The Stanford GraphBase: A Platform for Combinatorial Computing

Donald E. Knuth, Stanford University

A highly portable collection of programs and data is now available to researchers who study combinatorial algorithms and data structures. All files are in the public domain and usable with only one restriction: They must not be changed! A "change file" mechanism allows local customization while the master files stay intact.

The programs are intended to be interesting in themselves as examples of "literate programming." Thus, the Stanford GraphBase can also be regarded as a collection of approximately 30 essays for programmers to enjoy reading, whether or not they are doing algorithmic research. The programs are written in CWEB, a combination of TEX and C that is easy to use by anyone who knows those languages and easy to read by anyone familiar with the rudiments of C. (The CWEB system is itself portable and in the public domain.)

Four program modules constitute the *kernel* of the GraphBase:

GB_FLIP is a portable random number generator;

GB_GRAPH defines standard data structures for graphs and includes routines for storage allocation;

GB_IO reads data files and makes sure they are uncorrupted;

GB_SORT is a portable sorting routine for 32-bit keys in linked lists of nodes.

All of the other programs rely on GB_GRAPH and some subset of the other three parts of the kernel.

A dozen or so *generator modules* construct graphs that are of special interest in algorithmic studies. For example, GB_BASIC contains 12 subroutines to produce standard graphs, such as the graphs of queen moves on *d*-dimensional rectangular boards with "wrap-around" on selected coordinates. Another generator module, GB_RAND, produces several varieties of random graphs.

Each graph has a unique identifier that allows researchers all over the world to work with exactly the same graphs, even when those graphs are "random." Repeatable experiments and standard benchmarks will therefore be possible and widely available.

Most of the generator modules make use of *data sets*, which the author has been collecting for 20 years in an attempt to provide interesting and instructive examples for some forthcoming books on combinatorial algorithms (*The Art of Computer Programming*, Volumes 4A, 4B, and 4C). For example, one of the data sets is `words.dat`, a collection of 5-letter words of English that the author believes is "complete" from his own reading experience. Each word is accompanied by frequency counts in various standard corpuses of text, so that the most common terms can be singled out if desired. GB_WORDS makes a subset of words into a graph by saying that two words are adjacent when they agree in 4 out of 5 positions. Thus, we can get from `words` to `graph` in seven steps:

`words, wolds, golds, goads, grads, grade, grape, graph.`

This is in fact the shortest such chain obtainable from `words.dat`.

A dozen or so *demonstration modules* are also provided, as illustrations of how the generated graphs can be used. For example, the LADDERS module is an interactive program to construct chains of 5-letter words like the one just exhibited, using arbitrary subsets of the data. If we insist on restricting our choices to the 2000 most common words, instead of using the entire collection of about 5700, the shortest path from words to graph turns out to have length 20:

```
words, lords, loads, leads, leaps, leapt, least,
lease, cease, chase, chose, chore, shore, shone,
phone, prone, prove, grove, grave, grape, graph.
```

Several variations on this theme have also been implemented. If we consider the distance between adjacent words to be alphabetic distance, for example, the shortest path from words to graph turns out to be

words (3) woods (16) goods (14) goads (3) grads (14) grade (12) grape (3) graph,

total length 65.

The LADDERS module makes use of another GraphBase module called GB_DIJK, which carries out Dijkstra's algorithm for shortest paths and allows the user to plug in arbitrary implementations of priority queues so that the performance of different queuing methods can be compared.

The graphs produced by GB_WORDS are undirected. Other generator modules, like GB_ROGET, produce directed graphs. Roget's famous *Thesaurus* of 1882 classified all concepts into 1022 categories, which we can call the vertices of a graph; an arc goes from u to v when category u contains a cross reference to category v in Roget's book. A demonstration module called ROGET_COMPONENTS determines the strong components of graphs generated by GB_ROGET. This program is an exposition of Tarjan's algorithm for strong components and topological sorting of directed graphs.

Similarly, world literature leads to further interesting families of undirected graphs via the GB_BOOKS module. Five data sets anna.dat, david.dat, homer.dat, huck.dat, and jean.dat give information about *Anna Karenina*, *David Copperfield*, *The Iliad*, *Huckleberry Finn*, and *Les Misérables*. As you might expect, the characters of each work become the vertices of a graph. Two vertices are adjacent if the corresponding characters encounter each other, in selected chapters of the book. A demonstration program called BOOK_COMPONENTS finds the blocks (i.e., biconnected components) of these graphs using the elegant algorithm of Hopcroft and Tarjan.

