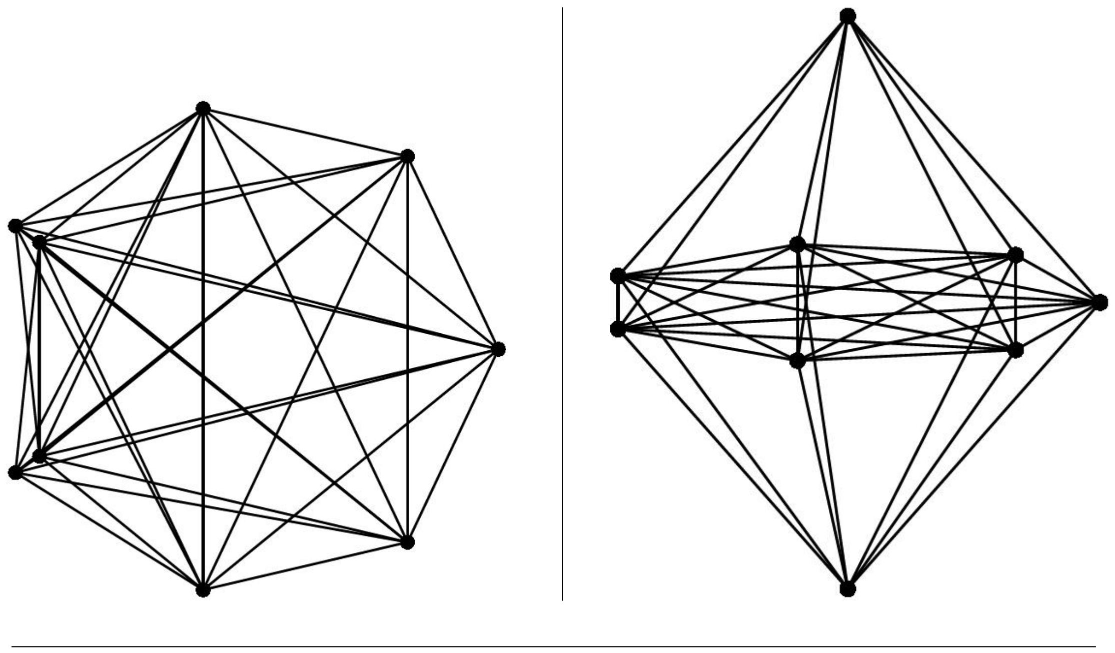

15.10 Drawing Graphs Nicely

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).