Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

AESTHETICS-BASED GRAPH LAYOUT FOR HUMAN CONSUMPTION

M. Coleman

March 1993
CSD-930004

UNIVERSITY OF CALIFORNIA

Los Angeles

Aesthetics-based Graph Layout for Human Consumption

A thesis submitted in partial satisfaction of the

requirements for the degree Master of Science

in Computer Science

by

Michael Karl Coleman

1993

The thesis of Michael Karl Coleman is approved.

David Martin

Andrew Kahng

D. Stott Parker, Committee Chair

University of California, Los Angeles

1993

To those who have

walked

with

me

and made my life

worth

living

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

glow—*v.* To show elation.

First and foremost, I would like to thank my advisor, Stott Parker, for his invaluable support during my stay at UCLA. Stott gave me the room and resources to explore, provoked my curiosity during many helpful discussions, and encouraged me when things looked dark. I am truly in his debt. The rest of my committee, Andrew Kahng and David Martin, also were very helpful and provided many useful comments on this work.

Paul Eggert provided much useful advice and encouragement on this research. My reviewers Junio Hamano, Scott Kalter, and Dorab Patel read and made many valuable comments on the text. Scott, Dorab, and Paul also provided much insight into the deeper significance of the university. John Armato, Kay Bailey, and Karla Harrington proofread various drafts, catching numerous errors and greatly improving the style and language of the thesis.

Dan Greening provided the source code from his version of Lam's simulated annealer ([Lam88]), which was of great use in the early stages of this research. Junio Hamano provided several large (and fascinating) graph data sets which appear in this thesis.

This research benefited from the use of `gnuplot`, written by Thomas Williams and Colin Kelley, and from the Free Software Foundation programs `emacs` and `gcc`, written by Richard Stallman et al. This thesis was prepared using Donald

Knuth's TEX typesetting system with Leslie Lamport's LATEX macros. All of this software is freely available.

I received substantial financial support under a UC MICRO grant funded jointly by Twin Sun, Inc., and the state of California. This funding was greatly appreciated. Some of the research was performed on equipment provided by National Science Foundation grant number IRI-8917907, and on equipment granted by International Business Machines, Inc., under a UC MICRO grant.

Finally, this thesis would have never been completed without the friendship, love, moral support, and forbearance of my friends and extended family: John and Diane Armato, Courtney Chatfield, Guylene Citta, Chris Coleman, David and Dorle Coleman, Steve Graham, Dan Greening, Al Goerner and Nancy Smith, Karla Harrington, Jean Hauser, Mark and Maureen LaRouche, Jennifer Peltz, Deb Trytten, Kim Woods, Jim Wyatt, my fellow Slothbusters (past and present), many others unnamed but not forgotten, and, especially, Monica Markiewicz. My deepest thanks to all of you. :-)

ABSTRACT OF THE THESIS

Aesthetics-based Graph Layout for Human Consumption

by

Michael Karl Coleman

Master of Science in Computer Science

University of California, Los Angeles, 1993

Professor D. Stott Parker, Chair

Automatic graph layout is an important and long-studied problem in computer science. The basic straight-edge graph layout problem is: Given a graph, position the vertices in a way which maximizes some measure of desirability. When graph layout is intended for human consumption, we call this measure of desirability an *aesthetic*. We seek an algorithm which produces graph layouts of high aesthetic quality, and which handles trees, directed acyclic graphs, and general graphs.

Our approach is to model graph layout as a multiobjective optimization problem, where the value of a layout is determined by a user-controlled set of layout aesthetics. We justify this model theoretically, and describe our Aesthetic Graph Layout (AGLO) algorithm and its implementation, the **aglo** library.

The AGLO algorithm combines the power and flexibility of the simulated annealing approach of Davidson and Harel (1989) with the relative speed of the

method of Fruchterman and Reingold (1991), and provides a better theoretical foundation for these methods. In addition, we have developed several new layout aesthetics to support new layout styles. Using these aesthetics, we are able to produce pleasing displays for graphs on which these other methods flounder.

CHAPTER 1

Introduction

lay out—*v.* To knock unconscious.

Automatic graph layout is an important and long-studied problem in computer science. Graphs are a fundamental and effective way of displaying relational information and appear everywhere in computer science, and other disciplines as well. Graph layout can be done by hand, but this is tedious for small graphs and impractical for the large graphs encountered in real tasks. If graph displays are to be widely used, automatic graph layout is required.

In this thesis, we present the Aesthetic Graph Layout (AGLO) algorithm, establish its theoretical foundations, describe the **aglo** library implementation of this algorithm, and present over 70 layout examples to demonstrate its capabilities.

1.1 The Problem

The basic straight-edge graph layout problem is:

> Given a graph, position the vertices in a way which maximizes some measure of desirability.

If the graph layout is intended for human consumption, we will call this measure of desirability an *aesthetic*. Examples of aesthetics include beauty, symmetry, and clarity. In this thesis, we will restrict our attention to the placement of vertices and straight edges in two dimensions. For us, a *layout* (or *placement*) is a positioning of vertices on a bounded plane, called the *tableau*. Since the edges are straight lines, their position is completely determined by the positioning of the vertices.

A good graph layout algorithm will strive for these properties:

- **good theoretical foundation**—It should be clear why the algorithm works and how it can modified to perform different styles of layout.
- **generality**—The algorithm should be able to lay out different kinds of graphs using different styles so that an appropriate style can be used.
- **uniformity**—The algorithm should be able to do layout of different classes of graphs using different aesthetic styles all within the same basic framework. Ad hoc solutions are less useful.
- **malleability**—The user should be able to understand the algorithm and should be able to tune it to do the style of layout desired.
- **competence**—The algorithm should scale up, or at least degrade gracefully for large problems.
- **speed**—The algorithm needs to be sufficiently fast for its intended purpose. For interactive use, a layout should take less than a minute on a Sun SPARC-

station ELC with eight megabytes of memory.

In this project, we consider only the problem of graph layout using aesthetics intended to maximize readability.

1.2 Approaches to Solving the Problem

In previous work, graph layout problems have been attacked using two different broad approaches. In both approaches, one starts with a class of graphs (e.g., tree, directed acyclic graph (DAG), general) that one wishes to lay out and a general or specific idea of what qualities the resulting layout should possess.

Using the *procedural* approach, one tries to come up with a simple algorithm that will lay out the graph in a desirable way. The aesthetics used are chosen by the designer of the algorithm, though the user may be able to adjust some algorithm parameters. The aesthetics are usually chosen so that they do not conflict, to avoid the need for a framework for resolving conflicts. If they do conflict, a simple method of conflict resolution is used, such as ordering the aesthetics and letting the "higher" aesthetic win over the "lower."

The emphasis in this approach is to come up with a fast algorithm, not necessarily one that is ideal or even well-founded theoretically. In terms of the objectives mentioned in the previous section, these algorithms are generally fast and scale up well, but they have an ad hoc character and lack generality, uniformity, malleability, and theoretical foundation. The bulk of previous work in graph layout has

been of this type. Typical examples of this approach include [WS79], [Vau80], [RT81], [Rob87], [GNV88], and [Tri88].

An alternative is the *declarative* approach, where one starts with a fairly specific, though not necessarily operational, set of aesthetics defining the desired appearance of a layout. Often, layout is modeled on some sort of physical force model, and the aesthetics are analogues of features of that model. For example, "springs" or electric charges may be used to model certain aesthetics. Whereas procedural aesthetics can be quite crude, the specific aesthetics used in the declarative approach provide a means of quantifying the quality of a given layout, thus allowing objective comparison of the quality of two layouts. This quantitative nature leads to a solution involving some form of numerical optimization. As a result, this approach more easily admits aesthetics of a conflicting nature, thereby providing a means for articulating aesthetic tradeoffs. It also allows a greater degree of user control over the layout aesthetics; in some cases, the user may devise aesthetics of his own.

The biggest problem with declarative algorithms is speed. The algorithms in this class work harder to produce a good solution; thus, they are typically slower than procedural algorithms, taking minutes and sometimes hours of CPU time on common small workstations. Much less work has been done in this area—the most significant examples are [Ead84], [KK88], [DH89], and [FR91]. With the exception of [DH89], none of this work has attempted to exploit the potential generality and

malleability of this approach, and [FR91] provides the only implementation fast enough for interactive use.

1.3 Overview

In this thesis, we present the Aesthetic Graph Layout (AGLO) algorithm and an implementation, the **aglo** library. AGLO expresses graph layout as the task of optimizing appropriate aesthetic functions, which is performed via multiobjective optimization methods. AGLO takes the declarative approach because of its expressive power, generality, flexibility, and better theoretical foundation but strives to match the speed and competence of the traditional procedural approach. The **aglo** library includes a basic set of layout aesthetics suitable for performing several different common layout styles. The library is intended to be used by client programs that need to do graph layout and is fast enough for practical use in a small workstation environment.

In the next chapter, we discuss layout aesthetics, the theoretical foundation for AGLO in multiobjective optimization and utility theory, and possible practical implementations. In Chapter 3, we cover the **aglo** library, its limitations and potential speedups, and our fundamental set of aesthetics. We present our results in Chapter 4, beginning with an extended example, moving on to a showcase of **aglo** layouts, and concluding with a discussion of the results. Finally, the results are summarized and possible future directions discussed in Chapter 5.

