes for both weighted and unweighted bipartite matching have been developed by Andrew Goldberg and his collaborators. `CSA` is a weighted bipartite matching code in C based on cost-scaling network flow, developed by Goldberg and Kennedy [GK95]. `BIM` is a faster unweighted bipartite matching code based on augmenting path methods, developed

by Cherkassky, et al. [CGM+98]. Both are available for noncommercial use from *http://www.avglab.com/andrew/soft.html*.

The First DIMACS Implementation Challenge [JM93] focused on network flows and matching. Several instance generators and implementations for maximum weight and maximum cardinality matching were collected, and can be obtained by anonymous FTP from *dimacs.rutgers.edu* in the directory *pub/netflow/matching*. These include

- A maximum-cardinality matching solver in Fortran 77 by R. Bruce Mattingly and Nathan P. Ritchey.
- A maximum-cardinality matching solver in C by Edward Rothberg, that implements Gabow's $O(n^3)$ algorithm.
- A maximum-weighted matching solver in C by Edward Rothberg. This is slower but more general than his unweighted solver just described.

GOBLIN (*http://www.math.uni-augsburg.de/~fremuth/goblin.html*) is an extensive C++ library dealing with all of the standard graph optimization problems, including weighted bipartite matching. LEDA (see Section 19.1.1 (page 658)) provides efficient implementations in C++ for both maximum-cardinality and maximum-weighted matching, on both bipartite and general graphs.

Blossum IV [CR99] is an efficient code in C for minimum-weight perfect matching available at *http://www2.isye.gatech.edu/~wcook/software.html*. An $O(mn\alpha(m,n))$ implementation of maximum-cardinality matching in general graphs (*http://www.cs.arizona.edu/~kece/Research/software.html*) is due to Kececioglu and Pecqueur [KP98].

The Stanford GraphBase (see Section 19.1.8 (page 660)) contains an implementation of the Hungarian algorithm for bipartite matching. To provide readily visualized weighted bipartite graphs, Knuth uses a digitized version of the Mona Lisa and seeks row/column disjoint pixels of maximum brightness. Matching is also used to construct clever, resampled "domino portraits".

Notes: Lovász and Plummer [LP86] is the definitive reference on matching theory and algorithms. Survey articles on matching algorithms include [Gal86]. Good expositions on network flow algorithms for bipartite matching include [CLRS01, Eve79a, Man89], and those on the Hungarian method include [Law76, PS98]. The best algorithm for maximum bipartite matching, due to Hopcroft and Karp [HK73], repeatedly finds the shortest augmenting paths instead of using network flow, and runs in $O(\sqrt{n}m)$. The Hungarian algorithm runs in $O(n(m + n \log n))$ time.

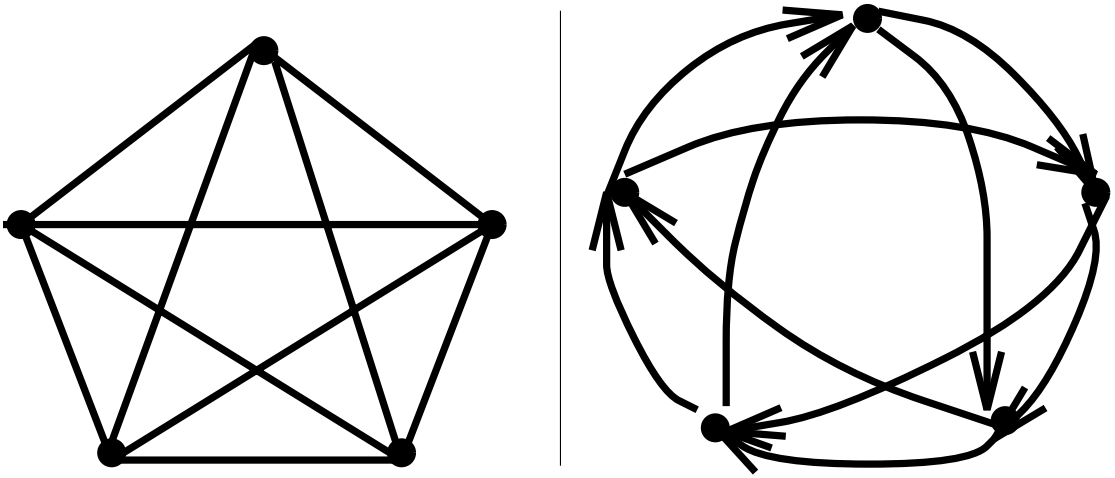
Edmond's algorithm [Edm65] for maximum-cardinality matching is of great historical interest for provoking questions on what problems can be solved in polynomial time. Expositions on Edmond's algorithm include [Law76, PS98, Tar83]. Gabow's [Gab76] implementation of Edmond's algorithm runs in $O(n^3)$ time. The best algorithm known for general matching runs in $O(\sqrt{n}m)$ [MV80].

Consider a matching of boys to girls containing edges (B_1, G_1) and (B_2, G_2), where B_1 and G_2 in fact prefer each other to their own spouses. In real life, these two would

run off with each other, breaking the marriages. A marriage without any such couples is said to be *stable*. The theory of stable matching is thoroughly treated in [GI89]. It is a surprising fact that no matter how the boys and girls rate each other, there is always at least one stable marriage. Further, such a marriage can be found in $O(n^2)$ time [GS62]. An important application of stable marriage occurs in the annual matching of medical residents to hospitals.

The maximum matching is equal in size to the minimum vertex cover in bipartite graphs. This implies that both the minimum vertex cover problem and maximum independent set problems can be solved in polynomial time on bipartite graphs.

Related Problems: Eulerian cycle (see page 502), network flow (see page 509).



INPUT OUTPUT

15.7 Eulerian Cycle/Chinese Postman

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

Problem description: Find the shortest tour visiting each edge of G at least once.

Discussion: Suppose you are given the map of a city and charged with designing the routes for garbage trucks, snow plows, or postmen. In each of these applications, every road in the city must be completely traversed at least once in order to ensure that all deliveries or pickups are made. For efficiency, you seek to minimize total drive time, or (equivalently) the total distance or number of edges traversed.

Alternately, consider a human-factors validation of telephone menu systems. Each "Press 4 for more information" option is properly interpreted as an edge between two vertices in a graph. Our tester seeks the most efficient way to walk over this graph and visit every link in the system at least once.

Such applications are variants of the *Eulerian cycle* problem, best characterized by the puzzle that asks children to draw a given figure completely without (1) without repeating any edges, or (2) lifting their pencil off the paper. They seek a path or cycle through a graph that visits each edge exactly once.

Well-known conditions exist for determining whether a graph contains an Eulerian cycle or path:

- An *undirected* graph contains an Eulerian *cycle* iff (1) it is connected, and (2) each vertex is of even degree.

- An *undirected* graph contains an Eulerian *path* iff (1) it is connected, and (2) all but two vertices are of even degree. These two vertices will be the start and end points of any path.

- A *directed* graph contains an Eulerian *cycle* iff (1) it is strongly-connected, and (2) each vertex has the same in-degree as out-degree.

- Finally, a *directed* graph contains an Eulerian *path* from x to y iff (1) it is connected, and (2) all other vertices have the same in-degree as out-degree, with x and y being vertices with in-degree one less and one more than their out-degrees, respectively.

This characterization of Eulerian graphs makes it is easy to test whether such a cycle exists: verify that the graph is connected using DFS or BFS, and then count the number of odd-degree vertices. Explicitly constructing the cycle also takes linear time. Use DFS to find an arbitrary cycle in the graph. Delete this cycle and repeat until the entire set of edges has been partitioned into a set of edge-disjoint cycles. Since deleting a cycle reduces each vertex degree by an even number, the remaining graph will continue to satisfy the same Eulerian degree-bound conditions. These cycles will have common vertices (since the graph is connected), and so can be spliced together in a "figure eight" at a shared vertex. By so splicing all the extracted cycles together, we construct a single circuit containing all of the edges.

An Eulerian cycle, if one exists, solves the motivating snowplow problem, since any tour that visits each edge only once must have minimum length. However, it is unlikely that your road network happens to satisfy the Eulerian degree conditions. Instead, we need to solve the more general *Chinese postman problem*, which minimizes the length of a cycle that traverses every edge at least once. This minimum cycle will never visit any edge more than twice, so good tours exist for any road network.

The optimal postman tour can be constructed by adding the appropriate edges to the graph G to make it Eulerian. Specifically, we find the shortest path between each pair of odd-degree vertices in G. Adding a path between two odd-degree vertices in G turns both of them to even-degree, moving G closer to becoming an Eulerian graph. Finding the best set of shortest paths to add to G reduces to identifying a minimum-weight perfect matching in a special graph G'.

For undirected graphs, the vertices of G' correspond the odd-degree vertices of G, with the weight of edge (i, j) defined to be the length of the shortest path from i to j in G. For directed graphs, the vertices of G' correspond to the degree-imbalanced vertices from G, with the bonus that all edges in G' go from out-degree deficient vertices to in-degree deficient ones. Thus, bipartite matching algorithms suffice when G is directed. Once the graph is Eulerian, the actual cycle can be extracted in linear time using the procedure just described.

Implementations: Several graph libraries provide implementations of Eulerian cycles, but Chinese postman implementations are rarer. We recommend the imple-

mentation of directed Chinese postman by Thimbleby [Thi03]. This Java implementation is available at *http://www.cs.swan.ac.uk/~csharold/cpp/index.html*.

GOBLIN (*http://www.math.uni-augsburg.de/~fremuth/goblin.html*) is an extensive C++ library dealing with all of the standard graph optimization problems, including Chinese postman for both directed and undirected graphs. LEDA (see Section 19.1.1 (page 658)) provides all the tools for an efficient implementation: Eulerian cycles, matching, and shortest-paths bipartite and general graphs.

Combinatorica [PS03] provides Mathematica implementations of Eulerian cycles and de Bruijn sequences. See Section 19.1.9 (page 661).

Notes: The history of graph theory began in 1736, when Euler [Eul36] first solved the seven bridges of Königsberg problem. Königsberg (now Kaliningrad) is a city on the banks of the Pregel river. In Euler's day there were seven bridges linking the banks and two islands, which can be modeled as a multigraph with seven edges and four vertices. Euler sought a way to walk over each of the bridges exactly once and return home—i.e. , an Eulerian cycle. Euler proved that such a tour is impossible, since all four of the vertices had odd degrees. The bridges were destroyed in World War II. See [BLW76] for a translation of Euler's original paper and a history of the problem.

Expositions on linear-time algorithms for constructing Eulerian cycles [Ebe88] include [Eve79a, Man89]. Fleury's algorithm [Luc91] is a direct and elegant approach to constructing Eulerian cycles. Start walking from any vertex, and erase any edge that has been traversed. The only criterion in picking the next edge is that we avoid using a bridge (an edge whose deletion disconnects the graph) until no other alternative remains.

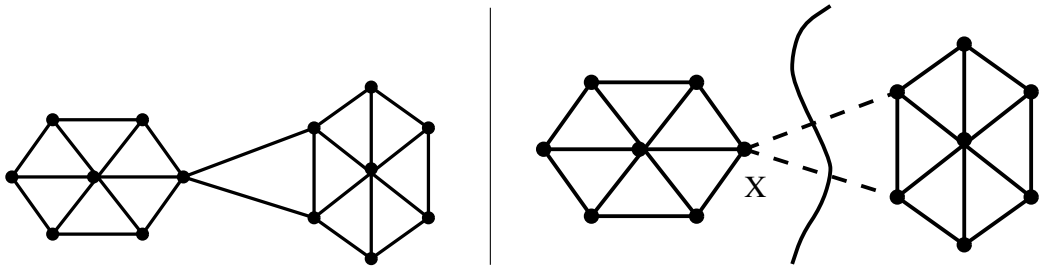
The Euler's tour technique is an important paradigm in parallel graph algorithms. Many parallel graph algorithms start by finding a spanning tree and then rooting the tree, where the rooting is done using the Euler tour technique. See parallel algorithms texts (e.g., [JÓ2]) for an exposition, and [CB04] for recent experience in practice. Efficient algorithms exist to count the number of Eulerian cycles in a graph [HP73].

The problem of finding the shortest tour traversing all edges in a graph was introduced by Kwan [Kwa62], hence the name *Chinese* postman. The bipartite matching algorithm for solving Chinese postman is due to Edmonds and Johnson [EJ73]. This algorithm works for both directed and undirected graphs, although the problem is NP-complete for mixed graphs [Pap76a]. Mixed graphs contain both directed and undirected edges. Expositions of the Chinese postman algorithm include [Law76].

A *de Bruijn* sequence S of span n on an alphabet Σ of size α is a circular string of length α^n containing all strings of length n as substrings of S, each exactly once. For example, for $n = 3$ and $\Sigma = \{0, 1\}$, the circular string 00011101 contains the following substrings in order: 000, 001, 011, 111, 110, 101, 010, 100. De Bruijn sequences can be thought of as "safe cracker" sequences, describing the shortest sequence of dial turns with α positions sufficient to try out all combinations of length n.

De Bruijn sequences can be constructed by building a directed graph whose vertices are all α^{n-1} strings of length $n-1$, where there is an edge (u, v) iff $u = s_1 s_2 \ldots s_{n-1}$ and $v = s_2 \ldots s_{n-1} s_n$. Any Eulerian cycle on this graph describes a de Bruijn sequence. Expositions on de Bruijn sequences and their construction include [Eve79a, PS03].

Related Problems: Matching (see page 498), Hamiltonian cycle (see page 538).



INPUT OUTPUT

15.8 Edge and Vertex Connectivity

Input description: A graph G. Optionally, a pair of vertices s and t.

Problem description: What is the smallest subset of vertices (or edges) whose deletion will disconnect G? Or which will separate s from t?

Discussion: Graph connectivity often arises in problems related to network reliability. In the context of telephone networks, the vertex connectivity is the smallest number of switching stations that a terrorist must bomb in order to separate the network—i.e. , prevent two unbombed stations from talking to each other. The edge connectivity is the smallest number of wires that need to be cut to accomplish the same objective. One well-placed bomb or snipping the right pair of cables suffices to disconnect the above network.

The edge (vertex) connectivity of a graph G is the smallest number of edge (vertex) deletions sufficient to disconnect G. There is a close relationship between the two quantities. The vertex connectivity is always less than or equal to the edge connectivity, since deleting one vertex from each edge in a cut set succeeds in disconnecting the graph. But smaller vertex subsets may be possible. The minimum vertex degree is an upper bound for both edge and vertex connectivity, since deleting all its neighbors (or cutting the edges to all its neighbors) disconnects the graph into one big and one single-vertex component.