Another module, GB_GAMES, generates graphs based on college football scores. All the games from the 1990 season between America's leading 120 teams are recorded in games.dat; this data leads to "cliquey" graphs, because most of the teams belong to leagues and they play every other team in their league. The overall graph is, however, connected. A demonstration module called FOOTBALL finds long chains of scores, to prove for instance that Stanford might have trounced Harvard by more than 2000 points if the two teams had met—because Stanford beat Notre Dame by 5, and Notre Dame beat Air Force by 30, and Air Force beat Hawaii by 24, and ... , and Yale beat Harvard by 15. (Conversely, a similar "proof" also ranks Harvard over Stanford by more than 2000 points.) No good algorithm is known for finding the optimum solution to problems like this, so the data provides an opportunity for researchers to exhibit better and better solutions with better and better techniques as algorithmic progress is made.

The GB_ECON module generates directed graphs based on the flow of money between industries in the US economy. A variety of graphs can be obtained, as the economy can be divided into any number of sectors from 2 to 79 in this model. A demonstration program ECON_ORDER attempts to rank the sectors in order from "suppliers" to "consumers," namely to permute rows and columns of a matrix so as to minimize the sum of entries above the diagonal. A reasonably efficient algorithm for this problem is known, but it is very complicated. Two heuristics are implemented for comparison, one "greedy" and the other "cautious." Greed appears to be victorious, at least in the economic sphere.

The highway mileage between 128 North American cities appears in `miles.dat`, and the GB_MILES module generates a variety of graphs from it. Of special interest is a demonstration module called MILES_SPAN, which computes the minimum spanning trees of graphs output by GB_MILES. Four algorithms for minimum spanning trees are implemented and compared, including some that are theoretically appealing but do not seem to fare so well in practice. An approach to comparison of algorithms called "mem counting" is shown in this demonstration to be an easily implemented machine-independent measure of efficiency that gives a reasonably fair comparison between competing techniques.

A generator module called GB_RAMAN produces "Ramanujan graphs," which are important because of their role as expander graphs, useful for communication. A demonstration module called GIRTH computes the shortest circuit and the diameter of Ramanujan graphs. Notice that some graphs, like those produced by GB_BASIC or GB_RAMAN, have a rigid mathematical structure; others, like those produced by GB_ROGET or GB_MILES, are more "organic" in nature. It is interesting and important to test algorithms on both kinds of graphs, in order to see if there is any significant difference in performance.

A generator module called GB_GATES produces graphs of logic circuits. One such family of graphs is equivalent to a simple RISC chip, a programmable microcomputer with a variable number of registers. Using such a "meta-network" of gates, algorithms for design automation can be tested for a range of varying parameters. A demonstration module TAKE_RISC simulates the execution of the chip on a sample program. Another meta-network of gates will perform parallel multiplication of m-bit numbers by n-bit numbers or by an n-bit constant; the MULTIPLY module demonstrates these circuits.

Planar graphs are generated by GB_PLANE, which includes among other things an implementation of the best currently known algorithm for Delaunay triangulation.

Pixel data can lead to interesting bipartite graphs. Leonardo's *Gioconda* is represented by `lisa.dat`, an array of pixels that is converted into graphs of different kinds by GB_LISA. A demonstration routine ASSIGN_LISA solves the assignment problem by choosing one pixel in each row and in each column so that the total brightness of selected pixels is maximized. Although the assignment problem being solved here has no relevance to art criticism or art appreciation, it does have great pedagogical value, because there is probably no better way to understand the characteristics of a large array of numbers than to perceive the array as an image.

A module called GB_SAVE converts GraphBase graphs to and from an ASCII format that readily interfaces with other systems for graph manipulation.

For further information see *The Stanford GraphBase*, published by ACM Press in 1993. The book could also be called "Fun and games with the Stanford GraphBase," because the demonstration programs are great toys to play with. Indeed, the author firmly believes that the best serious work is also good fun. We needn't apologize if we enjoy doing research.