CHAPTER 2

An Aesthetic Approach to Graph Layout

lie—*v.* To be placed.

In this chapter, we describe our Aesthetic Graph Layout (AGLO) method. We begin with a discussion of layout aesthetics, then discuss graph layout as an optimization problem, and finish with a description of several practical solutions to this problem.

2.1 Layout Aesthetics

The Aesthetic Graph Layout method uses aesthetics to do graph layout. We permit virtually any layout aesthetic but argue for aesthetics that seem particularly good based on intuition, past usage, and the principles of visual perception. Furthermore, we express graph layout as a multiobjective optimization problem and explore the ramifications of this formulation.

The foundation of the AGLO model of graph layout is the assumption that the aesthetic merit of a graph layout can be described by a computable function. We call such a function a *layout aesthetic.* A layout aesthetic is a function that takes a graph placement as input and returns a numerical evaluation of that placement,

where smaller values denote better layouts.

An example of a layout aesthetic would be intervertex distance. Intuitively, if the vertices of a graph are displayed too close to one another, the visual result is undesirable—it becomes difficult for a user to look at the display and understand the structure of the graph. So, the intervertex distance aesthetic might consider each pair of vertices, assess a penalty for pairs that are too close, and sum the penalties to determine the overall aesthetic "score" (i.e., the value of the aesthetic function).

In the AGLO model, aesthetic functions can be user-specified and multiple aesthetics combined to give composite aesthetics. Only a few existing layout algorithms take this approach (e.g., [DH89]).

2.1.1 Kinds of Aesthetics

Various kinds of aesthetics are possible. *Syntactic aesthetics*, such as constraints on intervertex distance, control the layout according to basic visual principles and the structure of the graph. *Semantic aesthetics* control the layout based on the meaning (secondary attributes) of graph elements. For example, the constraint

Don't place vertices too close together.

is a syntactic aesthetic, while the constraint

Vertices in a call graph that denote functions in the same module should be placed close together.

is a semantic aesthetic.

Static aesthetics specify layout without consideration of previous or future layouts. *Dynamic aesthetics* come into play when a graph is modified over time. An important dynamic aesthetic is "minimal edit disruption":

Placement of existing vertices and edges should change as little as possible when a change is made to the graph.

2.1.2 A Partial Enumeration of Layout Aesthetics

In this section, we present a brief list of layout aesthetics derived from the literature and common sense, together with comments.[1] This list is intended to be representative rather than exhaustive, as a huge number of reasonable graph layout aesthetics are possible.

- **Do not place vertices too close together.** (e.g., [DH89]) If vertices are placed so that they occlude other vertices, it will be difficult or impossible to see the structure of the graph. Placing them too close makes the layout harder to read. What will be considered "too close" may depend on the size and shape of the vertex's representation and the total number of vertices in the graph.

[1] We provide here a few references to the literature, but virtually every source mentions variations on a number of these aesthetics.

- **Do not place vertices too far apart.** This keeps the edges short (see below) and also keeps disconnected parts of the graph together.

- **Make the edges short.** (e.g., [DH89]) A layout which has many long edges will usually be difficult to read because there will tend to be much edge-crossing in the layout. Also, keeping edges short tends to force clique vertices to be placed near each other. The importance of this aesthetic depends somewhat upon whether straight or curved edges are used, as some edge-crossing may be avoided in the latter case.

- **Do not make the edges too short.** Similar to the first aesthetic above, except that it does not directly affect the distance between vertex pairs not connected by an edge.

- **Make edge lengths uniform.** (e.g., [FR91]) This tends not to work so well in practice, as it usually conflicts with other aesthetics.

- **Minimize edge crossings.** (e.g., [FR91]) Edge crossings make the layout hard to read. Though this aesthetic is seen as important, overemphasizing it with respect to other aesthetics can lead to unattractive layouts.

- **Do not place vertices too close to edges.** (e.g., [DH89]) In particular, allowing vertices to be placed on nonincident edges may produce a visually ambiguous layout.

- **Place vertices within the tableau.** (e.g., [FR91]) The *tableau* is the designated layout area.

- **Try to place vertices near the center of the tableau.**

- **Distribute the vertices evenly on the tableau.** (e.g., [FR91])

- **Minimize the width of the placement.** This is useful when the placement is expanding too far along one axis, as sometimes happens with tree and DAG layouts.

- **Place parent vertices to the left of their children.** This produces a left-rooted tree, for example.

- **Place vertices that are on the same level of a tree on the same vertical (giving them the same x-coordinate).** (e.g., [WS79]) This is a feature of most standard tree layout algorithms.

- **Place left child to the left of parent and right child to the right.** (e.g., [WS79]) This is useful when an ordered binary tree is to be placed.

- **Center parent over children.** (e.g., [WS79])

- **A tree and its mirror image should produce placements that are reflections of one another.** (e.g., [RT81]) This is a nice symmetry property, but it is not clear how important it is in practice.

- **Reflect inherent symmetry.** (e.g. [FR91]) This is a more general version of the above aesthetic.

- **A subtree should be placed the same regardless of where it occurs in the tree.** (e.g., [RT81]) This is probably unimportant, or even confusing, unless multiple isomorphic subtrees appear in a graph.

2.1.3 Advantages and Disadvantages of the Aesthetic Approach

There are several advantages to the layout aesthetic approach. If the aesthetics are fairly simple, the user will be able to understand the workings of this layout method. Furthermore, the user can exert significant control over the behavior of the layout algorithm by adding and deleting aesthetics and adjusting the way they are combined. Additionally, many existing graph layout algorithms can be modeled using this approach.

One drawback of the method is that aesthetic functions need to be polynomially computable, and preferably continuous, if they are to be of practical use in a search for a desired layout. Another drawback is that using these functions to find good layouts is computationally difficult. We turn now to this problem.

2.2 Layout as an Optimization Problem

In this section, we explore the ramifications of using numerical functions to model layout aesthetics. We desire a layout method based on these aesthetic func-

tions that is both theoretically defensible and tractable in practice. We will look at an overall approach, consider some significant details specific to our problem, and then examine practical questions.

2.2.1 Multiobjective Optimization

Given a graph $G = (V, E)$, where $V = \{v_1, v_2, ..., v_n\}$, we can describe a layout of dimensionality D by a vector of vertex positions $(p_1, p_2, ..., p_n)$, where each p_i $(\in \Re^D)$ is the position of v_i. For notational convenience, we flatten the set of vertex position vectors to $X \ni \mathbf{x} = (x_1, x_2, ..., x_m)$, where $m = nD$. We refer to a particular position vector $\mathbf{x}$ as a layout state, or just layout for short.

Given a vector of aesthetic functions $\mathbf{f} = (f_1, f_2, ..., f_k)$, each of which describes desirable features of graph layout, our problem can be stated as[2]

$$\min_{\mathbf{x} \in X} [f_1(\mathbf{x}), f_2(\mathbf{x}), ..., f_k(\mathbf{x})]$$

Thus, we have a multivariate multiobjective optimization problem.

Unfortunately, several of the most obvious aesthetic functions (e.g., total number of edge crossings) are nonconvex.[3] Additionally, most aesthetics embody some

[2]For vector-valued variables $\mathbf{y}, \mathbf{z} \in \Re^n$, $\mathbf{y} < \mathbf{z}$ if and only if $y_i \leq z_i$ for each $i = 1, ..., n$, with $y_i < z_i$ for at least one i.

[3]An aesthetic function f is *convex* iff

$$tf(\mathbf{x}) + (1-t)f(\mathbf{y}) \geq f(t\mathbf{x} + (1-t)\mathbf{y})$$

for all $t \in [0, 1]$.

Consider the edge-crossing minimization aesthetic. Suppose we move a vertex, which may be an endpoint of several edges, along a line to a new position. As the vertex is being moved, the total number of edge-crossings may vary up or down quite arbitrarily. Thus, the total number of edge crossings is not a convex function of the vector of vertex coordinates.

notion of (Euclidean) distance or variance, so they are nonlinear as well. Therefore, we must deal with a nonlinear nonconvex multivariate multiobjective optimization problem.

The individual aesthetic functions cannot generally be simultaneously minimized, because their minima do not coincide, so we will consider the set of noninferior solutions (also known as Pareto-optimal, efficient, or nondominated solutions).

Definition 1 *A layout* $\mathbf{x}^*$ *is* noninferior *if there exists no* $\mathbf{x}$ $(\neq \mathbf{x}^*)$ *such that* $\mathbf{f}(\mathbf{x}) \leq \mathbf{f}(\mathbf{x}^*)$.

Since our problem is nonconvex, we will not generally be able to find (globally) noninferior solutions, but we can find locally noninferior solutions.

Definition 2 *A layout* $\mathbf{x}^*$ *is* locally noninferior *if there exists a* $\delta > 0$ *such that* $\mathbf{x}^*$ *is noninferior in* $N(\mathbf{x}^*, \delta)$, *where* $N(\mathbf{x}, \delta) \stackrel{\text{def}}{=} \{\mathbf{y} | \mathbf{y} \in \Re^m, |\mathbf{x} - \mathbf{y}| < \delta\}$.