Several connectivity problems prove to be of interest:

- *Is the graph already disconnected?* – The simplest connectivity problem is testing whether the graph is in fact connected. A simple depth-first or breadth-first search suffices to identify all connected components in linear time, as discussed in Section 15.1 (page 477). For directed graphs, the issue is whether the graph is *strongly connected*, meaning there is a directed path between any pair of vertices. In a *weakly connected* graph, there may exist paths to nodes from which there is no way to return.

- *Is there one weak link in my graph?* – We say that G is *biconnected* if no single vertex deletion is sufficient to disconnect G. Any vertex that is such a weak point is called an *articulation vertex*. A *bridge* is the analogous concept for edges, meaning a single edge whose deletion disconnects the graph.

  The simplest algorithms for identifying articulation vertices (or bridges) try deleting vertices (or edges) one by one, and then use DFS or BFS to test whether the resulting graph is still connected. More sophisticated linear-time algorithms exist for both problems, based on depth-first search. Indeed, a full implementation is given in Section 5.9.2 (page 173).

- *What if I want to split the graph into equal-sized pieces?* – What is often sought is a small cut set that breaks the graph into roughly equal-sized pieces. For example, suppose we want to split a big computer program into two maintainable units. We can construct a graph whose the vertices represent subroutines. Edges can be added between any two subroutines that interact, namely where one calls the other. We now seek to partition the subroutines into roughly equal-sized sets so that few pairs of interacting routines span the divide.

  This is the *graph partition* problem, further discussed in Section 16.6 (page 541). Although the problem is NP-complete, reasonable heuristics exist to solve it.

- *Are arbitrary cuts OK, or must I separate a given pair of vertices?* – There are two flavors of the general connectivity problem. One asks for the smallest cut-set for the entire graph, the other for the smallest set to separate s from t. Any algorithm for $(s-t)$ connectivity can be used with each of the $n(n-1)/2$ possible pairs of vertices to give an algorithm for general connectivity. Less obviously, $n-1$ runs suffice for testing edge connectivity, since we know that vertex v_1 must end up in a different component from at least one of the other $n-1$ vertices after deleting any cut set.

Edge and vertex connectivity can both be found using network-flow techniques. Network flow, discussed in Section 15.9 (page 509), interprets a weighted graph as a network of pipes where each edge has a maximum capacity and we seek to maximize the flow between two given vertices of the graph. The maximum flow between v_i, v_j in G is exactly the weight of the smallest set of edges to disconnect v_i from v_j. Thus the edge connectivity can be found by minimizing the flow between v_i and each of the $n-1$ other vertices in an unweighted graph G. Why? After deleting the minimum-edge cut set, v_i will be separated from some other vertex.

Vertex connectivity is characterized by *Menger's theorem*, which states that a graph is k-connected if and only if every pair of vertices is joined by at least k vertex-disjoint paths. Network flow can again be used to perform this calculation, since a flow of k between a pair of vertices implies k *edge*-disjoint paths. To exploit Menger's theorem, we construct a graph G' such that any set of edge-disjoint paths

in G' corresponds to vertex-disjoint paths in G. This is done by replacing each vertex v_i of G with two vertices $v_{i,1}$ and $v_{i,2}$, such that edge $(v_{i,1}, v_{i,2}) \in G'$ for all $v_i \in G$, and by replacing every edge $(v_i, x) \in G$ by edges $(v_{i,j}, x_k)$, $j \neq k \in \{0, 1\}$ in G'. Thus two edge-disjoint paths in G' correspond to each vertex-disjoint path in G.

Implementations: `MINCUTLIB` is a collection of high-performance codes for several different cut algorithms, including both flow and contraction-based methods. They were implemented by Chekuri, et al. as part of a substantial experimental study [CGK+97]. The codes are available for noncommercial use at *http://www.avglab.com/andrew/soft.html*. Also included is the full version of [CGK+97]—an excellent presentation of these algorithms and the heuristics needed to make them run fast.

Most of the graph data structure libraries of Section 15.1 (page 477) include routines for connectivity and biconnectivity testing. The C++ Boost Graph Library [SLL02] (*http://www.boost.org/libs/graph/doc*) is distinguished by also including an implementation of edge connectivity testing.

GOBLIN (*http://www.math.uni-augsburg.de/~fremuth/goblin.html*) is an extensive C++ library dealing with all of the standard graph optimization problems, including both edge and vertex connectivity.

LEDA (see Section 19.1.1 (page 658)) contains extensive support for both low-level connectivity testing (both biconnected and triconnected components) and edge connectivity/minimum-cut in C++.

Combinatorica [PS03] provides Mathematica implementations of edge and vertex connectivity, as well as connected, biconnected, and strongly connected components with bridges and articulation vertices. See Section 19.1.9 (page 661).

Notes: Good expositions on the network-flow approach to edge and vertex connectivity include [Eve79a, PS03]. The correctness of these algorithms is based on Menger's theorem [Men27] that connectivity is determined by the number of edge and vertex disjoint paths separating a pair of vertices. The maximum-flow, minimum-cut theorem is due to Ford and Fulkerson [FF62].

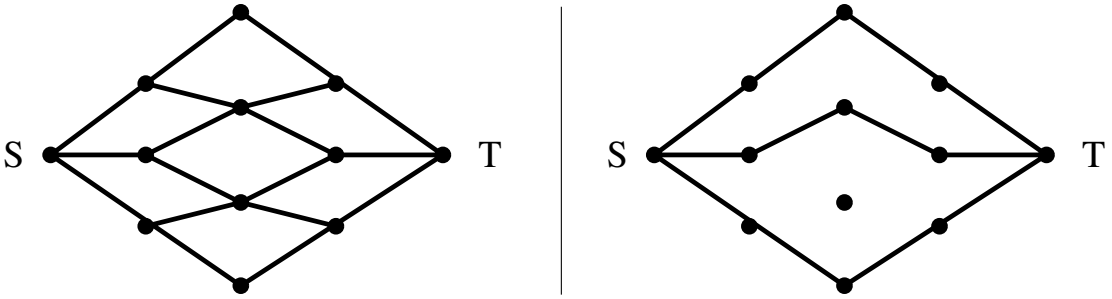
The theoretically fastest algorithms for minimum-cut/edge connectivity are based on graph contraction, not network flows. Contracting an edge (x, y) in a graph G merges the two incident vertices into one, removing self-loops but leaving multiedges. Any sequence of such contractions can raise (but not lower) the minimum cut in G, and leaves the cut unchanged if no edge of the cut is contracted. Karger gave a beautiful randomized algorithm for minimum cut, observing that the minimum cut is left unchanged with nontrivial probability over the course of any random series of deletions. The fastest version of Karger's algorithm runs in $(m \lg^3 n)$ expected time [Kar00]. See Motwani and Raghavan [MR95] for an excellent treatment of randomized algorithms, including a presentation of Karger's algorithm.

Nagamouchi and Ibaraki [NI92] give a deterministic contraction-based algorithm to find the minimum cut in $O(n(m + n \log n))$. In each round, this algorithm identifies and contracts an edge that is provably not in the minimum cut. See [CGK+97, NOI94] for experimental comparisons of algorithms for finding minimum cuts.

Minimum-cut methods have found many applications in computer vision, including image segmentation. Boykov and Kolmogorov [BK04] report on an experimental evaluation of minimum-cut algorithms in this context.

A nonflow-based algorithm for edge k-connectivity in $O(kn^2)$ is due to Matula [Mat87]. Faster k-connectivity algorithms are known for certain small values of k. All three-connected components of a graph can be generated in linear time [HT73a], while $O(n^2)$ suffices to test 4-connectivity [KR91].

Related Problems: Connected components (see page 477), network flow (see page 509), graph partition (see page 541).



INPUT OUTPUT

15.9 Network Flow

Input description: A directed graph G, where each edge $e = (i, j)$ has a capacity c_e. A source node s and sink node t.

Problem description: What is the maximum flow you can route from s to t while respecting the capacity constraint of each edge?

Discussion: Applications of network flow go far beyond plumbing. Finding the most cost-effective way to ship goods between a set of factories and a set of stores defines a network-flow problem, as do many resource-allocation problems in communications networks.

The real power of network flow is (1) that a surprising variety of linear programming problems arising in practice can be modeled as network-flow problems, and (2) that network-flow algorithms can solve these problems much faster than general-purpose linear programming methods. Several graph problems we have discussed in this book can be solved using network flow, including bipartite matching, shortest path, and edge/vertex connectivity.

The key to exploiting this power is recognizing that your problem can be modeled as network flow. This requires experience and study. My recommendation is that you first construct a linear programming model for your problem and then compare it with linear programs for the two primary classes of network flow problems: *maximum flow* and *minimum-cost flow*:

- *Maximum Flow* – Here we seek the heaviest possible flow from s to t, given the edge capacity constraints of G. Let x_{ij} be a variable accounting for the flow from vertex i through directed edge (i, j). The flow through this edge is constrained by its capacity c_{ij}, so

$$0 \leq x_{ij} \leq c_{ij} \text{ for } 1 \leq i, j \leq n$$

Furthermore, an equal flow comes in as goes out at each nonsource or sink vertex, so

$$\sum_{j=1}^{n} x_{ij} - \sum_{j=1}^{n} x_{ji} = 0 \text{ for } 1 \leq i \leq n$$

We seek the assignment that maximizes the flow into sink t, namely $\sum_{i=1}^{n} x_{it}$

- *Minimum Cost Flow* – Here we have an extra parameter for each edge (i, j), namely the cost (d_{ij}) of sending one unit of flow from i to j. We also have a targeted amount of flow f we want to send from s to t at minimum total cost. Hence, we seek the assignment that minimizes

  $$\sum_{j=1}^{n} d_{ij} \cdot x_{ij}$$

  subject to the edge and vertex capacity constraints of maximum flow, plus the additional restriction that $\sum_{i=1}^{n} x_{it} = f$.

Special considerations include:

- *What if I have multiple sources and/or sinks?* – No problem. We can handle this by modifying the network to create a vertex to serve as a super-source that feeds all the sources, and a super-sink that drains all the sinks.

- *What if all arc capacities are identical, either 0 or 1?* – Faster algorithms exist for 0-1 network flows. See the Notes section for details.

- *What if all my edge costs are identical?* – Use the simpler and faster algorithms for solving maximum flow as opposed to minimum-cost flow. Max-flow without edge costs arises in many applications, including edge/vertex connectivity and bipartite matching.

- *What if I have multiple types of material moving through the network?* – In a telecommunications network, every message has a given source and destination. Each destination needs to receive *exactly* those calls sent to it, not an equal amount of communication from arbitrary places. This can be modeled as a *multicommodity flow* problem, where each call defines a different commodity and we seek to satisfy all demands without exceeding the total capacity of any edge.

  Linear programming will suffice for multicommodity flow if fractional flows are permitted. Unfortunately, integral multicommodity flow is NP-complete, even for only two commodities.

Network flow algorithms can be complicated, and significant engineering is required to optimize performance. Thus, we strongly recommend that you use an existing code rather than implement your own. Excellent codes are available and described below. The two primary classes of algorithms are:

- *Augmenting path methods* – These algorithms repeatedly find a path of positive capacity from source to sink and add it to the flow. It can be shown that the flow through a network is optimal if and only if it contains no augmenting path. Since each augmentation adds something to the flow, we eventually find the maximum. The difference between network-flow algorithms is in *how* they select the augmenting path. If we are not careful, each augmenting path will add but a little to the total flow, and so the algorithm might take a long time to converge.
- *Preflow-push methods* – These algorithms push flows from one vertex to another, initially ignoring the constraint that in-flow must equal out-flow at each vertex. Preflow-push methods prove faster than augmenting-path methods, essentially because multiple paths can be augmented simultaneously. These algorithms are the method of choice and are implemented in the best codes described in the next section.

Implementations: High-performance codes for both maximum flow and minimum cost flow were developed by Andrew Goldberg and his collaborators. The codes `HIPR` and `PRF` [CG94] are provided for maximum flow, with the proviso that `HIPR` is recommended in most cases. For minimum-cost flow, the code of choice is `CS` [Gol97]. Both are written in C and available for noncommercial use from *http://www.avglab.com/andrew/soft.html*.

GOBLIN (*http://www.math.uni-augsburg.de/~fremuth/goblin.html*) is an extensive C++ library dealing with all of the standard graph optimization problems, including several different algorithms for both maximum flow and minimum-cost flow. The same holds true for LEDA, if a commercial-strength solution is needed. See Section 19.1.1 (page 658).

The First DIMACS Implementation Challenge on Network Flows and Matching [JM93] collected several implementations and generators for network flow, which can be obtained by anonymous FTP from *dimacs.rutgers.edu* in the directory *pub/netflow/maxflow*. These include: (1) a preflow-push network flow implementation in C by Edward Rothberg, and (2) an implementation of eleven network flow variants in C, including the older Dinic and Karzanov algorithms by Richard Anderson and Joao Setubal.

Notes: The primary book on network flows and its applications is [AMO93]. Good expositions on network flow algorithms old and new include [CCPS98, CLRS01, PS98]. Expositions on the hardness of multicommodity flow [Ita78] include [Eve79a].

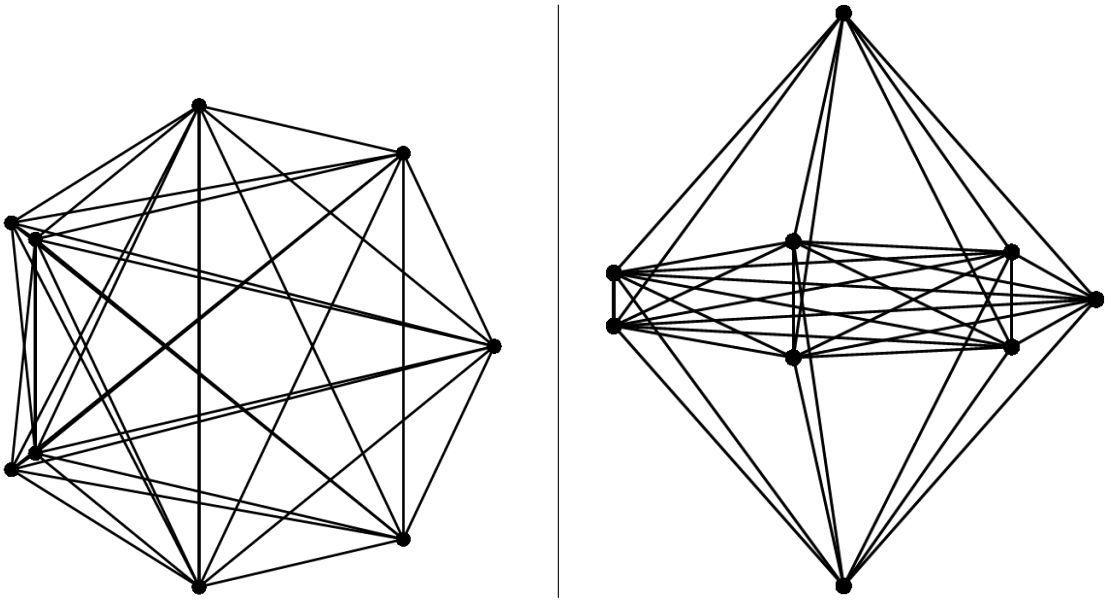
There is a very close connection to maximum flow and edge connectivity in graphs. The fundamental maximum-flow, minimum-cut theorem is due to Ford and Fulkerson

[FF62]. See Section 15.8 (page 505) for simpler and more efficient algorithms to compute the minimum cut in a graph.

Conventional wisdom holds that network flow should be computable in $O(nm)$ time, and there has been steady progress in lowering the time complexity. See [AMO93] for a history of algorithms for the problem. The fastest known general network flow algorithm runs in $O(nm \lg(n^2/m))$ time [GT88]. Empirical studies of minimum-cost flow algorithms include [GKK74, Gol97].

Information flows through a network can be modeled as multicommodity flows through a network, with the observation that replicating and manipulating information at internal nodes can eliminate the need for distinct sources to sink paths when multiple sinks are interested in the same information. The field of *network coding* [YLCZ05] uses such ideas to achieve information flows through networks at the theoretical limits of the max-flow, min-cut theorem.

Related Problems: Linear programming (see page 411), matching (see page 498), connectivity (see page 505).



INPUT OUTPUT

15.10 Drawing Graphs Nicely

Input description: A graph G.

Problem description: Draw a graph G so as to accurately reflect its structure.

Discussion: Graph drawing is a problem that constantly arises in applications, yet it is inherently ill-defined. What exactly does a nice drawing mean? We seek an algorithm that shows off the structure of the graph so the viewer can best understand it. Simultaneously, we seek a drawing that looks aesthetically pleasing.

Unfortunately, these are "soft" criteria for which it is impossible to design an optimization algorithm. Indeed, it is easy to come up with radically different drawings of a given graph, each of which is most appropriate in a certain context. Three different drawings of the Petersen graph are given on page 550. Which of these is the "right" one?

Several "hard" criteria can partially measure the quality of a drawing:

- *Crossings* – We seek a drawing with as few pairs of crossing edges as possible, since they are distracting.

- *Area* – We seek a drawing that uses as little paper as possible, while ensuring that no pair of vertices gets placed too close to each other.
- *Edge length* – We seek a drawing that avoids long edges, since they tend to obscure other features of the drawing.
- *Angular resolution* – We seek a drawing avoiding small angles between two edges incident on a given vertex, as the resulting lines tend to partially or fully overlap.
- *Aspect ratio* – We seek a drawing whose aspect ratio (width/height) reflects the desired output medium (typically a computer screen at 4/3) as closely as possible.

Unfortunately, these goals are mutually contradictory, and the problem of finding the best drawing under any nonempty subset of them is likely to be NP-complete.

Two final warnings before getting down to business. For graphs without inherent symmetries or structure, it is likely that no really nice drawing exists. This is especially for true for graphs with more than 10 to 15 vertices. The shear amount of ink needed to draw any large, dense graph will overwhelm any display. A drawing of the complete graph on 100 vertices (K_{100}) contains approximately 5,000 edges. On a 1000×1000 pixel display, this works out to 200 pixels per edge. What can you hope to see except a black blob in the center of the screen?

Once all this is understood, it must be admitted that graph-drawing algorithms can be quite effective and fun to play with. To help choose the right one, ask yourself the following questions:

- *Must the edges be straight, or can I have curves and/or bends?* – Straight-line drawing algorithms are relatively simple, but have their limitations. Orthogonal polyline drawings seem to work best to visualize complicated graphs such as circuit designs. *Orthogonal* means that all lines must be drawn either horizontal or vertical, with no intermediate slopes. *Polyline* means that each graph edge is represented by a chain of straight-line segments, connected by vertices or bends.
- *Is there a natural, application-specific drawing?* – If your graph represents a network of cities and roads, you are unlikely to find a better drawing than placing the vertices in the same position as the cities on a map. This same principle holds for many different applications.
- *Is your graph either planar or a tree?* – If so, use one of the special planar graph or tree drawing algorithms of Sections 15.11 and 15.12.
- *Is your graph directed?* – Edge direction has a significant impact on the nature of the desired drawing. When drawing directed acyclic graphs (DAGs), it is often important that all edges flow in a logical direction—perhaps left-right or top-down.

- *How fast must your algorithm be?* – Your graph drawing algorithm had better be very fast if it will be used for interactive update and display. You are presumably limited to using incremental algorithms, which change the vertex positions only in the immediate neighborhood of the edited vertex. You can afford more time for optimization if instead you are printing a pretty picture for extended study.

- *Does your graph contain symmetries?* – The output drawing above is attractive because the graph contains symmetries—namely two vertices identically connected to a core K_5. The inherent symmetries in a graph can be identified by computing its *automorphisms*, or self-isomorphisms. Graph isomorphism codes (see Section 16.9 (page 550)) can be readily used to find all automorphisms.

As a first quick and dirty drawing, I recommend simply spacing the vertices evenly on a circle, and then drawing the edges as straight lines between vertices. Such drawings are easy to program and fast to construct. They have the substantial advantage that no two edges will obscure each other, since no three vertices will be collinear. Such artifacts can be hard to avoid as soon as you allow internal vertices into your drawing. An unexpected pleasure with circular drawings is the symmetry sometimes revealed because vertices appear in the order they were inserted into the graph. Simulated annealing can be used to permute the circular vertex order to minimize crossings or edge length, and thus significantly improve the drawing.

A good, general purpose graph-drawing heuristic models the graph as a system of springs and then uses energy minimization to space the vertices. Let adjacent vertices attract each other with a force proportional to (say) the logarithm of their separation, while all nonadjacent vertices repel each other with a force proportional to their separation distance. These weights provide incentive for all edges to be as short as possible, while spreading the vertices apart. The behavior of such a system can be approximated by determining the force acting on each vertex at a particular time and then moving each vertex a small amount in the appropriate direction. After several such iterations, the system should stabilize on a reasonable drawing. The input and output figures above demonstrate the effectiveness of the spring embedding on a particular small graph.

If you need a polyline graph-drawing algorithm, my recommendation is that you study the systems presented next or described in [JM03] to decide whether one of them can do the job. You will have to do a significant amount of work before you can hope to develop a better algorithm.

Drawing your graph opens another can of worms, namely where to place the edge/vertex labels. We seek to position labels very close to the edges or vertices they identify, and yet to place them such that they do not overlap each other or other important graph features. Optimizing label placement can be shown to be an NP-complete problem, but heuristics related to bin packing (see Section 17.9 (page 595)) can be effectively used.

Implementations: GraphViz (*http://www.graphviz.org*) is a popular and well-supported graph-drawing program developed by Stephen North of Bell Laboratories. It represents edges as splines and can construct useful drawings of quite large and complicated graphs. It has sufficed for all of my professional graph-drawing needs over the years.

All of the graph data structure libraries of Section 12.4 (page 381) devote some effort to visualizing graphs. The Boost Graph Library provides an interface to GraphViz instead of reinventing the wheel. The Java graph libraries, most notably JGraphT (*http://jgrapht.sourceforge.net/*), are particularly suitable for interactive applications.

Graph drawing is a problem where very good commercial products exist, including those from Tom Sawyer Software (*www.tomsawyer.com*), yFiles (*www.yworks.com*), and iLOG's JViews (*www.ilog.com/products/jviews/*). Pajek [NMB05] is a package particularly designed for drawing social networks, and available at *http://vlado.fmf.uni-lj.si/pub/networks/pajek/*. All of these have free trial or noncommercial use downloads.

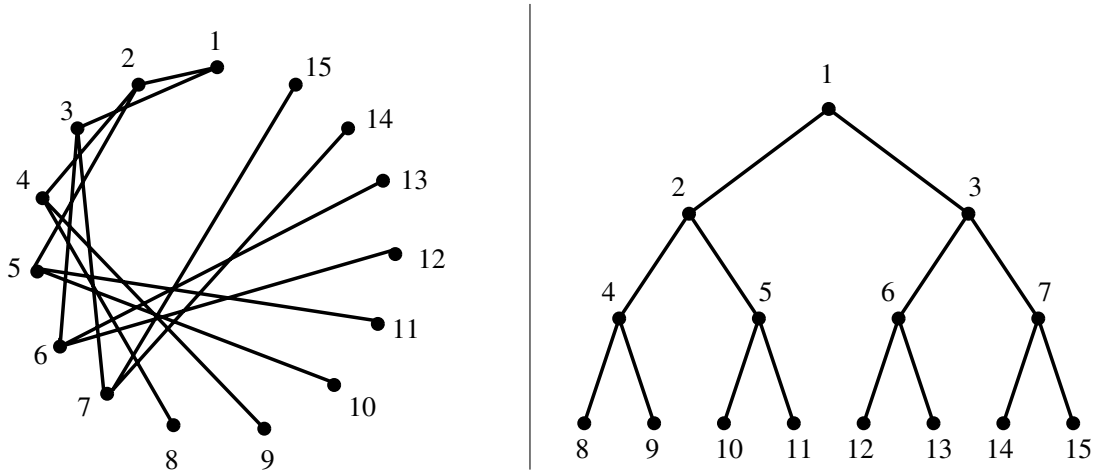
Combinatorica [PS03] provides Mathematica implementations of several graph-drawing algorithms, including circular, spring, and ranked embeddings. See Section 19.1.9 (page 661) for further information on Combinatorica.

Notes: A significant community of researchers in graph drawing exists, fueled by or fueling an annual conference on graph drawing. The proceedings of this conference are published by Springer-Verlag's Lecture Notes in Computer Science series. Perusing a volume of the proceedings will provide a good view of the state-of-the-art and of what kinds of ideas people are thinking about. The forthcoming *Handbook of Graph Drawing and Visualization* [Tam08] promises to be the most comprehensive review of the field.

Two excellent books on graph-drawing algorithms are Battista, et al. [BETT99] and Kaufmann and Wagner [KW01]. A third book by Jünger and Mutzel [JM03] is organized around systems instead of algorithms, but provides technical details about the drawing methods each system employs. Map-labeling heuristics are described in [BDY06, WW95].

It is trivial to space n points evenly along the boundary of a circle. However, the problem is considerably more difficult on the surface of a sphere. Extensive tables of such spherical codes for $n \leq 130$ have been constructed by Hardin, Sloane, and Smith [HSS07].

Related Problems: Drawing trees (see page 517), planarity testing (see page 520).



INPUT OUTPUT

15.11 Drawing Trees

Input description: A tree T, which is a graph without any cycles.

Problem description: Create a nice drawing of the tree T.

Discussion: Many applications require drawing pictures of trees. Tree diagrams are commonly used to display and traverse the hierarchical structure of file system directories. My attempts to Google "tree drawing software" revealed special-purpose applications for visualizing family trees, syntax trees (sentence diagrams), and evolutionary phylogenic trees all in the top twenty links.

Different aesthetics follow from each application. That said, the primary issue in tree drawing is establishing whether you are drawing free or rooted trees:

- *Rooted trees* define a hierarchical order, emanating from a single source node identified as the root. Any drawing should reflect this hierarchical structure, as well as any additional application-dependent constraints on the order in which children must appear. For example, family trees are rooted, with sibling nodes typically drawn from left to right in the order of birth.

- *Free trees* do not encode any structure beyond their connection topology. There is no root associated with the minimum spanning tree (MST) of a graph, so a hierarchical drawing will be misleading. Such free trees might well inherit their drawing from that of the full underlying graph, such as the map of the cities whose distances define the MST.

Trees are always planar graphs, and hence can and should be drawn so no two edges cross. Any of the planar drawing algorithms of Section 15.12 (page 520) could be used to do so. However, such algorithms are overkill, because much simpler algorithms can be used to construct planar drawings of trees. The spring-embedding heuristics of Section 15.10 (page 513) work well on free trees, although they may be too slow for certain applications.

The most natural tree-drawing algorithms assume rooted trees. However, they can be used equally well with free trees, after selecting one vertex to serve as the root of the drawing. This faux-root can be selected arbitrarily, or, even better, by using a *center* vertex of the tree. A center vertex minimizes the maximum distance to other vertices. For trees, the center always consists of either one vertex or two adjacent vertices. This tree center can be identified in linear time by repeatedly trimming all the leaves until only the center remains.

Your two primary options for drawing rooted trees are *ranked* and *radial* embeddings:

- *Ranked embeddings* – Place the root in the top center of your page, and then partition the page into the root-degree number of top-down strips. Deleting the root creates the root-degree number of subtrees, each of which is assigned to its own strip. Draw each subtree recursively, by placing its new root (the vertex adjacent to the old root) in the center of its strip a fixed distance down from the top, with a line from old root to new root. The output figure above is a nicely ranked embedding of a balanced binary tree.

  Such ranked embeddings are particularly effective for rooted trees used to represent a hierarchy—be it a family tree, data structure, or corporate ladder. The top-down distance illustrates how far each node is from the root. Unfortunately, such repeated subdivision eventually produces very narrow strips, until most of the vertices are crammed into a small region of the page. Try to adjust the width of each strip to reflect the total number of nodes it will contain, and don't be afraid of expanding into neighboring region's turf once their shorter subtrees have been completed.

- *Radial embeddings* – Free trees are better drawn using a radial embedding, where the root/center of the tree is placed in the center of the drawing. The space around this center vertex is divided into angular sectors for each subtree. Although the same problem of cramping will eventually occur, radial embeddings make better use of space than ranked embeddings and appear considerably more natural for free trees. The rankings of vertices in terms of distance from the center is illustrated by the concentric circles of vertices.

Implementations: GraphViz (*http://www.graphviz.org*) is a popular and well-supported graph-drawing program developed by Stephen North of Bell Laboratories. It represents edges as splines and can construct useful drawings of quite large

and complicated graphs. It has sufficed for all of my professional graph drawing needs over the years.