Informally, a noninferior layout is one that cannot be improved (by adjusting vertex positions) with respect to any of our aesthetics without simultaneously making it worse with respect to another. Clearly the best (i.e., with respect to our aesthetics) layouts are all locally noninferior, but how do we go about finding them?

2.2.2 Utility Theory

In our model, we assume the existence of a model user who is to be the "consumer" of our layout. We also assume that this user can consistently determine

the relative quality of two layouts (with respect to a given set of aesthetics). That is, we are assuming a preference order $\succeq$ ("is preferred to") on the set of possible layouts. We also assume that $\succeq$ is a weak order (i.e., transitive, reflexive, and complete), which matches our intuition about how rational users compare layouts.[4]

If we further assume that the set of equivalence classes (indifference classes) induced on X by $\succeq$ is countable, we are guaranteed[5] the existence of a *value function* $v : X \rightarrow \Re$ such that

$$\mathbf{x} \succeq \mathbf{y} \Longleftrightarrow v(\mathbf{x}) \geq v(\mathbf{y}).$$

We cannot prove this countability assumption, of course, but we argue that it is reasonable because the set of layouts a human user can perceive, or that can be shown on a bitmap display, is finite (albeit enormous).

The solution to the question above, then, is to find a suitable value function and use it to choose a layout from the noninferior set.

2.2.3 Composition of Aesthetics

In real situations, a number of aesthetics are used together and may conflict. We seek a composite aesthetic value function that will combine the various aesthetic functions' results into a single value. Chankong and Haimes [CH83] list many possible compositions, but suggest choosing the simplest workable compo-

[4]We concede that there may be users whose layout preferences are not transitive. We have no thoughts about how such users might be satisfied.

[5]See Theorem 3.1 in [CH83].

sition from the additive, multiplicative, and quasiadditive forms (additive being simplest). Guided by their criteria, we choose the additive form,

$$f(\mathbf{x}) = \sum_{i=1}^{k} w_i f_i(\mathbf{x})$$

where all w_i's are positive constants.

Once we have determined a value function that corresponds to the user's preference order, we have a computable method for comparing two layouts. Thus, we can use this function to search for good layouts.

We could also use this value function to compare different layouts produced by different algorithms, i.e., to decide which algorithm's placements were better according to the chosen aesthetics. This provides an objective way of evaluating how well a layout algorithm conforms to the style specified by those aesthetics.

Intuitively, in order for additive composition to make sense, we agree to a constant tradeoff between our various aesthetics. If we use $f(\mathbf{x}) = f_1(\mathbf{x}) + f_2(\mathbf{x})$ as our value function, for example, then we are willing to trade a decline of one unit in $f_1(\mathbf{x})$ for an improvement of one unit in $f_2(\mathbf{x})$ (and vice versa). We can extend this argument for three or more component functions and for nonequal weights w_i.

The problem that can arise is that the component aesthetic functions may be of different orders with respect to our unit of value (e.g., quadratic, cubic, trigonometric, exponential, step functions, etc.). As a result the tradeoff discussed above is usually not entirely "fair" and cannot really be made fair by any adjustment of the constant weights w_i. Care needs to be taken to choose aesthetic functions of

comparable strength so that none is washed out by the others. (Currently, we rely on intuition and experimentation.)

In order to make use of the weighted sum, we need to have a means of choosing weights. Currently, we let the user choose the weights to produce layouts that match his preferences. One promising alternative is a method called the surrogate worth tradeoff method ([HH74]). This method offers a means by which weights could be chosen by pairwise comparison of layouts drawn from a set of sample layouts.

2.3 Practical Solutions

There are a number of different methods for solving nonlinear, nonconvex optimization problems of this nature. An important early example of solving layout problems by optimization was Sutherland's SKETCHPAD system ([Sut63]). SKETCHPAD optimized via relaxation, but unfortunately that method is of limited use for nonconvex problems. Instead, we have examined simulated annealing and force-directed placement.

2.3.1 Simulated Annealing

We first tried solving the layout optimization problem using simulated annealing (SA) ([Dav87, Gre89, KGV82]) with Lam's cooling schedule ([Lam88]). Davidson and Harel ([DH89]) also use SA to tackle the layout problem.

Figure 2.1 shows a 3-D layout of a call graph laid out using simulated annealing and rendered with a raytracer.[6] In addition to showing the structure of the call graph, the figure gives a visual representation of timing data produced with the `gprof` profiling utility. The vertex/vertex repulsion, edge length minimization, and parent left aesthetics are used to draw the call graph as a DAG. A special semantic aesthetic is used to pull vertices representing CPU-intensive functions forward while pushing others back. The amount of time spent in each function controls the size and placement of the corresponding vertex, with CPU-intensive functions being represented by larger vertices placed nearer to the viewer. The three-dimensional placement is a little difficult to see when shown in a single 2-D projection, but the main CPU-intensive call path can be clearly seen in the foreground.

Although simulated annealing does work and does produce good solutions, it is prohibitively slow. The small graph in Figure 2.1 took several hours to lay out using simulated annealing. According to the estimated time equation given in [DH89], their SA-based approach would take 89 days to lay out the graph in Figure 4.68 (p. 83), versus our 15 minutes.

The primary reason for this is that simulated annealing does not exploit all of the information known about the structure of the aesthetic layout problem. It is a zero-order optimization method, meaning that it uses only aesthetic function *values*, ignoring aesthetic function *gradients*. Exact gradients of layout aesthetics

[6]The actual graph, composed of gray vertices and white edges, is in the center, a cast (black) shadow is in the lower right, and a faint reflection appears in the center near the lower edge.

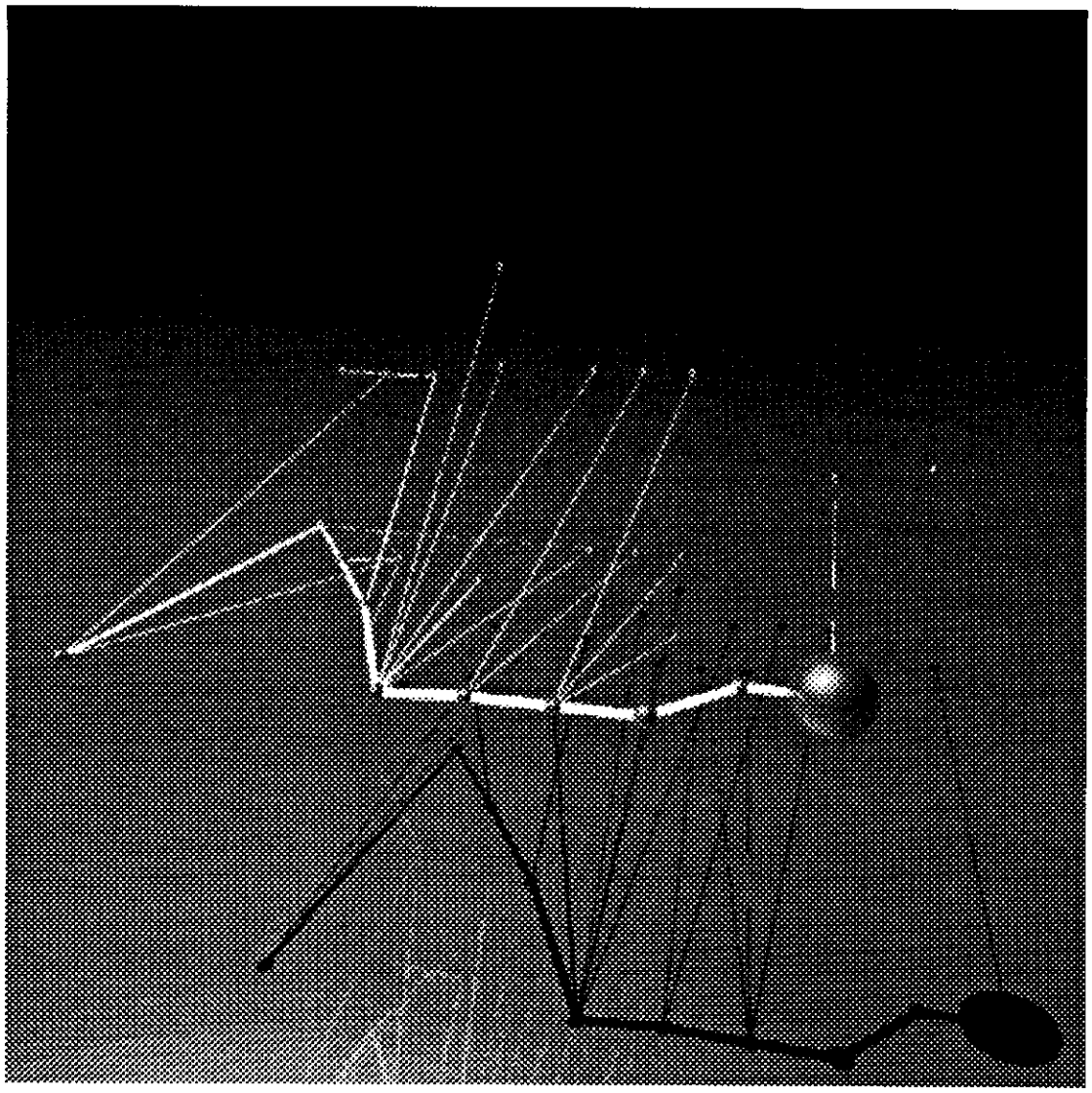

Figure 2.1: A 3-D call graph layout

are not always available, but frequently a *pseudo-gradient*, a heuristic estimate of the direction of a better solution, can be constructed.