Graph/tree drawing is a problem where very good commercial products exist, including those from Tom Sawyer Software (*www.tomsawyer.com*), yFiles (*www.yworks.com*), and iLOG's JViews (*www.ilog.com/products/jviews/*). All of these have free trial or noncommercial use downloads.

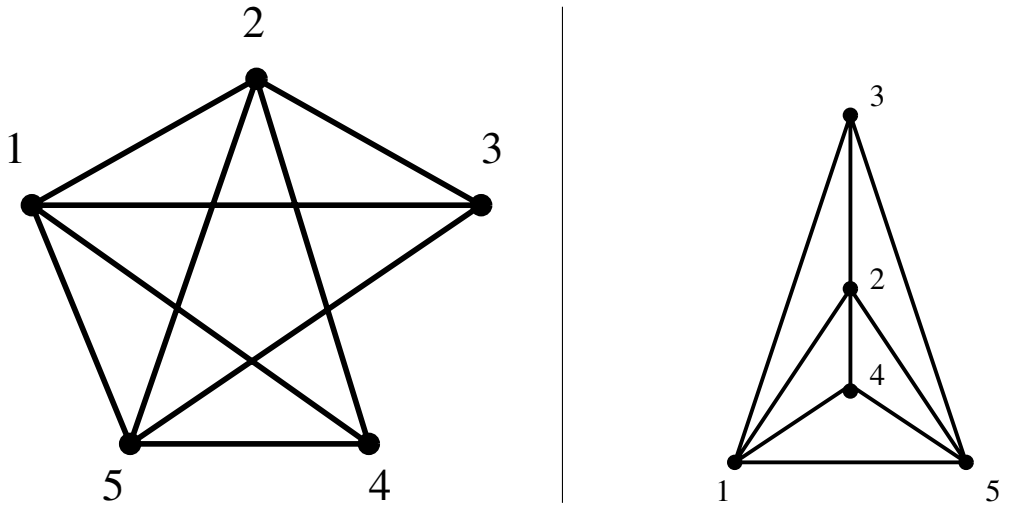
Combinatorica [PS03] provides Mathematica implementations of several tree drawing algorithms, including radial and rooted embeddings. See Section 19.1.9 (page 661) for further information on Combinatorica.

Notes: All books and surveys on graph drawing include discussions of specific tree-drawing algorithms. The forthcoming *Handbook of Graph Drawing and Visualization* [Tam08] promises to be the most comprehensive review of the field. Two excellent books on graph drawing algorithms are Battista, et al. [BETT99] and Kaufmann and Wagner [KW01]. A third book by Jünger and Mutzel [JM03] is organized around systems instead of algorithms, but provides technical details about the drawing methods each system employs.

Heuristics for tree layout have been studied by several researchers [RT81, Moe90], with Buchheim, et al. [BJL06] reflective of the state-of-the-art. Under certain aesthetic criteria, the problem is NP-complete [SR83].

Certain tree layout algorithms arise from non-drawing applications. The Van Emde Boas layout of a binary tree offers better external memory performance than conventional binary search, at a cost of greater complexity. See the survey of Arge, et al. [ABF05] for more on this and other cache-oblivious data structures.

Related Problems: Drawing graphs (see page 513), planar drawings (see page 520).



INPUT OUTPUT

15.12 Planarity Detection and Embedding

Input description: A graph G.

Problem description: Can G be drawn in the plane such that no two edges cross? If so, produce such a drawing.

Discussion: Planar drawings (or *embeddings*) make clear the structure of a given graph by eliminating crossing edges, which can be confused as additional vertices. Graphs defined by road networks, printed circuit board layouts, and the like are inherently planar because they are completely defined by surface structures.

Planar graphs have a variety of nice properties that can be exploited to yield faster algorithms for many problems. The most important fact to know is that every planar graph is *sparse*. Euler's formula shows that $|E| \leq 3|V| - 6$ for every nontrivial planar graph $G = (V, E)$. This means that every planar graph contains a linear number of edges, and further that every planar graph contains a vertex of degree ≤ 5. Every subgraph of a planar graph is planar, so there must always a sequence of low-degree vertices to delete from G, reducing it to the empty graph.

To gain a better appreciation of the subtleties of planar drawings, I encourage the reader to construct a planar (noncrossing) embedding for the graph $K_5 - e$, shown on the input figure above. Then try to construct such an embedding where all the edges are straight. Finally, add the missing edge to the graph and try to do the same for K_5 itself.

The study of planarity has motivated much of the development of graph theory. It must be confessed, however, that the need for planarity testing arises relatively infrequently in applications. Most graph-drawing systems do not explicitly seek planar embeddings. *"Planarity Detection"* proved to be among the least frequently hit pages of the Algorithm Repository (*http://www.cs.sunysb.edu/∼algorith*) [Ski99]. That said, it is still very useful to know how to deal with planar graphs when you encounter them.

Thus, it pays to distinguish the problem of planarity testing (does a graph have a planar drawing?) from constructing planar embeddings (actually finding the drawing), although both can be done in linear time. Many efficient planar graph algorithms do not make any use of the drawing, but instead exploit the low-degree deletion sequence described above.

Algorithms for planarity testing begin by embedding an arbitrary cycle from the graph in the plane and then considering additional paths in G, connecting vertices on this cycle. Whenever two such paths cross, one must be drawn outside the cycle and one inside. When three such paths mutually cross, there is no way to resolve the problem, so the graph cannot be planar. Linear-time algorithms for planarity detection are based on depth-first search, but they are subtle and complicated enough that you are wise to seek an existing implementation.

Such path-crossing algorithms can be used to construct a planar embedding by inserting the paths into the drawing one by one. Unfortunately, because they work in an incremental manner, nothing prevents them from inserting many vertices and edges into a relatively small area of the drawing. Such cramping is a major problem, for it leads to ugly drawings that are hard to understand. Better algorithms have been devised that construct *planar-grid embeddings*, where each vertex lies on a $(2n - 4) \times (n - 2)$ grid. Thus, no region can get too cramped and no edge can get too long. Still, the resulting drawings tend not to look as natural as one might hope.

For nonplanar graphs, what is often sought is a drawing that minimizes the number of crossings. Unfortunately, computing the crossing number of a graph is NP-complete. A useful heuristic extracts a large planar subgraph of G, embeds this subgraph, and then inserts the remaining edges one by one to minimize the number of crossings. This won't do much for dense graphs, which are doomed to have a large number of crossings, but it will work well for graphs that are almost planar, such as road networks with overpasses or printed circuit boards with multiple layers. Large planar subgraphs can be found by modifying planarity-testing algorithms to delete troublemaking edges when encountered.

Implementations: LEDA (see Section 19.1.1 (page 658)) includes linear-time algorithms for both planarity testing and constructing straight-line planar-grid embeddings. Their planarity tester returns an obstructing Kuratowski subgraph (see notes) for any graph deemed nonplanar, yielding concrete proof of its nonplanarity.

JGraphEd (*http://www.jharris.ca/JGraphEd/*) is a Java graph-drawing framework that includes several planarity testing/embedding algorithms, including both the Booth-Lueker PQ-tree algorithm and the modern straight-line grid embedding.

PIGALE (*http://pigale.sourceforge.net/*) is a C++ graph editor/algorithm library focusing on planar graphs. It contains a variety of algorithms for constructing planar drawings as well as efficient algorithms to test planarity and identify an obstructing subgraph ($K_{3,3}$ or K_5), if one exists.

Greedy randomized adaptive search (GRASP) heuristics for finding the largest planar subgraph have been implemented by Ribeiro and Resende [RR99] as Algorithm 797 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/*.

Notes: Kuratowski [Kur30] gave the first characterization of planar graphs, namely that they do not contain a subgraph homeomorphic to $K_{3,3}$ or K_5. Thus, if you are still working on the exercise to embed K_5, now is an appropriate time to give it up. Fary's theorem [FÁ48] states that every planar graph can be drawn in such a way that each edge is straight.

Hopcroft and Tarjan [HT74] gave the first linear-time algorithm for drawing graphs. Booth and Lueker [BL76] developed an alternate planarity-testing algorithm based on PQ-trees. Simplified planarity-testing algorithms include [BCPB04, MM96, SH99]. Efficient $2n \times n$ planar grid embeddings were first developed by [dFPP90]. The book by Nishizeki and Rahman [NR04] provide a good overview of the spectrum of planar drawing algorithms.

Outerplanar graphs are those that can be drawn so all vertices lie on the outer face of the drawing. Such graphs can be characterized as having no subgraph homeomorphic to $K_{2,3}$ and can be recognized and embedded in linear time.

Related Problems: Graph partition (see page 541), drawing trees (see page 517).

16

Graph Problems: Hard Problems

A cynical view of graph algorithms is that "everything we want to do is hard." Indeed, no polynomial-time algorithms are known for any of the problems in this section. All of them are provably NP-complete with the exception of graph isomorphism—whose complexity status remains an open question.

The theory of NP-completeness demonstrates that *all* NP-complete problems must have polynomial-time algorithms if *any* one of them does. This prospect is sufficiently preposterous that an NP-completeness reduction suffices as de facto proof that no efficient algorithm exists to solve the given problem.

Still, do not abandon hope if your problem resides in this chapter. We provide a recommended line of attack for each problem, be it combinatorial search, heuristics, approximation algorithms, or algorithms for restricted instances. Hard problems require a different methodology to work with than polynomial-time problems, but with care they can usually be dealt with successfully.

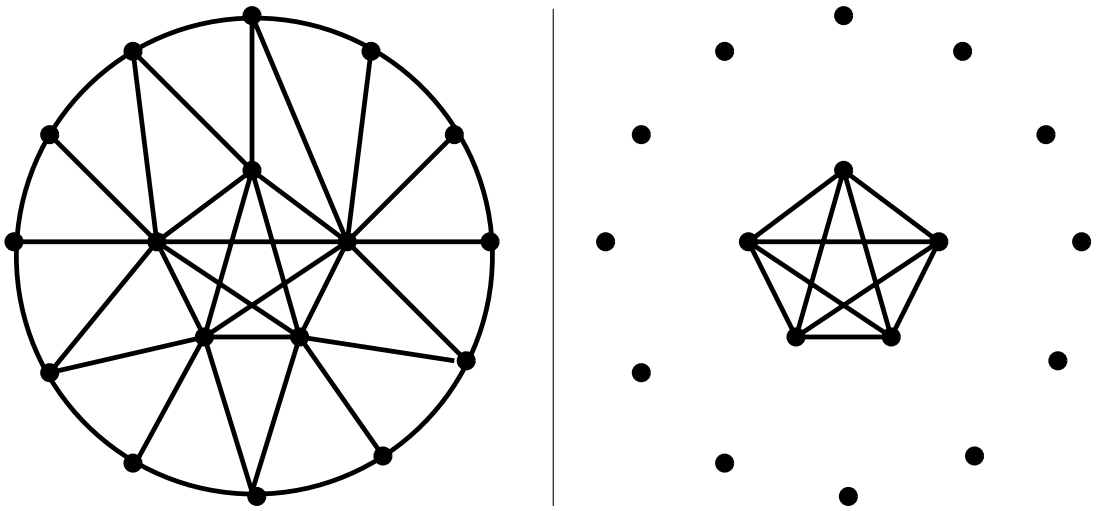
The following books will help you deal with NP-complete problems:

- *Garey and Johnson* [GJ79] – This is the classic reference on the theory of NP-completeness. Most notably, it contains a concise catalog of over 400 NP-complete problems, with associated references and comments. Browse through the catalog as soon as you question the existence of an efficient algorithm for your problem. Indeed, this is the single book in my library that I reach for most often.

- *Crescenzi and Kann* [ACG+03] – This book serves as the "Garey and Johnson" for the world of approximation algorithms. Its reference section, *The Compendium of NP Optimization Problems*, is maintained online at *www.nada.kth.se/∼viggo/problemlist/* and should be the first place to look for a provably good heuristic for any given problem.

S.S. Skiena, *The Algorithm Design Manual*, 2nd ed., DOI: 10.1007/978-1-84800-070-4_16,

- *Vazirani* [Vaz04] – A complete treatment of the theory of approximation algorithms by a highly regarded researcher in the field.
- *Hochbaum* [Hoc96] – This nice book was the first survey of approximation algorithms for NP-complete problems, but rapid developments have left it somewhat dated.
- *Gonzalez* [Gon07] – This *Handbook of Approximation Algorithms and Metaheuristics* contains current surveys on a variety of techniques for dealing with hard problems, both applied and theoretical.



INPUT OUTPUT

16.1 Clique

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

Problem description: What is the largest $S \subset V$ such that for all $x, y \in S$, $(x, y) \in E$?

Discussion: In high school, everybody complained about the "clique,"—a group of friends who all hung around together and seemed to dominate everything social. Consider a graph representing the school's social network. Vertices correspond to people, with edges between any pair of people who are friends. Thus, the high school clique defines a (complete subgraph) clique in this friendship graph.

Identifying "clusters" of related objects often reduces to finding large cliques in graphs. An interesting example arose in a program the Internal Revenue Service (IRS) developed to detect organized tax fraud. In this scam, large groups of phony tax returns are submitted in the hopes of getting undeserved refunds. But generating large numbers of *different* phony tax returns is hard work. The IRS constructs graphs with vertices corresponding to submitted tax forms and edges between any two forms that appear suspiciously similar. Any large clique in this graph points to fraud.

Since any edge in a graph represents a clique of two vertices, the challenge lies not in finding a clique, but in finding a large clique. And it is indeed a challenge, for finding a maximum clique is NP-complete. To make matters worse, it is provably

hard to approximate even to within a factor of $n^{1/2-\epsilon}$. Theoretically, clique is about as hard as a problem in this book can get. So what can we hope to do about it?

- *Will a maximal clique suffice?* – A *maximal* clique is a clique that cannot be enlarged by adding any additional vertex. This doesn't mean that it has to be large relative to the largest possible clique, but it might be. To find a nice maximal (and hopefully large) clique, sort the vertices from highest degree to lowest degree, put the first vertex in the clique, and then test each of the other vertices to see whether it is adjacent to all the clique vertices added thus far. If so, add it; if not, continue down the list. By using a bit vector to mark which vertices are currently in the clique, this can be made to run in $O(n+m)$ time. An alternative approach might incorporate some randomness into the vertex ordering, and accept the largest maximal clique you find after a certain number of trials.

- *What if I will settle for a large, dense subgraph?* – Insisting on cliques to define clusters in a graph can be risky, since a single missing edge will eliminate a vertex from consideration. Instead, we should seek large *dense* subgraphs—i.e. , subsets of vertices that contain a large number of edges between them. Cliques are, by definition, the densest subgraphs possible.

  The largest set of vertices whose induced (defined) subgraph has minimum vertex degree $\geq k$ can be found with a simple linear-time algorithm. Begin by deleting all the vertices whose degree is less than k. This may reduce the degree of other vertices below k, if they were adjacent to sufficiently deleted low-degree vertices. Repeating this process until all remaining vertices have degree $\geq k$ constructs the largest high-degree subgraph. This algorithm can be implemented in $O(n+m)$ time by using adjacency lists and the constant-width priority queue of Section 12.2 (page 373). If we continue to delete the lowest-degree vertices, we eventually end up with a clique or set of cliques, – but they may be as small as two vertices.

- *What if the graph is planar?* – Planar graphs cannot have cliques of a size larger than four, or else they cease to be planar. Since each edge defines a clique of size 2, the only interesting cases are cliques of three and four vertices. Efficient algorithms to find such small cliques consider the vertices from lowest to highest degree. Any planar graph must contain a vertex of at most 5 degrees (see Section 15.12 (page 520)), so there is only a constant-sized neighborhood to check exhaustively for a clique containing it. We then delete this vertex to leave a smaller planar graph, containing another low-degree vertex. Repeat this check and delete processes until the graph is empty.

If you *really* need to find the largest clique in a graph, an exhaustive search via backtracking provides the only real solution. We search through all k-subsets of the vertices, pruning a subset as soon as it contains a vertex that is not adjacent to all the rest. A simple upper bound on the maximum clique in G is the highest vertex

degree plus 1. A better upper bound comes from sorting the vertices in order of decreasing degree. Let j be the largest index such that degree of the jth vertex is at least $j-1$. The largest clique in the graph contains no more than j vertices, since no vertex of degree $< (j-1)$ can appear in a clique of size j. To speed our search, we should delete all such useless vertices from G.

Heuristics for finding large cliques based on randomized techniques such as simulated annealing are likely to work reasonably well.

Implementations: `Cliquer` is a set of C routines for finding cliques in arbitrary weighted graphs by Patric Östergard. It uses an exact branch-and-bound algorithm, and is available at *http://users.tkk.fi/~pat/cliquer.html*.

Programs for finding cliques and independent sets were sought for the Second DIMACS Implementation Challenge [JT96]. Programs and data from the challenge are available by anonymous FTP from *dimacs.rutgers.edu*. Source codes are available under *pub/challenge/graph* and test data under *pub/djs*. `dfmax.c` implements a simple-minded branch-and-bound algorithm similar to [CP90]. `dmclique.c` uses a "semi-exhaustive greedy" scheme for finding large independent sets from [JAMS91].

Kreher and Stinson [KS99] provide branch-and-bound programs in C for finding the maximum clique using a variety of lower-bounds, available at *http://www.math.mtu.edu/~kreher/cages/Src.html*.

GOBLIN (*http://www.math.uni-augsburg.de/~fremuth/goblin.html*) implements branch-and-bound algorithms for finding large cliques. They claim to be able to work with graphs as large as 150 to 200 vertices.

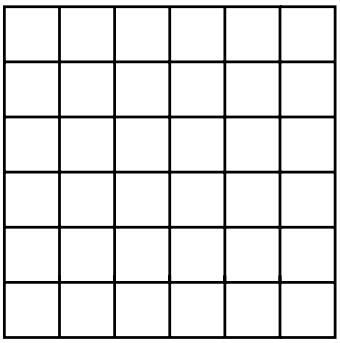
Notes: Bomze, et al. [BBPP99] give the most comprehensive survey on the problem of finding maximum cliques. Particularly interesting is the work from the operations research community on branch-and-bound algorithms for finding cliques effectively. More recent experimental results are reported in [JS01].

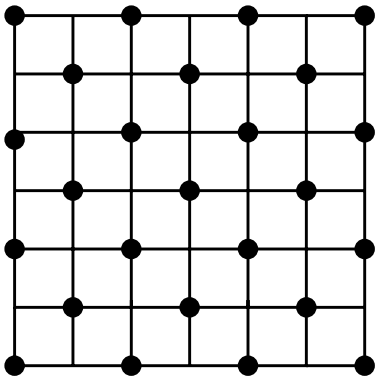
The proof that clique is NP-complete is due to Karp [Kar72]. His reduction (given in Section 9.3.3 (page 327)) established that clique, vertex cover, and independent set are very closely related problems, so heuristics and programs that solve one of them should also produce reasonable solutions for the other two.

The *densest subgraph problem* seeks the subset of vertices whose induced subgraph has the highest average vertex degree. A clique of k vertices is clearly the densest subgraph of its size, but larger, noncomplete subgraphs may achieve higher average degree. The problem is NP-complete, but simple heuristics based on repeatedly deleting the lowest-degree vertex achieve reasonable approximation ratios [AITT00]. See [GKT05] for an interesting application of densest subgraph, namely detecting link spam on the Web.

That clique cannot be approximated to within a factor of $n^{1/2-\epsilon}$ unless $P = NP$ (and $n^{1-\epsilon}$ under weaker assumptions) is shown in [Has82].

Related Problems: Independent set (see page 528), vertex cover (see page 530).





INPUT OUTPUT

16.2 Independent Set

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

Problem description: What is the largest subset S of vertices of V such that for each edge $(x, y) \in E$, either $x \notin E$ or $y \notin E$?

Discussion: The need to find large independent sets arises in facility dispersion problems, where we seek a set of mutually separated locations. It is important that no two locations of our new "McAlgorithm" franchise service be placed close enough to compete with each other. We can construct a graph where the vertices are the set of possible locations, and then add edges between any two locations deemed close enough to interfere. The maximum independent set gives the largest number of franchises we can sell without cannibalizing sales.

Independent sets (also known as *stable sets*) avoid conflicts between elements, and hence arise often in coding theory and scheduling problems. Define a graph whose vertices represent the set of possible code words, and add edges between any two code words sufficiently similar to be confused due to noise. The maximum independent set of this graph defines the highest capacity code for the given communication channel.

Independent set is closely related to two other NP-complete problems:

- *Clique* – Watch what you say, for a clique is what you get if you give an independent set a complement. The *complement* of $G = (V, E)$ is a graph $G' = (V, E')$ where $(i, j) \in E'$ iff (i, j) is not in E. In other words, we replace each edge by a non-edge and vice versa. The maximum independent set in G is exactly the maximum clique in G', so the two problems are algorithmically

identical. Thus, the algorithms and implementations in Section 16.1 (page 525) can easily be used for independent set.

- *Vertex coloring* – The vertex coloring of a graph $G = (V, E)$ is a partition of V into a small number of sets (colors), where no two vertices of the same color can have an edge between them. Each color class defines an independent set. Many scheduling applications of independent set are really coloring problems, since all tasks eventually must be completed.

  Indeed, one heuristic to find a large independent set is to use any vertex coloring algorithm/heuristic, and take the largest color class. One consequence of this observation is that all graphs with small chromatic numbers (such as planar and bipartite graphs) have large independent sets.

The simplest reasonable heuristic is to find the lowest-degree vertex, add it to the independent set, and then delete it and all vertices adjacent to it. Repeating this process until the graph is empty gives a *maximal* independent set, in that it can't be made larger by just adding vertices. Using randomization or perhaps some degree of exhaustive search might result in somewhat larger independent sets.

The independent set problem is in some sense dual to the graph-matching problem. The former asks for a large set of vertices with no edge in common, while the latter asks for a large set of edges with no vertex in common. This suggests trying to rephrase your problem as an efficiently-computable matching problem instead of maximum independent set problem, which is NP-complete.

The maximum independent set of a tree can be found in linear time by (1) stripping off the leaf nodes, (2) adding them to the independent set, (3) deleting all adjacent nodes, and then (4) repeating from the first step on the resulting trees until it is empty.

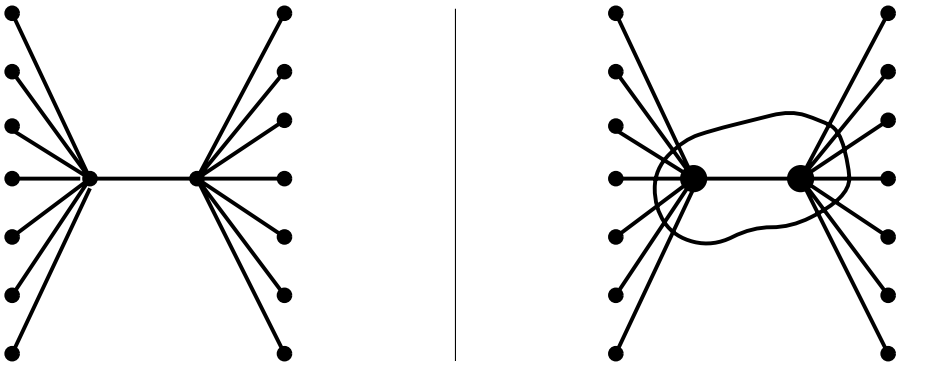
Implementations: Any program for computing the maximum clique in a graph can find maximum independent sets by just complementing the input graph. Therefore, we refer the reader to the clique-finding programs of Section 16.1 (page 525).

GOBLIN (*http://www.math.uni-augsburg.de/~fremuth/goblin.html*) implements a branch-and-bound algorithm for finding independent sets (called stable sets in the manual).

Greedy randomized adaptive search (GRASP) heuristics for independent set set have been implemented by Resende, et al. [RFS98] as Algorithm 787 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/*.

Notes: The proof that independent set is NP-complete is due to Karp [Kar72]. It remains NP-complete for planar cubic graphs [GJ79]. Independent set can be solved efficiently for bipartite graphs [Law76]. This is not trivial—indeed the larger of the "part" of a bipartite graph is not necessarily its maximum independent set.

Related Problems: Clique (see page 525), vertex coloring (see page 544), vertex cover (see page 530).



INPUT OUTPUT

16.3 Vertex Cover

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

Problem description: What is the smallest subset of $S \subset V$ such that each edge $(x, y) \in E$ contains at least one vertex of S?

Discussion: Vertex cover is a special case of the more general *set cover* problem, which takes as input an arbitrary collection of subsets $S = (S_1, \ldots, S_n)$ of the universal set $U = \{1, \ldots, m\}$. We seek the smallest subset of subsets from S whose union is U. Set cover arises in many applications associated with buying things sold in fixed lots or assortments. See Section 18.1 (page 621) for a discussion of set cover.

To turn vertex cover into a set cover problem, let universal set U represent the set E of edges from G, and define S_i to be the set of edges incident on vertex i. A set of vertices defines a vertex cover in graph G iff the corresponding subsets define a set cover in this particular instance. However, since each edge can be in only two different subsets, vertex cover instances are simpler than general set cover. Vertex cover is a relative lightweight among NP-complete problems, and can be more effectively solved than general set cover.

Vertex cover and independent set are very closely related graph problems. Since every edge in E is (by definition) incident on a vertex in any cover S, there can be no edge both endpoints are in $V - S$. Thus, $V - S$ must be an independent set. Since minimizing S is the same as maximizing $V - S$, the problems are equivalent. This means that any independent set solver can be applied to vertex cover as well. Having two ways of looking at your problem is helpful, since one may appear easier in a given context.

The simplest heuristic for vertex cover selects the vertex with highest degree, adds it to the cover, deletes all adjacent edges, and then repeats until the graph is empty. With the right data structures, this can be done in linear time, and should "usually" produce a "pretty good" cover. However, this cover might be $\lg n$ times worse than the optimal cover for certain input graphs.

Fortunately, we can always find a vertex cover whose size is at most twice as large as optimal. Find a *maximal* matching M in the graph—i.e. , a set of edges no two of which share a vertex in common and which cannot be enlarged by adding additional edges. Such a maximal matching can be constructed incrementally, by picking an arbitrary edge e in the graph, deleting any edge sharing a vertex with e, and repeating until the graph is out of edges. Taking *both* of the vertices for each edge in a maximal matching gives us a vertex cover. Why? Because *any* vertex cover must contain *at least* one of the two vertices in each matching edge just to cover the edges of M, this cover is at most twice as large as the minimum cover.

This heuristic can be tweaked to perform somewhat better in practice, if not in theory. We can select the matching edges to "kill off" as many other edges as possible, which should reduce the size of the maximal matching and hence the number of pairs of vertices in the vertex cover. Also, some of the vertices from M may in fact not be necessary, since all of their incident edges might also have been covered using other selected vertices. We can identify and delete these losers by making a second pass through our cover.

The vertex cover problem seeks to cover all edges using few vertices. Two other important problems have similar sounding objectives:

- *Cover all vertices using few vertices* – The *dominating set* problem seeks the smallest set of vertices D such that every vertex in $V - D$ is adjacent to at least one vertex in the dominating set D. Every vertex cover of a nontrivial connected graph is also a dominating set, but dominating sets can be much smaller. Any single vertex represents the minimum dominating set of complete graph K_n, while $n-1$ vertices are needed for a vertex cover. Dominating sets tend to arise in communications problems, since they represent the hubs or broadcast centers sufficient to communicate with all sites/users.

  Dominating set problems can be easily expressed as instances of set cover (see Section 18.1 (page 621)). Each vertex v_i defines a subset of vertices consisting of itself plus all the vertices it is adjacent to. The greedy set cover heuristic running on this instance yields a $\Theta(\lg n)$ approximation to the optimal dominating set.

- *Cover all vertices using few edges* – The *edge cover* problem seeks the smallest set of edges such that each vertex is included in one of the edges. In fact, edge cover can be solved efficiently by finding a maximum cardinality matching (see Section 15.6 (page 498)) and then selecting arbitrary edges to account for the unmatched vertices.

Implementations: Any program for computing the maximum clique in a graph can be applied to vertex cover by complementing the input graph and selecting the vertices which do not appear in the clique. Therefore, we refer the reader to check out the clique-finding programs of Section 16.1 (page 525).

`COVER` [RHG07] is a very effective vertex cover solver based on a stochastic local search algorithm. It is available at *http://www.nicta.com.au/people/richters/*.