2.3.2 Force-directed Placement and AGLO

Currently, we are using a variant of the force-directed placement (**fdp**) algorithm described in [QB79] and applied to graph layout in [FR91]. This algorithm is a simple first-order optimization method. In particular, aesthetic gradient functions[7] are used to decide which direction to go in each step. Our version of the algorithm, which we call Aesthetic Graph Layout, is given in Figure 2.2. (Figure 2.3 shows the **fdp** algorithm, for comparison.)

Both algorithms use a simple geometric cooling schedule identical to that used in the basic simulated annealing algorithm—better cooling schedules should be possible (see Section 5.2).

Our algorithm differs from the version of **fdp** described in [FR91] in several ways. First, we use a weighted sum of a collection of aesthetic functions to guide our layout, whereas **fdp** uses a simple sum of two specific aesthetic functions (vertex/vertex repulsion and edge length minimization).

Secondly, we do temperature clipping of the movement vectors in a way we believe is theoretically superior to that of **fdp**, which uses the temperature to limit

[7] In the rest of the thesis we will refer to both aesthetic gradient functions (which indicate the direction to move to get to a better state) and aesthetic (value) functions (which indicate the quality of the current state) simply as aesthetic functions, since for our purposes they convey the same information (i.e., the answer to the question "What is good?").

1. For the current state $\mathbf{x}$, compute the gradient vector $\Delta_i(\mathbf{x})$ for each individual aesthetic i.

2. Calculate the weighted sum of the individual gradient vectors.

$$\Delta(\mathbf{x}) \leftarrow \sum w_i \Delta_i(\mathbf{x})$$

3. Shorten the sum (movement) vector, if necessary, so that its length is less than the current temperature t.

$$\Delta'(\mathbf{x}) \leftarrow \left[\frac{\Delta(\mathbf{x})}{|\Delta(\mathbf{x})|}\right] \cdot \min(|\Delta(\mathbf{x})|, t)$$

4. Add the resulting sum vector to the current state, producing the new state.

$$\mathbf{x}' \leftarrow \mathbf{x} + \Delta'(\mathbf{x})$$

5. Calculate the new t.

$$t \leftarrow \mathrm{cool}(t)$$

6. Repeat until the desired number of iterations has been completed.

Figure 2.2: The AGLO algorithm

2′. Shorten both gradient vectors (**fdp** uses only two aesthetics), if necessary, so that the length of each is less than the current temperature t.

$$\Delta_1'(\mathbf{x}) \leftarrow \left[\frac{\Delta_1(\mathbf{x})}{|\Delta_1(\mathbf{x})|}\right] \cdot \min(|\Delta_1(\mathbf{x})|, t)$$

$$\Delta_2'(\mathbf{x}) \leftarrow \left[\frac{\Delta_2(\mathbf{x})}{|\Delta_2(\mathbf{x})|}\right] \cdot \min(|\Delta_2(\mathbf{x})|, t)$$

3′. Calculate the sum vector.

$$\Delta'(\mathbf{x}) \leftarrow \Delta_1'(\mathbf{x}) + \Delta_2'(\mathbf{x})$$

Figure 2.3: The **fdp** algorithm, for comparison

the distance moved by each vertex individually. In our algorithm, all position information is represented as one large m-dimensional state vector,[8] and temperature is used to clip the length of *that* vector.

To make the difference concrete, suppose we are doing a 2-D layout of a two-vertex graph, and our aesthetics determine that the two vertices should be moved a distance of 1 and 2 at an iteration where the temperature is 0.5. If we used **fdp**'s clipping method, we would move each vertex 0.5, while AGLO will calculate the move as

$$\begin{aligned}
\Delta'((1,0,2,0)) &= \left[\frac{\Delta(\mathbf{x})}{|\Delta(\mathbf{x})|}\right] \cdot \min(|\Delta(\mathbf{x})|, t) \\
&= \left[\frac{(1,0,2,0)}{|(1,0,2,0)|}\right] \cdot \min(|(1,0,2,0)|, 0.5) \\
&= \left[\frac{(1,0,2,0)}{\sqrt{1^2+2^2}}\right] \cdot 0.5 \\
&\approx (0.224, 0, 0.447, 0),
\end{aligned}$$

moving the vertices 0.224 and 0.447, respectively.[9]

In theoretical terms, our method always moves in the direction of steepest descent (as specified by our aesthetic functions), whereas **fdp** will move along a vector that may differ by an arbitrary amount from this steepest-descent vector. This inefficiency notwithstanding, **fdp** does always proceed in a downhill direction, which seems to be sufficient to keep it from floundering.

[8] Recall that $m = nD$, the product of the number of vertices in the graph and the dimensionality of the layout space.

[9] We picked movement in the positive x-direction for both vertices, but the reader can verify that the result is the same regardless of the direction chosen.

Note carefully that though both algorithms proceed in a downhill direction at each step, it is *not* the case that they actually find a state that is downhill from the previous state. They take a step of a determined length in that direction without concern for whether the new state is an improvement; the new state may actually be worse than the previous one. This "sloppiness" gives both algorithms a simulated annealing quality—they choose downhill steps most of the time, but they will sometimes go uphill, the latter more often when the temperature is higher. Because of the similarity of these algorithms and simulated annealing, it seems plausible that much of the existing research on speeding up annealing may also be applicable to **fdp** and AGLO as well (see Section 5.2).

We also address a point that [FR91] does not mention. If all of the vertices in a graph achieve collinear positions during the layout process, they will become "stuck" on that line, even though the aesthetics indicate that motion off the line in any direction would be an improvement. Our solution is to jitter the vertices slightly by moving a randomly chosen vertex a minuscule distance in an arbitrary direction at each step. Assuming that one of the basic repulsion aesthetics is being used, this will cause the vertex in question to shoot off of the line, destroying the undesired metastable state and allowing useful search to resume.

Finally, and perhaps most significantly, our method is based on a stronger theoretical foundation. Fruchterman and Reingold cast their aesthetics as "forces" and **fdp** as essentially a many-body particle simulation of these forces, but they

provide little rationale for why this simulation might be expected to find aesthetically pleasing graph layouts. The AGLO algorithm, in contrast, is based on a multiobjective optimization problem derived from first principles (see Section 2.2).

2.3.3 Other Methods

There are a number of other numerical optimization algorithms that can find solutions for unconstrained nonconvex nonlinear problems such as ours.[10] Of these, the conjugate-gradient class of algorithms looks the most promising as these algorithms do not require the maintenance of large matrices that most other order-1 algorithms do, nor do they depend on the availability of second derivatives.

We have not yet explored the conjugate-gradient method, but we foresee two potential pitfalls. First, this method assumes the existence of both the objective function and its gradient (our aesthetic value functions and aesthetic gradient functions). It is not clear that we will be able to find differentiable forms for all of our aesthetic value functions and using our "pseudo-gradients" may introduce error and reduce the quality of the results.

The other problem is that the goal of conjugate-gradient optimization is to rapidly find a local minimum near the starting state. It always proceeds downhill and never chooses a new state that is inferior to the current one. When it finds a local minimum, it stops. Thus, the algorithm lacks the annealing quality of the other algorithms we have discussed—it never makes uphill moves to keep from

[10]See [GMW81] for a good practical overview.

getting stuck in a local minimum of poor quality. However, it may be possible to modify the algorithm to sometimes move uphill, or simply to restart each time it finds a local minimum.

CHAPTER 3

Implementation

gloss—*n.* A deceptively good appearance.

3.1 The aglo Library

We have implemented a library, **aglo**, that performs the Aesthetic Graph Layout algorithm. The current version does layout of trees, DAGs, and general graphs and includes seven fundamental aesthetics. The library, written in C, is designed to be linked to a client program that needs to do graph layout (see Figure 3.1). A simple interface is provided so that client programmers can easily implement and use their own aesthetics in addition to, or instead of, the fundamental aesthetics provided with the library. Vector math routines are also provided by **aglo**, further easing the implementation of new aesthetics. (Figure 3.2 shows the actual implementation of the node/node[1] repulsion aesthetic described in Section 3.2.1.)

The library is capable of layout either two or three dimensions,[2] but not all aesthetics make sense in all cases. In particular, several of the fundamental aes-

[1]We also refer to graph vertices as "nodes" here, as that is the term used in the **aglo** library.
[2]The library can actually do layout in any number of dimensions.