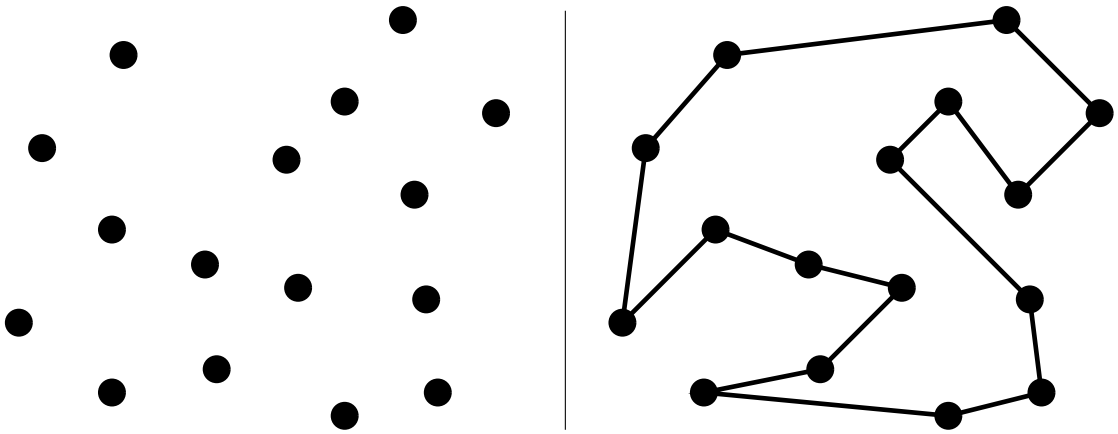
JGraphT (*http://jgrapht.sourceforge.net/*) is a Java graph library that contains greedy and 2-approximate heuristics for vertex cover.

Notes: Karp [Kar72] first proved that vertex-cover is NP-complete. Several different heuristics yield 2-approximation algorithms for vertex cover, including randomized rounding. Good expositions on these 2-approximation algorithms include [CLRS01, Hoc96, Pas97, Vaz04]. The example that the greedy algorithm can be as bad as $\lg n$ times optimal is due to [Joh74] and presented in [PS98]. Experimental studies of vertex cover heuristics include [GMPV06, GW97, RHG07].

Whether there exists a better than 2-factor approximation for vertex cover is one of the major open problems in approximation algorithms. Hastad [Has97] proved there does not exist a better than 1.1666-factor approximation algorithm for vertex cover.

The primary reference on dominating sets is the monograph of Haynes et al. [HHS98]. Heuristics for the connected dominating set problem are presented in [GK98]. Dominating set cannot be approximated to better than the $\Omega(\lg n)$ factor [ACG+03] of set cover.

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



INPUT OUTPUT

16.4 Traveling Salesman Problem

Input description: A weighted graph G.

Problem description: Find the cycle of minimum cost, visiting each vertex of G exactly once.

Discussion: The traveling salesman problem is the most notorious NP-complete problem. This is a function both of its general usefulness and the ease with which it can be explained to the public at large. Imagine a traveling salesman planning a car trip to visit a set of cities. What is the shortest route that will enable him to do so and return home, thus minimizing his total driving?

The traveling salesman problem arises in many transportation and routing problems. Other important applications involve optimizing tool paths for manufacturing equipment. For example, consider a robot arm assigned to solder all the connections on a printed circuit board. The shortest tour that visits each solder point exactly once defines the most efficient route for the robot.

Several issues arise in solving TSPs:

- *Is the graph unweighted?* – If the graph is unweighted, or all the edges have one of two possible cost values, the problem reduces to finding a *Hamiltonian cycle*. See Section 16.5 (page 538) for a discussion of this problem.

- *Does your input satisfy the triangle inequality?* – Our sense of how proper distance measures behave is captured by the *triangle inequality*. This property states that $d(i, j) \leq d(i, k) + d(k, j)$ for all vertices $i, j, k \in V$. Geometric

distances all satisfy the triangle inequality because the shortest distance between two points is as the crow flies. Commercial air fares do *not* satisfy the triangle inequality, which is why it is so hard to find the cheapest airfare between two points. TSP heuristics work much better on sensible graphs that do obey the triangle inequality.

- *Are you given n points as input or a weighted graph?* – Geometric instances are often easier to work with than a graph representation. Since pair of points define a complete graph, there is never an issue of finding a feasible tour. We can save space by computing these distances on demand, thus eliminating the need to store an $n \times n$ distance matrix. Geometric instances inherently satisfy the triangle inequality, so they can exploit performance guarantees from certain heuristics. Finally, one can take advantage of geometric data structures like kd-trees to quickly identify close unvisited sites.

- *Can you visit a vertex more than once?* – The restriction that the tour not revisit any vertex is irrelevant in many applications. In air travel, the cheapest way to visit all vertices might involve repeatedly visiting an airport hub. Note that this issue does not arise when the input observes the triangle inequality.

  TSP with repeated vertices is easily solved by using any conventional TSP code on a new cost matrix D, where $D(i, j)$ is the shortest path distance from i to j. This matrix can be constructed by solving an all-pairs shortest path (see Section 15.4 (page 489)) and satisfies the triangle inequality.

- *Is your distance function symmetric?* – A distance function is *asymmetric* when there exists x, y such that $d(x, y) \neq d(y, x)$. The asymmetric traveling salesman problem (ATSP) is much harder to solve in practice than symmetric (STSP) instances. Try to avoid such pathological distance functions. Be aware that there is a reduction converting ATSP instances to symmetric instances containing twice as many vertices [GP07], that may be useful because symmetric solvers are so much better.

- *How important is it to find the optimal tour?* – Usually heuristic solutions will suffice for applications. There are two different approaches if you insist on solving your TSP to optimality, however. *Cutting plane methods* model the problem as an integer program, then solve the linear programming relaxation of it. Additional constraints designed to force integrality are added if the optimal solution is not at an integer point. *Branch-and-bound algorithms* perform a combinatorial search while maintaining careful upper and lower bounds on the cost of a tour. In the hands of professionals, problems with thousands of vertices can be solved. Maybe you can too, if you use the best solver available.

Almost any flavor of TSP is going to be NP-complete, so the right way to proceed is with heuristics. These typically come within a few percent of the optimal

solution, which is close enough for engineering work. Unfortunately, literally dozens of heuristics have been proposed for TSP, so the situation can be confusing. Empirical results in the literature are sometime contradictory. However, we recommend choosing from among the following heuristics:

- *Minimum spanning trees* – This heuristic starts by finding the minimum spanning tree (MST) of the sites, and then does a depth-first search of the resulting tree. In the course of DFS, we walk over each of the $n-1$ edges exactly twice: once going down to discover a new vertex, and once going up when we backtrack. Now define a tour by ordering the vertices by when they were discovered. If the graph obeys the triangle inequality, the resulting tour is at most twice the length of the optimal TSP tour. In practice, it is usually better, typically 15% to 20% over optimal. Furthermore, the running time is bounded by that of computing the MST, which is only $O(n \lg n)$ in the case of points in the plane (see Section 15.3 (page 484)).

- *Incremental insertion methods* – A different class of heuristics starts from a single vertex, and then inserts new points into this partial tour one at a time until the tour is complete. The version of this heuristic that seems to work best is *furthest point* insertion: of all remaining points, insert the point v into a partial tour T such that

  $$\max_{v \in V} \min_{i=1}^{|T|} (d(v, v_i) + d(v, v_{i+1}))$$

  The "min" ensures that we insert the vertex in the position that adds the smallest amount of distance to the tour, while the "max" ensures that we pick the worst such vertex first. This seems to work well because it "roughs out" a partial tour first before filling in details. Such tours are typically only 5% to 10% longer than optimal.

- *K-optimal tours* – Substantially more powerful are the Kernighan-Lin, or *k-opt*, class of heuristics. The method applies local refinements to an initially arbitrary tour in the hopes of improving it. In particular, subsets of k edges are deleted from the tour and the k remaining subchains rewired to form a different tour with hopefully a better cost. A tour is k-optimal when no subset of k edges can be deleted and rewired to reduce the cost of the tour. Two-opting a tour is a fast and effective way to improve any other heuristic. Extensive experiments suggest that 3-optimal tours are usually within a few percent of the cost of optimal tours. For $k > 3$, the computation time increases considerably faster than the solution quality. Simulated annealing provides an alternate mechanism to employ edge flips to improve heuristic tours.

Implementations: Concorde is a program for the symmetric traveling salesman problem and related network optimization problems, written in ANSI C. This

world record-setting program by Applegate, Bixby, Chvatal, and Cook [ABCC07] has obtained the optimal solutions to 106 of TSPLIB's 110 instances; the largest of which has 15,112 cities. Concorde is available for academic research use from *http://www.tsp.gatech.edu/concorde*. It is the clear choice among available TSP codes. Their *http://www.tsp.gatech.edu/* website features very interesting material on the history and applications of TSP.

Lodi and Punnen [LP07] put together an excellent survey of available software for solving TSP. Current links to all programs mentioned are maintained at *http://www.or.deis.unibo.it/research_pages/tspsoft.html*.

TSPLIB [Rei91] provides the standard collection of hard instances of TSPs that arise in practice. The best-supported version of TSPLIB is available from *http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/*, although the instances are also available from Netlib (see Section 19.1.5 (page 659)).

Tsp_solve is a C++ code by Chad Hurwitz and Robert Craig that provides both heuristic and optimal solutions. Geometric problems of size up to 100 points are manageable. It is available from *http://www.cs.sunysb.edu/∼algorith* or by e-mailing Chad Hurrwitz at churritz@cts.com. GOBLIN (*http://www.math.uni-augsburg.de/∼fremuth/goblin.html*) implements branch-and-bound algorithms for both symmetric and asymmetric TSP, as well as a variety of heuristics.

Algorithm 608 [Wes83] of the *Collected Algorithms of the ACM* is a Fortran implementation of a heuristic for the quadratic assignment problem—a more general problem that includes the traveling salesman as a special case. Algorithm 750 [CDT95] is a Fortran code for the exact solution of asymmetric TSP instances. See Section 19.1.5 (page 659) for details.

Notes: The book by Applegate, Bixby, Chvatal, and Cook [ABCC07] documents the techniques they used in their record-setting TSP solvers, as well as the theory and history behind the problem. Gutin and Punnen [GP07] now offer the best reference on all aspects and variations of the traveling salesman problem, displacing an older but much beloved book by Lawler et al. [LLKS85].

Experimental results on heuristic methods for solving large TSPs include [Ben92a, GBDS80, Rei94]. Typically, it is possible to get within a few percent of optimal with such methods.

The Christofides heuristic [Chr76] is an improvement over the minimum-spanning tree heuristic and guarantees a tour whose cost is at most 3/2 times optimal on Euclidean graphs. It runs in $O(n^3)$, where the bottleneck is the time it takes to find a minimum-weight perfect matching (see Section 15.6 (page 498)). The minimum spanning tree heuristic is due to [RSL77].

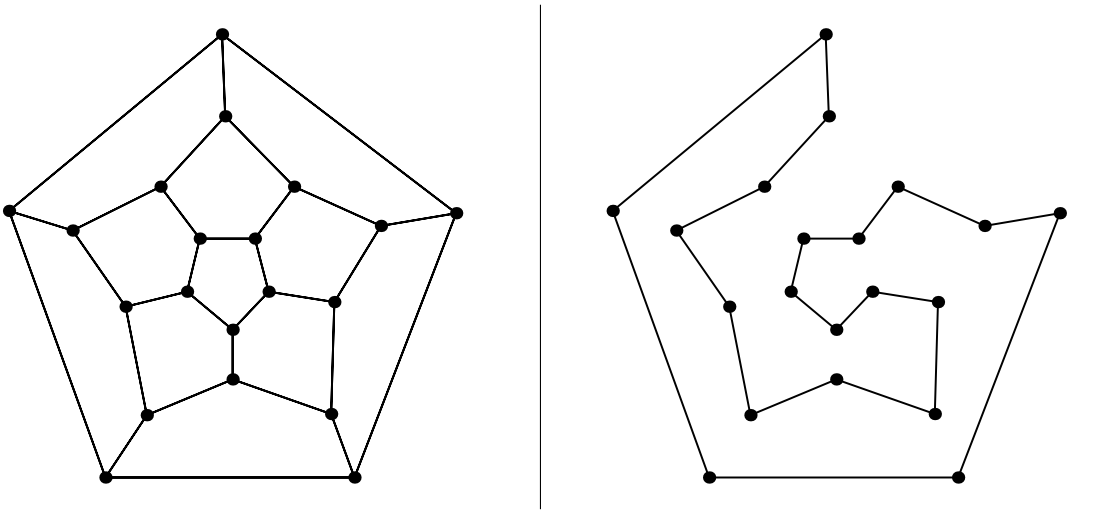
Polynomial-time approximation schemes for Euclidean TSP have been developed by Arora [Aro98] and Mitchell [Mit99], which offer $1 + \epsilon$ factor approximations in polynomial time for any $\epsilon > 0$. They are of great theoretical interest, although any practical consequences remain to be determined.

The history of progress on optimal TSP solutions is inspiring. In 1954, Dantzig, Fulkerson, and Johnson solved a symmetric TSP instance of 42 United States cities [DFJ54]. In 1980, Padberg and Hong solved an instance on 318 vertices [PH80]. Applegate et al. [ABCC07] have recently solved problems that are twenty times larger than this. Some of

this increase is due to improved hardware, but most is due to better algorithms. The rate of growth demonstrates that exact solutions to NP-complete problems can be obtained for large instances if the stakes are high enough. Fortunately or unfortunately, they seldom are.

Size is not the only criterion for hard instances. One can easily construct an enormous graph consisting of one cheap cycle, for which it would be easy to find the optimal solution. For sets of points in convex position in the plane, the minimum TSP tour is described by its convex hull (see Section 17.2 (page 568)), which can be computed in $O(n \lg n)$ time. Other easy special cases are known.

Related Problems: Hamiltonian cycle (see page 538), minimum spanning tree (see page 484), convex hull (see page 568).



INPUT OUTPUT

16.5 Hamiltonian Cycle

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

Problem description: Find a tour of the vertices using only edges from G, such that each vertex is visited exactly once.

Discussion: Finding a Hamiltonian cycle or path in a graph G is a special case of the traveling salesman problem G'—one where each edge in G has distance 1 in $'G$. Non-edge vertex pairs are separated by a greater distance, say 2. Such a weighted graph has TSP tour of cost n in G' iff G is Hamiltonian.

Closely related is the problem of finding the longest path or cycle in a graph. This arises often in pattern recognition problems. Let the vertices in the graph correspond to possible symbols, with edges linking pairs of symbols that might occur next to each other. The longest path through this graph is a good candidate for the proper interpretation.