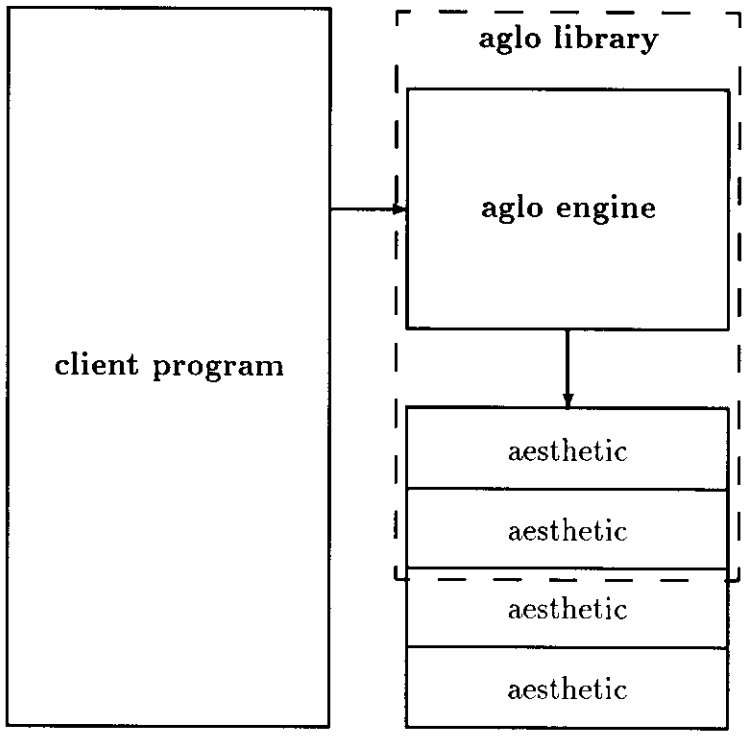

Figure 3.1: Organization of an **aglo** application

```
 1  /* aesthetic 'node repulsion' - makes nodes spread apart */
 2
 3  #include <aesth.h>
 4
 5  declare_aesth(node_repulsion);
 6
 7  define_setup(node_repulsion)
 8  {
 9     return;
10  }
11
12  define_aesth(node_repulsion)
13  {
14     unsigned i, j;
15
16     for (i=1;i<graph->vertices;i++)
17        for (j=0;j<i;j++)
18        {
19           aglo_point delta;
20           aglo_real mag2;
21           aglo_point force_delta;
22
23           aglo_point_sub(delta, state[i], state[j]);
24           aglo_point_mag2(&mag2, delta);
25           mag2 = fmax(mag2, 1e-8); /* avoid div by 0 */
26
27           aglo_point_scalar_mult(force_delta, 1/mag2, delta);
28
29           aglo_point_add(gradient[i], gradient[i], force_delta);
30           aglo_point_sub(gradient[j], gradient[j], force_delta);
31        }
32  }
```

Figure 3.2: Code for the node/node repulsion aesthetic

thetics are defined only in the two-dimensional case. This is the case that we have explored.

To aid in experimenting with and debugging aesthetics, **aglo** has a monitoring feature that, if enabled, will show the progress of the layout. The intermediate layouts are shown on an X Windows display using the freely available `gnuplot` plotting tool. Depending on the arguments used, the monitoring display will be updated either after a fixed number of iterations of the layout algorithm or after a fixed time has elapsed (two seconds by default). The latter option is particularly useful for watching a layout without substantially slowing its computation.

A sample client, **gloss**, is provided that will read a graph from its standard input, do a layout according to the command-line parameters supplied, using the **aglo** library, and print the resulting layout on its standard output. Figure 3.3 shows the usage message for **gloss**. The **gloss** client is particularly useful for experimenting with new aesthetics and combinations of aesthetics.

Figure 3.4 shows the first part of a **gloss** input file. Vertices are numbered consecutively, beginning with 1. Each line describes an edge in the graph.[3] The first number specifies the edge head and the second the edge tail if the graph is considered to be directed, but it is up to each aesthetic to make this distinction.[4]

[3]Singleton vertices can be entered by including each alone on a line. The current aesthetics were not designed for disconnected graphs, so no singleton vertices are shown in any of our figures. Disconnected graphs could be laid out using appropriately designed aesthetics, or their connected subgraphs could be laid out separately using the provided aesthetics.

[4]Of those included with **aglo**, only the "parent left placement" (Section 3.2.6) and "level variance minimization" (Section 3.2.7) aesthetics do so.

```
usage:
        -it <iterations> (1000)
        -bt <beginning temperature> (100.000000)
        -et <ending temperature> (0.001000)
        -m [turn monitor on]
        -mr <monitor update rate (seconds)> (2)
        -s [sleep until ^C at end]
        -knr <node repulsion aesthetic>
        -kmel <minimize edge length aesthetic>
        -kcp <centripetal (repulsion from centroid) aesthetic>
        -kner <node/edge repulsion aesthetic>
        -kmei <minimize edge intersections aesthetic>
        -kmei2 <minimize edge intersections v2 aesthetic>
        -kpl <place parent to left of children aesthetic>
        -kmlv <minimize intralevel variance aesthetic>
```

Figure 3.3: **gloss** usage message

```
1 2
2 3
3 4
4 5
5 6
6 7
7 8
8 1
1 9
2 10
3 11
4 12
.
.
.
```

Figure 3.4: Sample **gloss** input file

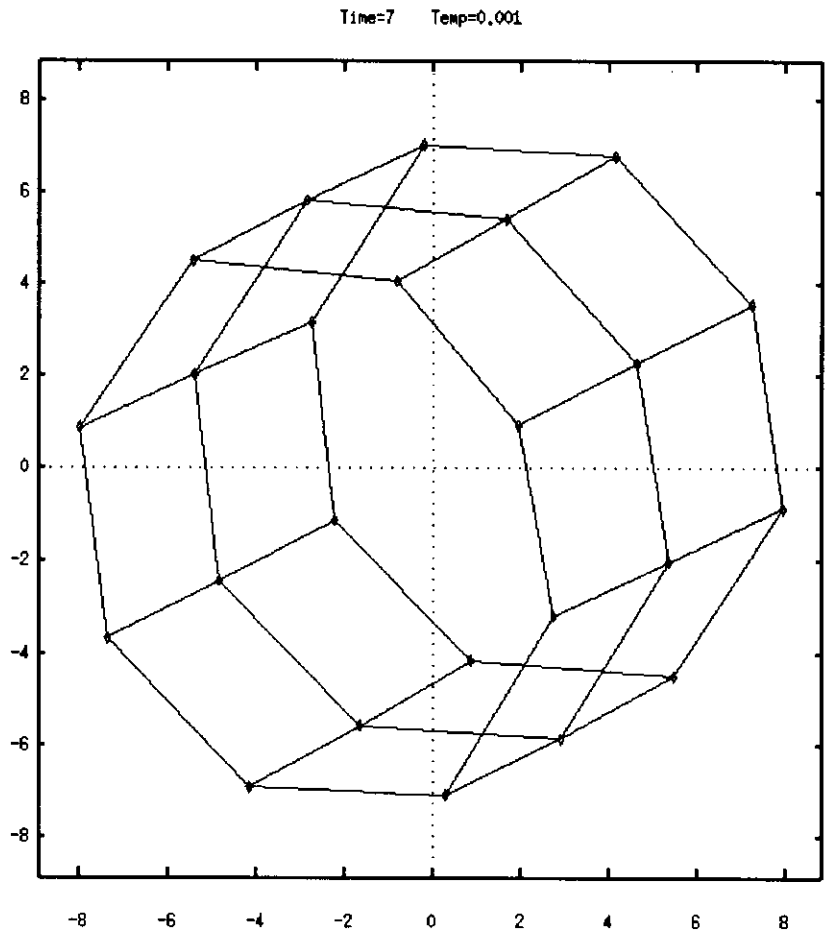

Figure 3.5: Sample monitor display

Figure 3.5 shows a sample monitor screen from a **gloss** run on the input file in Figure 3.4. In addition to the current layout state, the screen indicates the current elapsed time (in seconds) and temperature.

When **gloss** has finished, it prints out the coordinates of the endpoints of each edge in the graph. The coordinates are normalized so that the placement will fit exactly within the unit square with corners $(0,0)$ and $(1,1)$. (The aspect ratio

of the placement is not altered.) The **aglo** library provides the input and output routines used by **gloss**.

3.2 Fundamental Aesthetics

In this section, we will describe each of the fundamental aesthetics currently provided with the **aglo** library. The first two aesthetics are essentially the same as those given in [FR91].

3.2.1 Node/Node Repulsion

This aesthetic causes each node to repel all of the rest. The magnitude of the repulsion vector is $1/d$, where d is the magnitude of the delta vector (i.e., the vector from a node to the repelled node). Since all pairs of vertices are considered, the time complexity of this aesthetic is $\Theta(|V|^2)$.

3.2.2 Edge Length Minimization

This aesthetic causes each edge to be shortened, i.e., it causes the endpoints of an edge to attract each other. The magnitude of the attraction vector is d^2, the magnitude of the delta vector. All edges are considered, so the time complexity of this aesthetic is $\Theta(|E|)$.

3.2.3 Node/Edge Repulsion

This aesthetic causes each node to repel all edges (and vice versa).[5] The magnitude of the repulsion vector is $1/d$, where d is the magnitude of the delta vector, which in this case is the vector from the node to the closest point on the edge.

Note that there are actually two cases here: either the point on the edge closest to the node is one of the endpoints or not. If not, the delta vector is normal to the edge. If it is, we basically have node/node repulsion again.[6]

Since each node/edge pair is considered, the time complexity of this aesthetic is $\Theta(|V||E|)$.

3.2.4 Edge Intersection Minimization

The purpose of this aesthetic is to minimize the number of edge intersections in a layout. This aesthetic presents a special difficulty: given two intersecting edges, the best gradient to an improved layout is nonobvious. We think that this problem is due to this aesthetic being nonconvex (see Section 2.2.1).

After some experimentation, we have arrived at a version that seems to work pretty well in practice: if two edges intersect, we cause their midpoints to repel each other, as in node/node repulsion, except that the magnitude of the repulsion vector is 1 instead of $1/d$.

(We provide a second, stronger version that uses d as the magnitude of the

[5]This aesthetic makes sense in 3-D, but the version in the library handles only the 2-D case.

[6]In our current implementation, we apply no force in this latter case, assuming that co-application of the node/node repulsion aesthetic will make this redundant.

repulsion vector, but is otherwise identical. Neither version clearly dominates the other in terms of layout results, so we include both.)

Since all edge pairs are considered, the time complexity of this aesthetic is $\Theta(|E|^2)$.

In three dimensions, edge intersection *per se* is not a useful concept, so our version only handles the 2-D case. For the 3-D case, we would probably want to generalize this aesthetic to do some form of edge/edge repulsion.

3.2.5 Centripetal Repulsion

The purpose of this aesthetic is to repel all vertices from the centroid[7] of the layout. The position of the centroid is calculated, and each node is repelled from this point as in node/node repulsion. The magnitude of the repulsion vector is $1/d$, where d is the magnitude of the delta vector (i.e., the vector from the node to the centroid). Since each node is considered, the time complexity of this aesthetic is $\Theta(|V|)$.

3.2.6 Parent Left Placement

These last two aesthetics are for tree- and DAG-style layout. Unlike the previous aesthetics, these two treat edges as directed by distinguishing between parents and children.

The purpose of this aesthetic is to cause the parent (or parents, in a DAG) of

[7]The *centroid* vector $\overline{\mathbf{x}}$ is the average of all vertex position vectors $\mathbf{x}_i$.

a node to be placed to its left. (A buffer space is used so that the parent is not placed directly beside the child.) If the parent is not already positioned to the left of the buffer space, the parent and child repel each other along the delta vector, which is parallel to the x-axis. The magnitude of the repulsion vector is d^2, where d is the magnitude of the delta vector. Since each edge is considered, the time complexity of this aesthetic is $\Theta(|E|)$.

3.2.7 Level Variance Minimization

The purpose of this aesthetic is to place all vertices on the same level of the graph near the same x-position. Each node is attracted to the average x-coordinate of all the vertices on the same level. The magnitude of the attraction vector is d^2, where d is the magnitude of the delta vector. Since each node is considered, the time complexity of this aesthetic is $\Theta(|V|)$.

3.3 Limitations

The current implementation of the **aglo** library disallows self-edges and multiple edges between a given pair of vertices (including directed 2-cycles). The aesthetics are designed for wholly connected graphs—since there are no aesthetics to hold disconnected subgraphs together, they will tend to fly far apart during layout, producing an unattractive result.

Currently the input graph has a hard limit of 500 vertices, but this limit may

be trivially increased changing a constant and recompiling **aglo**.

3.4 Potential Speedups

Aside from tuning the current library code, there are a couple of avenues that we expect would speed up the **aglo** layout procedure.

3.4.1 Adaptive Cooling

The current implementation handles cooling quite naively: a beginning and ending temperature and an iteration count are specified a priori, and simple geometric cooling is done based on these parameters. The default values used by **gloss** are conservative so as to perform well on most inputs, but layout can be done at least an order of magnitude faster with a good choice of parameters.[8] To search for better parameters, first lower the beginning temperature, then raise the ending temperature, and finally decrease the iteration count, in each case stopping just before layout quality is compromised.

Probably the easiest improvement would be to add a test to check for a "frozen" state. If the layout has stopped changing in any substantial way through a small number of iterations, we can guess that a local minimum has been reached and nothing will be gained by further computation.

The choice of the overall cooling schedule is a much harder problem. The

[8] For example, a layout of the graph in Figure 4.5 (of similar quality) can be done in under three seconds using the cooling parameters `-bt 12 -et 0.05 -it 80`, as opposed to approximately 30 seconds using the default cooling parameters.

analogous problem for the simulated annealing algorithm has been studied (e.g., [Lam88]), but it is not obvious that results from this work can be applied to our problem.

3.4.2 Distal-Force Optimization

Our placement algorithm bears a computational resemblance to the many-body simulation problem. One significant algorithmic device for the many-body problem involves approximating the forces applied to a node by "distant" vertices as a single force. This optimization is described in [Gre88] and, under the best circumstances, will produce an $\Theta(|V|)$ speedup of the algorithm.

This optimization is used by [FR91] in their implementation. They are aided by the fact that they use only two aesthetics, and these aesthetics are quite similar to the forces used in the many-body problem.

It seems plausible that we could also make use of this optimization, but we will have a harder time of it, because we have a larger collection of aesthetics and we would like users of the **aglo** library to be able to extend this collection with ease. Furthermore, some aesthetics may not be similar to many-body forces and, thus, may not fit the optimization framework.

3.4.3 Other Order-1 Nonlinear Optimization Algorithms

Although the aesthetics we are using are not necessarily true analytic derivatives (gradients), it is plausible that we may successfully apply other known order-1

nonlinear optimization algorithms to our aesthetic layout problem. The conjugate-gradient method (described in [GMW81]) in particular looks promising. (see Section 2.3.3.)

CHAPTER 4

Results

aglossia—*n.* [<Gk: want of eloquence]

In this chapter, we demonstrate **aglo** on several dozen input graphs. Many of the graphs are drawn from previous graph layout papers (including [FR91], [DH89], [KK88], [KK89], and [Ead84]). Depending on the aesthetics and weights chosen, we can produce layouts of tree, DAG, or general graph form.

All of the layouts shown were produced with **gloss**, *using the* **aglo** *library, unless the caption indicates that they are proposed placements.* The source of each graph, including the original figure number, is indicated as appropriate.

All layouts were produced using the default choices for iteration count (1000 iterations) and beginning and ending temperatures, unless noted. These choices will produce good layouts most of the time, though an order-of-magnitude improvement in layout time can often be produced if the parameters are tuned.

The caption of each figure indicates the total CPU time (in seconds) taken to produce the layout on a Sun SPARCstation ELC.[1] All code was compiled using version 2.2.2 of the GNU C compiler (`gcc`) with the `-O2` option.

[1]Specifically, the reported time is the sum of user and system CPU time, to the nearest tenth of a second, as reported by `/bin/time`.

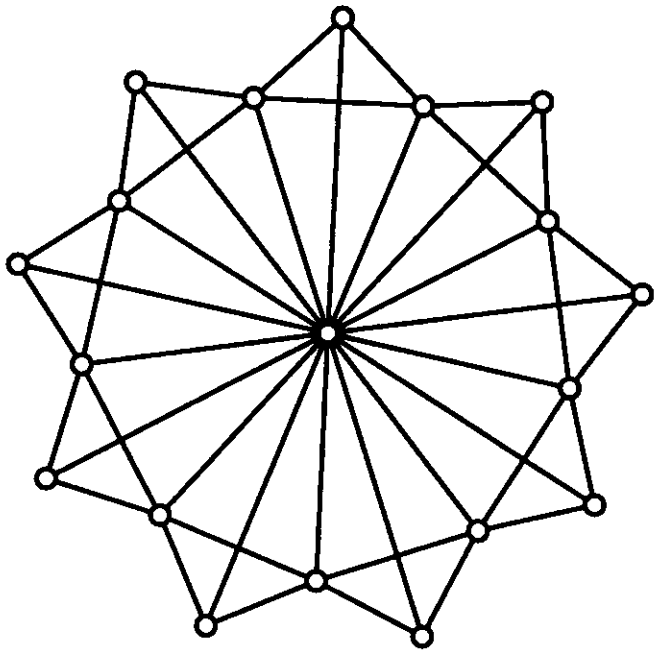

Figure 4.1: Figure 1 from Davidson and Harel (2.8s)

We begin with an extended example, then show examples of the layout of general graphs, DAGs, trees, and large-scale graphs, and conclude with a discussion of these results.

4.1 An Extended Example

In this example, we see how the style of a layout can be modified by changing the aesthetics and their weights. In figure 4.1, the graph given in Figure 1 of Davidson and Harel ([DH89]) is laid out using the command:[2]

```
gloss -knr 1 -kmel 1 -bt 25
```

[2]We are tuning the layout in this case by adjusting the starting temperature (`-bt 25`).

argument	*aesthetic*
`-knr`	node/node repulsion
`-kmel`	edge length minimization
`-kcp`	centripetal repulsion
`-kner`	node/edge repulsion
`-kmei`	edge intersection minimization
`-kmei2`	edge intersection minimization (v.2)
`-kpl`	parent left
`-kmlv`	level variance minimization

Table 4.1: **gloss** aesthetic arguments

(Interpretation of **gloss** aesthetic parameters is summarized in Table 4.1.) This style is used by Fruchterman and Reingold ([FR91]) for all of their two-dimensional layouts, and the resulting layout is similar to theirs.