The problems of finding longest cycles and paths are both NP-complete, even on very restrictive classes of unweighted graphs. There are several possible lines of attack, however:

- *Is there a serious penalty for visiting vertices more than once?* – Reformulating the Hamiltonian cycle problem instead of minimizing the total number of vertices visited on a complete tour turns it into an optimization problem.

This allows possibilities for heuristics and approximation algorithms. Finding a spanning tree of the graph and doing a depth-first search, as discussed in Section 16.4 (page 533), yields a tour with at most $2n$ vertices. Using randomization or simulated annealing might bring the size of this down considerably.

- *Am I seeking the longest path in a directed acyclic graph (DAG)?* – The problem of finding the longest path in a DAG can be solved in linear time using dynamic programming. Conveniently, the algorithm for finding the *shortest* path in a DAG (presented in Section 15.4 (page 489)) does the job if we replace min with max. DAGs are the most interesting case of longest path for which efficient algorithms exist.

- *Is my graph dense?* – Sufficiently dense graphs always contain Hamiltonian cycles. Further, the cycles implied by such sufficiency conditions can be efficiently constructed. In particular, any graph where all vertices have degree $\geq n/2$ must be Hamiltonian. Stronger sufficient conditions also hold; see the Notes section.

- *Are you visiting all the vertices or all the edges?* – Verify that you really have a vertex-tour problem and not an edge-tour problem. With a little cleverness it is sometimes possible to reformulate a Hamiltonian cycle problem in terms of Eulerian cycles, which instead visit every edge of a graph. Perhaps the most famous such instance is the problem of constructing de Bruijn sequences, discussed in Section 15.7 (page 502). The benefit is that fast algorithms exist for Eulerian cycles and many related variants, while Hamiltonian cycle is NP-complete.

If you *really* must know whether your graph is Hamiltonian, backtracking with pruning is your only possible solution. Certainly check whether your graph is biconnected (see Section 15.8 (page 505)). If not, this means that the graph has an articulation vertex whose deletion will disconnect the graph and so cannot be Hamiltonian.

Implementations: The construction described above (weight 1 for an edge and 2 for a non-edge) reduces Hamiltonian cycles to a symmetric TSP problem that obeys the triangle inequality. Thus we refer the reader to the TSP solvers discussed in Section 16.4 (page 533). Foremost among them is `Concorde`, a program for the symmetric traveling salesman problem and related network optimization problems, written in ANSI C. `Concorde` is available for academic research use from *http://www.tsp.gatech.edu/concorde*. It is the clear choice among available TSP codes.

An effective program for solving Hamiltonian cycle problems resulted from the masters thesis of Vandegriend [Van98]. Both the code and the thesis are available from *http://web.cs.ualberta.ca/∼joe/Theses/vandegriend.html*.

Lodi and Punnen [LP07] put together an excellent survey of available TSP software, including the special case of Hamiltonian cycle. Current links to all programs are maintained at *http://www.or.deis.unibo.it/research_pages/tspsoft.html*.

The football program of the Stanford GraphBase (see Section 19.1.8 (page 660)) uses a stratified greedy algorithm to solve the asymmetric longest-path problem. The goal is to derive a chain of football scores to establish the superiority of one football team over another. After all, if Virginia beat Illinois by 30 points, and Illinois beat Stony Brook by 14 points, then by transitivity Virginia would beat Stony Brook by 44 points if they played, right? We seek the longest simple path in a graph where the weight of edge (x, y) denotes the number of points by which x beat y.

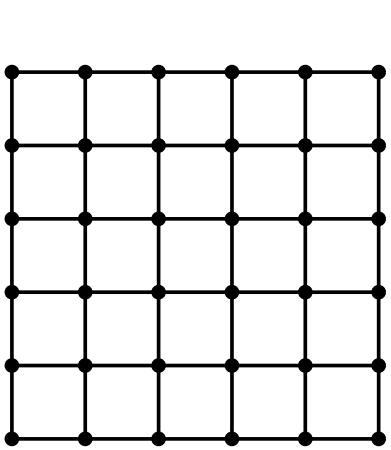
Nijenhuis and Wilf [NW78] provide an efficient routine to enumerate all Hamiltonian cycles of a graph by backtracking. See Section 19.1.10 (page 661). Algorithm 595 [Mar83] of the *Collected Algorithms of the ACM* is a similar Fortran code that can be used as either an exact procedure or a heuristic by controlling the amount of backtracking. See Section 19.1.5 (page 659).

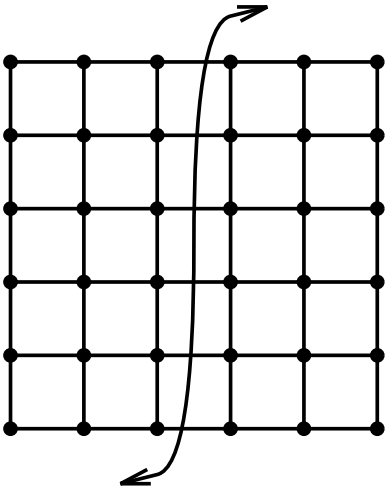
Notes: Hamiltonian cycles apparently first arose in Euler's study of the knight's tour problem, although they were popularized by Hamilton's "Around the World" game in 1839. See [ABCC07, GP07, LLKS85] for comprehensive references on the traveling salesman problem, including discussions on Hamiltonian cycle.

Most good texts in graph theory review sufficiency conditions for graphs to be Hamiltonian. My favorite is West [Wes00].

Techniques for solving optimization problems in the laboratory using biological processes have attracted considerable attention. In the original application of these "biocomputing" techniques, Adleman [Adl94] solved a seven-vertex instance of the directed Hamiltonian path problem. Unfortunately, this approach requires an exponential number of molecules, and Avogadro's number implies that such experiments are inconceivable for graphs beyond $n \approx 70$.

Related Problems: Eulerian cycle (see page 502), traveling salesman (see page 533).





INPUT OUTPUT

16.6 Graph Partition

Input description: A (weighted) graph $G = (V, E)$ and integers k and m.

Problem description: Partition the vertices into m roughly equal-sized subsets such that the total edge cost spanning the subsets is at most k.

Discussion: Graph partitioning arises in many divide-and-conquer algorithms, which gain their efficiency by breaking problems into equal-sized pieces such that the respective solutions can easily be reassembled. Minimizing the number of edges cut in the partition usually simplifies the task of merging.

Graph partition also arises when we need to cluster the vertices into logical components. If edges link "similar" pairs of objects, the clusters remaining after partition should reflect coherent groupings. Large graphs are often partitioned into reasonable-sized pieces to improve data locality or make less cluttered drawings.

Finally, graph partition is a critical step in many parallel algorithms. Consider the finite element method, which is used to compute the physical properties (such as stress and heat transfer) of geometric models. Parallelizing such calculations requires partitioning the models into equal-sized pieces whose interface is small. This is a graph-partitioning problem, since the topology of a geometric model is usually represented using a graph.

Several different flavors of graph partitioning arise depending on the desired objective function:

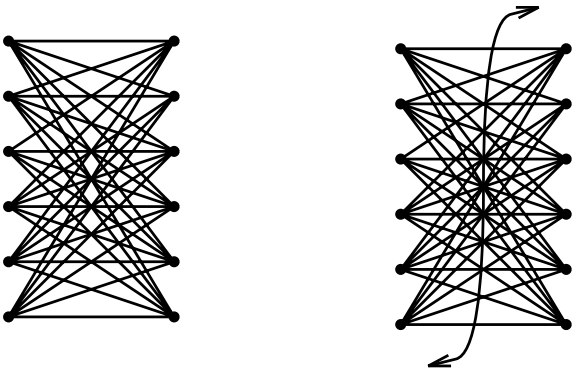


Figure 16.1: The maximum cut of a graph

- *Minimum cut set* – The *smallest* set of edges to cut that will disconnect a graph can be efficiently found using network flow or randomized algorithms. See Section 15.8 (page 505) for more on connectivity algorithms. The smallest cutset might split off only a single vertex, so the resulting partition could be very unbalanced.

- *Graph partition* – A better partition criterion seeks a small cut that partitions the vertices into roughly equal-sized pieces. Unfortunately, this problem is NP-complete. Fortunately, the heuristics discussed below work well in practice.

  Certain special graphs always have small *separators*, that partition the vertices into balanced pieces. For any tree, there always exists a single vertex whose deletion partitions the tree so that no component contains more than $n/2$ of the original n vertices. These components need not always be connected; consider the separating vertex of a star-shaped tree. This separating vertex can be found in linear time using depth first-search. Every planar graph has a set of $O(\sqrt{n})$ vertices whose deletion leaves no component with more than $2n/3$ vertices. These separators provide a useful way to decompose geometric models, which are often defined by planar graphs.

- *Maximum cut* – Given an electronic circuit specified by a graph, the *maximum cut* defines the largest amount of data communication that can simultaneously occur in the circuit. The highest-speed communications channel should thus span the vertex partition defined by the maximum edge cut. Finding the maximum cut in a graph is NP-complete [Kar72], however heuristics similar to those of graph partitioning work well.

The basic approach for dealing with graph partitioning or max-cut problems is to construct an initial partition of the vertices (either randomly or according

to some problem-specific strategy) and then sweep through the vertices, deciding whether the size of the cut would improve if we moved this vertex over to the other side. The decision to move vertex v can be made in time proportional to its degree, by identifying which side of the partition contains more of v's neighbors. Of course, the desirable side for v may change after its neighbors jump, so several iterations are likely to be needed before the process converges on a local optimum. Even so, such a local optimum can be arbitrarily far away from the global max-cut.

There are many variations of this basic procedure, by changing the order we test the vertices in or moving clusters of vertices simultaneously. Using some form of randomization, particularly simulated annealing, is almost certain to be a good idea. When more than two components are desired, the partitioning heuristic should be applied recursively.

Spectral partitioning methods use sophisticated linear algebra techniques to obtain a good partitioning. The spectral bisection method uses the second-lowest eigenvector of the *Laplacian matrix* of the graph to partition it into two pieces. Spectral methods tend to do a good job of identifying the general area to partition, but the results can be improved by cleaning up with a local optimization method.

Implementations: Chaco is a widely-used graph partitioning code designed to partition graphs for parallel computing applications. It employs several different partitioning algorithms, including both Kernighan-Lin and spectral methods. Chaco is available at *http://www.cs.sandia.gov/~bahendr/chaco.html*.

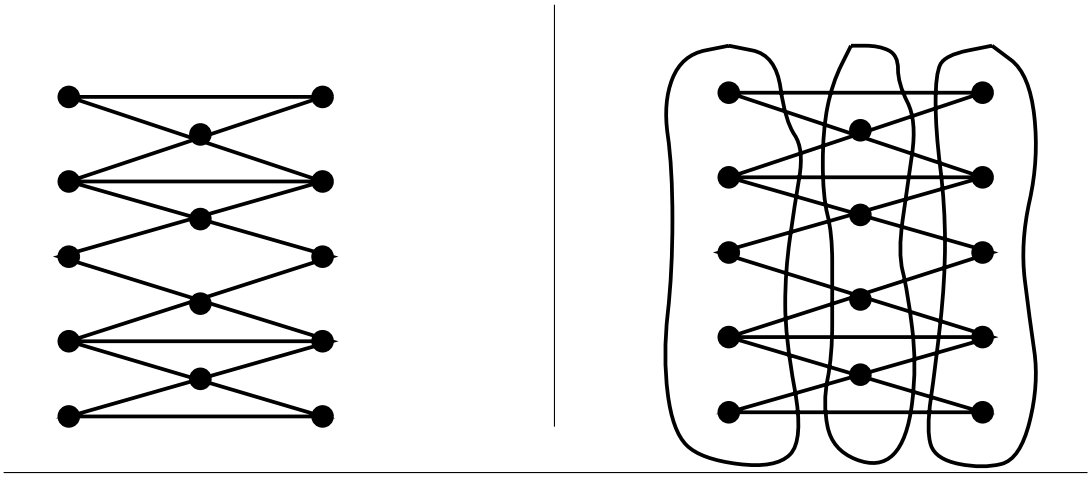
METIS (*http://glaros.dtc.umn.edu/gkhome/views/metis*) is another well-regarded code for graph partitioning. It has successfully partitioned graphs with over 1,000,000 vertices. Available versions include one variant designed to run on parallel machines and another suitable for partitioning hypergraphs. Other respected codes include Scotch (*http://www.labri.fr/perso/pelegrin/scotch/*) and JOSTLE (*http://staffweb.cms.gre.ac.uk/~wc06/jostle/*).

Notes: The fundamental local improvement heuristics for graph partitioning are due to Kernighan-Lin [KL70] and Fiduccia-Mattheyses [FM82]. Spectral methods for graph partition are discussed in [Chu97, PSL90]. Empirical results on graph partitioning heuristics include [BG95, LR93].

The planar separator theorem and an efficient algorithm for finding such a separator are due to Lipton and Tarjan [LT79, LT80]. For experiences in implementing planar separator algorithms, see [ADGM04, HPS+05].

Any random vertex partition will expect to cut half of the edges in the graph, since the probability that the two vertices defining an edge end up on different sides of the partition is 1/2. Goemans and Williamson [GW95] gave an 0.878-factor approximation algorithm for maximum-cut, based on semi-definite programming techniques. Tighter analysis of this algorithm was followed by Karloff [Kar96].

Related Problems: Edge/vertex connectivity (see page 505), network flow (see page 509).



INPUT OUTPUT

16.7 Vertex Coloring

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

Problem description: Color the vertices of V using the minimum number of colors such that i and j have different colors for all $(i, j) \in E$.

Discussion: Vertex coloring arises in many scheduling and clustering applications. Register allocation in compiler optimization is a canonical application of coloring. Each variable in a given program fragment has a range of times during which its value must be kept intact, in particular after it is initialized and before its final use. Any two variables whose life spans intersect cannot be placed in the same register. Construct a graph where each vertex corresponds to a variable, with an edge between any two vertices whose variable life spans intersect. Since none of the variables assigned the same color clash, they all can be assigned to the same register.

No conflicts will occur if each vertex is colored using a distinct color. But computers have a limited number of registers, so we seek a coloring using the fewest colors. The smallest number of colors sufficient to vertex-color a graph is its *chromatic number*.

Several special cases of interest arise in practice:

- *Can I color the graph using only two colors?* – An important special case is testing whether a graph is *bipartite*, meaning it can be colored using only two different colors. Bipartite graphs arise naturally in such applications as mapping workers to possible jobs. Fast, simple algorithms exist for problems

such as matching (see Section 15.6 (page 498)) when restricted to bipartite graphs.