But now suppose we wish the layout style to be more like that of [DH89]. They render this graph differently—similar to Figure 4.5. What do we do?

The idea is to ask what is wrong with the present layout. What undesired aesthetic quality is present? Then we experiment with the set of existing aesthetics or invent and implement new ones.

In this case, looking at Figure 4.1, an obvious problem is that there are edge crossings where we wish there were none. Specifically, the outer "point" vertices could have been placed inside instead. In order to remedy this problem, we try this command:

```
gloss -knr 1 -kmel 1 -kmei 1 -bt 25
```

producing the layout in Figure 4.2.

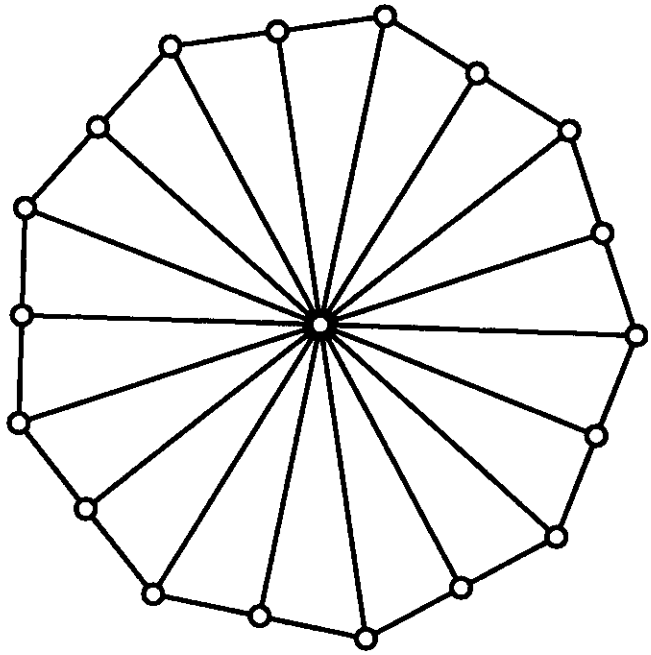

Figure 4.2: Figure 1 from Davidson and Harel, variant 2 (13.4s)

This solves the problem, since the "point" vertices are no longer sticking out, but now we have a new problem: these "point" vertices have been placed on or very near nonincident edges, obscuring edges in the graph and making it aesthetically displeasing. In this case, the problem is quite obvious: we want to be sure that vertices are not placed too close to nonincident edges. The node/edge repulsion aesthetic does just that with

```
gloss -knr 1 -kmel 1 -kmei 1 -kner 1 -bt 25
```

giving us Figure 4.3.

Some of the "point" vertices are inside, where we want them, and others have been placed outside. In fact, if we run **aglo** several times with these same parame-

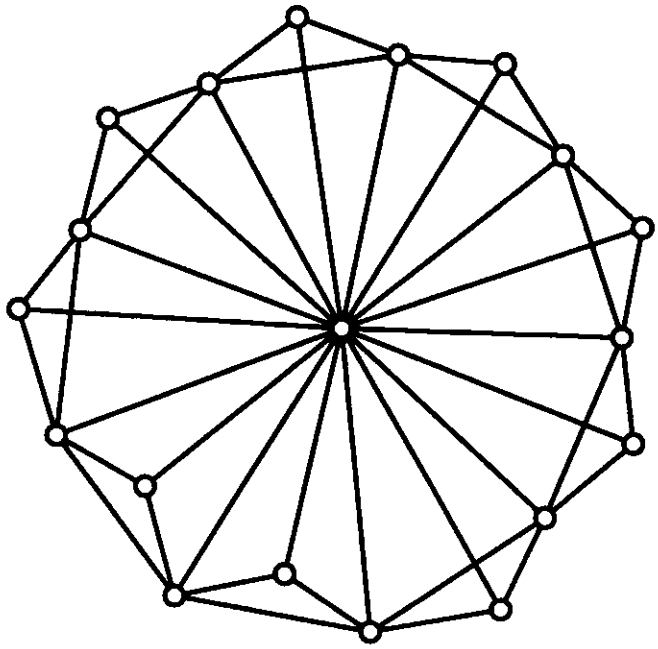

Figure 4.3: Figure 1 from Davidson and Harel, variant 3 (29.3s)

ters, we will get different inside/outside pattern; this is an unstable configuration. We will increase the weight of edge crossing minimization in the hopes that this will push the layout into the stable configuration we desire:

```
gloss -knr 1 -kmel 1 -kmei 10 -kner 1 -bt 25
```

giving us Figure 4.4, the layout we were seeking.

We get a virtually identical layout by replacing the node/node repulsion aesthetic with the centripetal aesthetic thus:

```
gloss -kcp 1 -kmel 1 -kmei 10 -kner 1 -bt 25
```

leading to Figure 4.5.

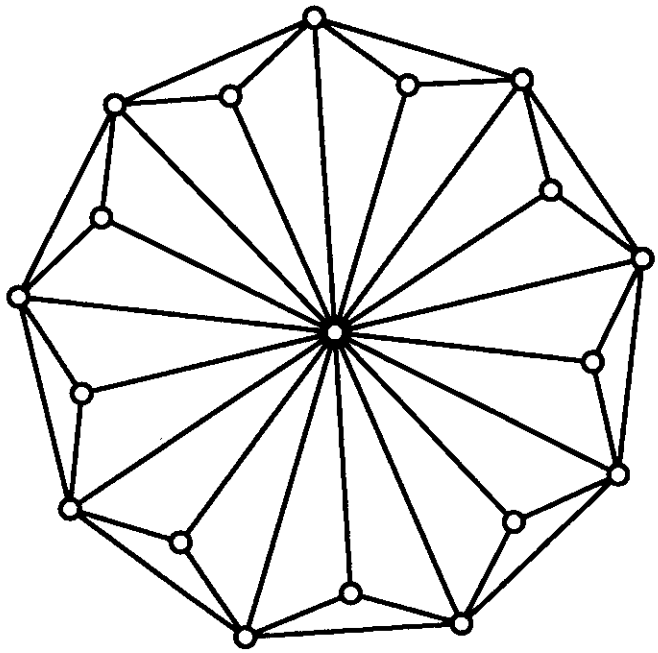

Figure 4.4: Figure 1 from Davidson and Harel, variant 4 (29.1s)

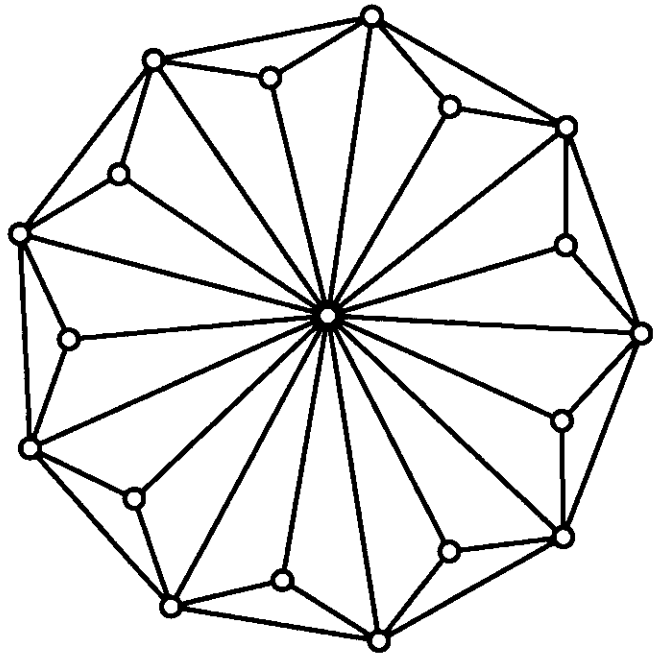

Figure 4.5: Figure 1 from Davidson and Harel, variant 5 (27.3s)

Though we can gain more control by using more aesthetics, these examples also demonstrate that we pay a penalty in added computation time for that control. It is interesting to see that the centripetal aesthetic can sometimes work as a substitute for node repulsion, since the former is $O(|N|)$ faster than the latter in our implementation.

Thus, by choosing different aesthetics and varying their weights, we can exert control over the style of layout. This example demonstrates malleability, competence, and speed of the AGLO algorithm.

4.2 A Graph Layout Gallery

We now present a large collection of **aglo** layouts. Many of the graphs are drawn from other papers for the sake of comparison; others are included to show off the abilities of **aglo**. This collection of examples, taken as a whole, shows off the generality and uniformity of the AGLO approach.

4.2.1 General Graphs

In this section, we show how **aglo** handles general, undirected graphs. Many of the graphs are examples that appear in [FR91] and [DH89], the existing work most similar to ours. We use these graphs to compare and contrast **aglo** with the methods proposed in those papers and argue that the AGLO algorithm is in general superior to those methods.

Unless otherwise mentioned, all figures in this section were created with the command:

```
gloss -knr 1 -kmel 1
```

which is essentially the style of [FR91].

Figures 4.6 through 4.11 show the layout of several simple, symmetric graphs. We observe that our placements are similar to those produced in [FR91] and [KK89]. Figure 4.6 has an unnecessary edge crossing that we can eliminate by including the edge crossing minimization aesthetic, as in Figure 4.7.