Testing whether a graph is bipartite is easy. Color the first vertex blue, and then do a depth-first search of the graph. Whenever we discover a new, uncolored vertex, color it opposite of its parent, since the same color would cause a clash. The graph cannot be bipartite if we ever find an edge (x, y) where both x and y have been colored identically. Otherwise, the final coloring will be a 2-coloring, constructed in $O(n + m)$ time. An implementation of this algorithm is given in Section 5.7.2 (page 167).

- *Is the graph planar, or are all vertices of low degree?* – The famous four-color theorem states that every planar graph can be vertex colored using at most four distinct colors. Efficient algorithms for finding a four-coloring on planar graphs are known, although it is NP-complete to decide whether a given planar graph is three-colorable.

  There is a very simple algorithm to find a vertex coloring of a planar graph using at most six colors. In any planar graph, there exists a vertex of at most five degree. Delete this vertex and recursively color the graph. This vertex has at most five neighbors, which means that it can always be colored using one of the six colors that does not appear as a neighbor. This works because deleting a vertex from a planar graph leaves a planar graph, meaning that it must also have a low-degree vertex to delete. The same idea can be used to color any graph of maximum degree Δ using $\leq \Delta + 1$ colors in $O(n\Delta)$ time.

- *Is this an edge-coloring problem?* – Certain vertex coloring problems can be modeled as *edge coloring*, where we seek to color the edges of a graph G such that no two edges are colored the same if they have a vertex in common. The payoff is that there is an efficient algorithm that always returns a near-optimal edge coloring. Algorithms for edge coloring are the focus of Section 16.8 (page 548).

Computing the chromatic number of a graph is NP-complete, so if you need an exact solution you must resort to backtracking, which can be surprisingly effective in coloring certain random graphs. It remains hard to compute a good approximation to the optimal coloring, so expect no guarantees.

Incremental methods are the heuristic of choice for vertex coloring. As in the previously-mentioned algorithm for planar graphs, vertices are colored sequentially, with the colors chosen in response to colors already assigned in the vertex's neighborhood. These methods vary in how the next vertex is selected and how it is assigned a color. Experience suggests inserting the vertices in nonincreasing order of degree, since high-degree vertices have more color constraints and so are most likely to require an additional color if inserted late. Brèlaz's heuristic [Brè79] dynamically selected the uncolored vertex of highest *color degree* (i.e. , adjacent to the most different colors), and colors it with the lowest-numbered unused color.

Incremental methods can be further improved by using *color interchange*. Taking a properly colored graph and exchanging two of the colors (painting the red vertices blue and the blue vertices red) leaves a proper vertex coloring. Now suppose we take a properly colored graph and delete all but the red and blue vertices. We can repaint one or more of the resulting connected components, again leaving a proper coloring. After such a recoloring, some vertex v previously adjacent to both red and blue vertices might now be only adjacent to blue vertices, thus freeing v to be colored red.

Color interchange is a win in terms of producing better colorings, at a cost of increased time and implementation complexity. Implementations are described next. Simulated annealing algorithms that incorporate color interchange to move from state to state are likely to be even more effective.

Implementations: Graph coloring has been blessed with two useful Web resources. Culberson's graph coloring page, *http://web.cs.ualberta.ca/~joe/Coloring/*, provides an extensive bibliography and programs to generate and solve hard graph coloring instances. Trick's page, *http://mat.gsia.cmu.edu/COLOR/color.html*, provides a nice overview of graph coloring applications, an annotated bibliography, and a collection of over 70 graph-coloring instances arising in applications such as register allocation and printed circuit board testing. Both contain a C language implementation of the DSATUR coloring algorithm.

Programs for the closely related problems of finding cliques and vertex coloring graphs were sought for at the Second DIMACS Implementation Challenge [JT96], held in October 1993. Programs and data from the challenge are available by anonymous FTP from *dimacs.rutgers.edu*. Source codes are available under *pub/challenge/graph* and test data under *pub/djs*, including a simple "semi-exhaustive greedy" scheme used in the graph-coloring algorithm XRLF [JAMS91].

GraphCol (*http://code.google.com/p/graphcol/*) contains tabu search and simulated annealing heuristics for constructing colorings in C.

The C++ Boost Graph Library [SLL02] (*http://www.boost.org/libs/graph/doc*) contains an implementation of greedy incremental vertex coloring heuristics. GOBLIN (*http://www.math.uni-augsburg.de/~fremuth/goblin.html*) implements a branch-and-bound algorithm for vertex coloring.

Pascal implementations of backtracking algorithms for vertex coloring and several heuristics, including largest-first and smallest-last incremental orderings and color interchange, appear in [SDK83]. See Section 19.1.10 (page 662).

Nijenhuis and Wilf [NW78] provide an efficient Fortran implementation of chromatic polynomials and vertex coloring by backtracking. See Section 19.1.10 (page 661).

Combinatorica [PS03] provides Mathematica implementations of bipartite graph testing, heuristic colorings, chromatic polynomials, and vertex coloring by backtracking. See Section 19.1.9 (page 661).

Notes: An old but excellent source on vertex coloring heuristics is Syslo, Deo, and Kowalik [SDK83], which includes experimental results. Classical heuristics for vertex coloring include [Brè79, MMI72, Tur88]; see [GH06, HDD03] for more recent results.

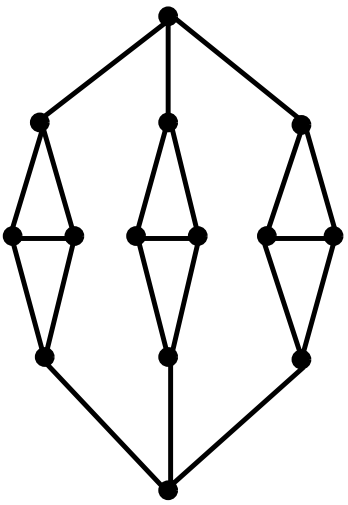
Wilf [Wil84] proved that backtracking to test whether a random graph has chromatic number k runs in *constant time*, dependent on k but independent of n. This is not as interesting as it sounds, because only a vanishingly small fraction of such graphs are indeed k-colorable. A number of provably efficient (but still exponential) algorithms for vertex coloring are known. See [Woe03] for a survey.

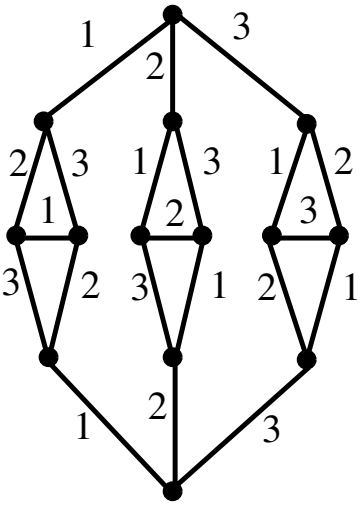
Paschos [Pas03] reviews what is known about provably good approximation algorithms for vertex coloring. On one hand, it is provably hard to approximate within a polynomial factor [BGS95]. On the other hand, there are heuristics that offer some nontrivial guarantees in terms of various parameters, such as Wigderson's [Wig83] factor of $n^{1-1/(\chi(G)-1)}$ approximation algorithm, where $\chi(G)$ is the chromatic number of G.

Brook's theorem states that the chromatic number $\chi(G) \leq \Delta(G) + 1$, where $\Delta(G)$ is the maximum degree of a vertex of G. Equality holds only for odd-length cycles (which have chromatic number 3) and complete graphs.

The most famous problem in the history of graph theory is the four-color problem, first posed in 1852 and finally settled in 1976 by Appel and Haken using a proof involving extensive computation. Any planar graph can be five-colored using a variation of the color interchange heuristic. Despite the four-color theorem, it is NP-complete to test whether a particular planar graph requires four colors or if three suffice. See [SK86] for an exposition on the history of the four-color problem and the proof. An efficient algorithm to four-color a graph is presented in [RSST96].

Related Problems: Independent set (see page 528), edge coloring (see page 548).





INPUT OUTPUT

16.8 Edge Coloring

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

Problem description: What is the smallest set of colors needed to color the edges of G such that no two same-color edges share a common vertex?

Discussion: The edge coloring of graphs arises in scheduling applications, typically associated with minimizing the number of noninterfering rounds needed to complete a given set of tasks. For example, consider a situation where we must schedule a given set of two-person interviews, where each interview takes one hour. All meetings could be scheduled to occur at distinct times to avoid conflicts, but it is less wasteful to schedule nonconflicting events simultaneously. We construct a graph whose vertices are people and whose edges represent the pairs of people who need to meet. An edge coloring of this graph defines the schedule. The color classes represent the different time periods in the schedule, with all meetings of the same color happening simultaneously.

The National Football League solves such an edge-coloring problem each season to make up its schedule. Each team's opponents are determined by the records of the previous season. Assigning the opponents to weeks of the season is an edge-coloring problem, complicated by extra constraints of spacing out rematches and making sure that there is a good game every Monday night.

The minimum number of colors needed to edge color a graph is called its *edge-chromatic number* by some and its *chromatic index* by others. Note that an even-length cycle can be edge-colored with 2 colors, while odd-length cycles have an edge-chromatic number of 3.

Edge coloring has a better (if less famous) theorem associated with it than vertex coloring. Vizing's theorem states that any graph with a maximum vertex degree of Δ can be edge colored using at most $\Delta + 1$ colors. To put this in perspective, note that *any* edge coloring must have at least Δ colors, since all the edges incident on any vertex must be distinct colors.

The proof of Vizing's theorem is constructive, meaning it can be turned into an $O(nm\Delta)$ algorithm to find an edge-coloring with $\Delta + 1$ colors. Since deciding whether we can get away using one less color than this is NP-complete, it hardly seems worth the effort to worry about it. An implementation of Vizing's theorem is described below.

Any edge-coloring problem on G can be converted to the problem of finding a vertex coloring on the *line graph* $L(G)$, which has a vertex of $L(G)$ for each edge of G and an edge of $L(G)$ if and only if the two edges of G share a common vertex. Line graphs can be constructed in time linear to their size, and any vertex-coloring code can be employed to color them. That said, it is disappointing to go the vertex coloring route. Vizing's theorem is our reward for the extra thought needed to see that we have an edge-coloring problem.

Implementations: Yan Dong produced an implementation of Vizing's theorem in C++ as a course project for my algorithms course while a student at Stony Brook. It can be found on the algorithm repository site at *http://www.cs.sunysb.edu/~algorith*.

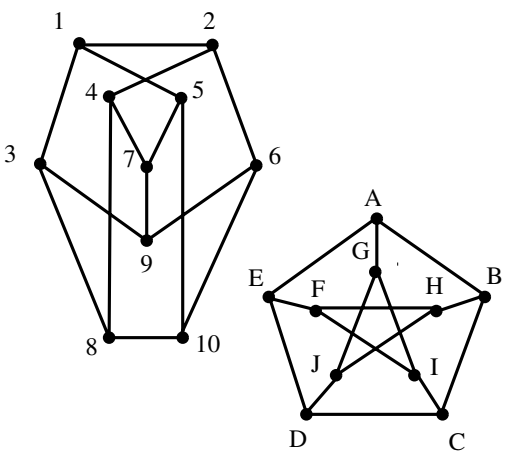
GOBLIN (*http://www.math.uni-augsburg.de/~fremuth/goblin.html*) implements a branch-and-bound algorithm for edge coloring.

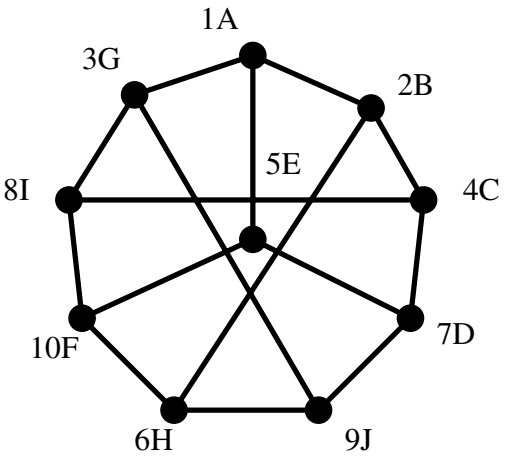
See Section 16.7 (page 544) for a larger collection of vertex-coloring codes and heuristics, which can be applied to the line graph of your target graph. Combinatorica [PS03] provides Mathematica implementations of edge coloring in this fashion, via the line graph transformation and vertex coloring routines. See Section 19.1.9 (page 661) for more information on Combinatorica.

Notes: Graph-theoretic results on edge coloring are surveyed in [FW77, GT94]. Vizing [Viz64] and Gupta [Gup66] independently proved that any graph can be edge colored using at most $\Delta + 1$ colors. Misra and Gries give a simple constructive proof of this result [MG92]. Despite these tight bounds, it is NP-complete to compute the edge-chromatic number [Hol81]. Bipartite graphs can be edge-colored in polynomial time [Sch98].

Whitney, in introducing line graphs [Whi32], showed that with the exception of K_3 and $K_{1,3}$, any two connected graphs with isomorphic line graphs are isomorphic. It is an interesting exercise to show that the line graph of an Eulerian graph is both Eulerian and Hamiltonian, while the line graph of a Hamiltonian graph is always Hamiltonian.

Related Problems: Vertex coloring (see page 544), scheduling (see page 468).





INPUT OUTPUT

16.9 Graph Isomorphism

Input description: Two graphs, G and H.

Problem description: Find a (or all) mapping f from the vertices of G to the vertices of H such that G and H are identical; i.e. , (x, y) is an edge of G iff $(f(x), f(y))$ is an edge of H.

Discussion: Isomorphism is the problem of testing whether two graphs are really the same. Suppose we are given a collection of graphs and must perform some operation on each of them. If we can identify which of the graphs are duplicates, we can discard copies to avoid redundant work.

Certain pattern recognition problems are readily mapped to graph or subgraph isomorphism detection. The structure of chemical compounds are naturally described by labeled graphs, with each atom represented by a vertex. Identifying all molecules in a structure database containing a particular functional group is an instance of subgraph isomorphism testing.

We need some terminology to settle what is meant when we say two graphs are the same. Two labeled graphs $G = (V_g, E_g)$ and $H = (V_h, E_h)$ are *identical* when $(x, y) \in E_g$ iff $(x, y) \in E_h$. The isomorphism problem consists of finding a mapping from the vertices of G to H such that they are identical. Such a mapping is called an *isomorphism*; the problem of finding the mapping is sometimes called *graph matching*.

Identifying symmetries is another important application of graph isomorphism. A mapping of a graph to itself is called an *automorphism*, and the collection of