Figure 4.12 is drawn as in [DH89] and [FR91] using the command

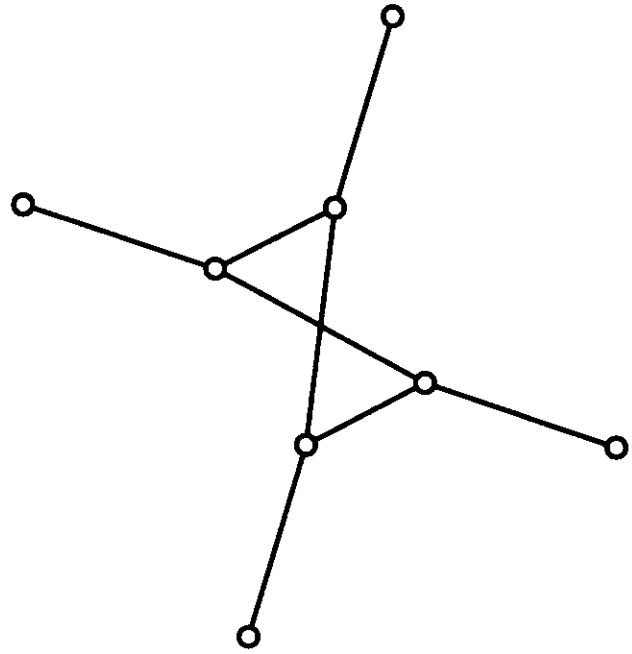

Figure 4.6: Figure 6(a) from Kamada and Kawai (0.5s)

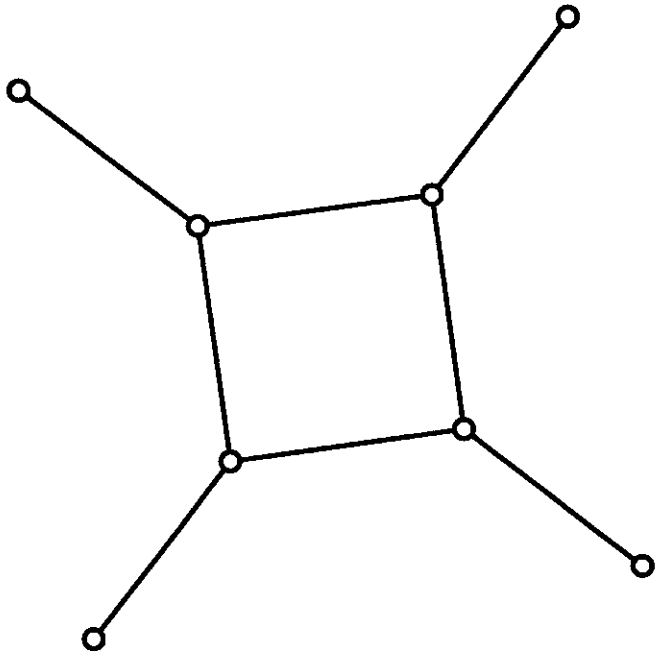

Figure 4.7: Figure 6(a) from Kamada and Kawai, better (0.8s)

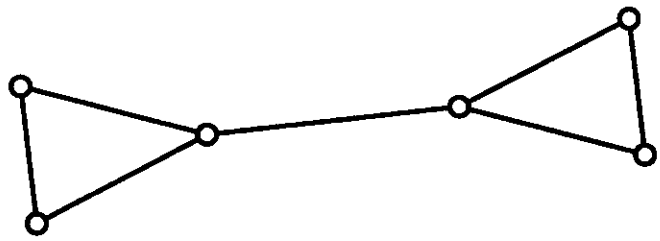

Figure 4.8: Figure 4 from Kamada and Kawai (0.3s)

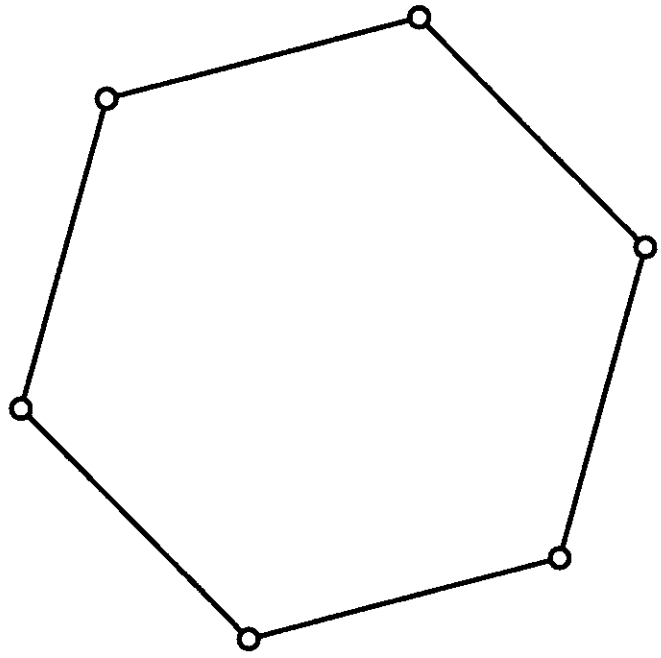

Figure 4.9: Figure 3 from Kamada and Kawai (0.3s)

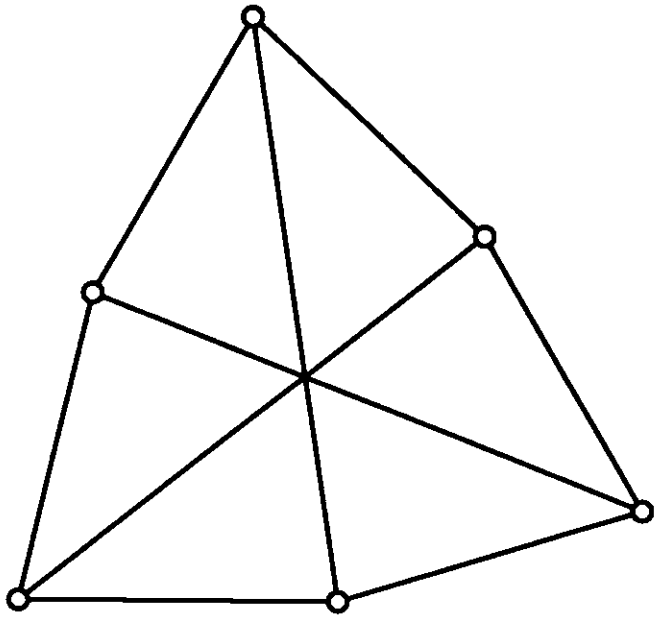

Figure 4.10: $K_{3,3}$ (0.4s)

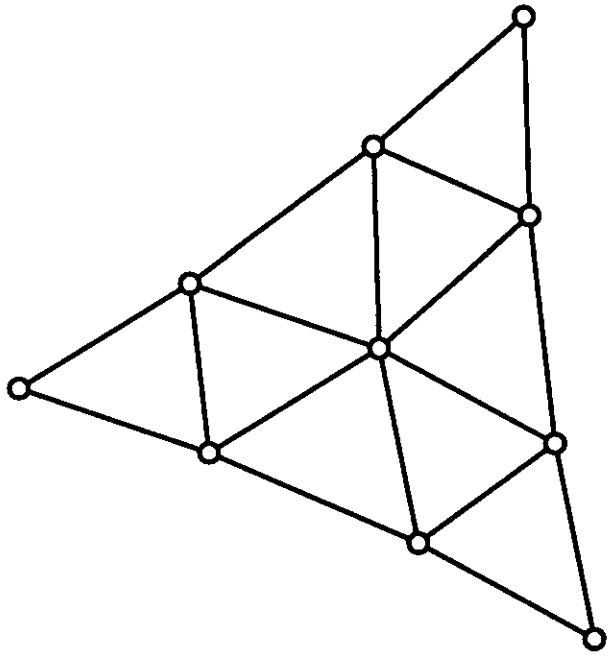

Figure 4.11: Figure 6(c) from Kamada and Kawai (0.9s)

```
gloss -knr 1 -kmel 1 -kcp 1
```

The centripetal aesthetic helps assure that all of the "point" vertices remain at the outside. We can also draw this graph with the "point" vertices *inside*, as in Figure 4.5, using

```
gloss -knr 1 -kmel 1 -kmei 1
```

to give Figure 4.13. In this figure, we see a drawback of our current implementation of the edge crossing minimization aesthetic. The proximity of the midpoints of the two center crossing edges causes these two edges to be pushed out of their natural position, with the result that the center square is somewhat skewed. It may be helpful to remove this aesthetic during final cooling, but an aesthetic that lacked this fault would be better.

The graph in Figure 4.14 is very similar to Figure 27 of [FR91]. Disappointingly, we were unable to adjust our current aesthetics to produce the planar layout of this graph shown in Figure 20 of [DH89]. It would be useful to study this further to see if we can discover whether different aesthetics are needed or whether this might indicate a limitation of our method.

Figure 4.15 is an icosahedron that has been modified by removing one node and its incident edges. [FR91] includes two layouts of this graph, one of which is essentially the same as this figure. The other looks a little like Figure 4.16, but our version is better because we have only four edge crossings rather than eight and maintain a much better separation between vertices and nonincident edges.

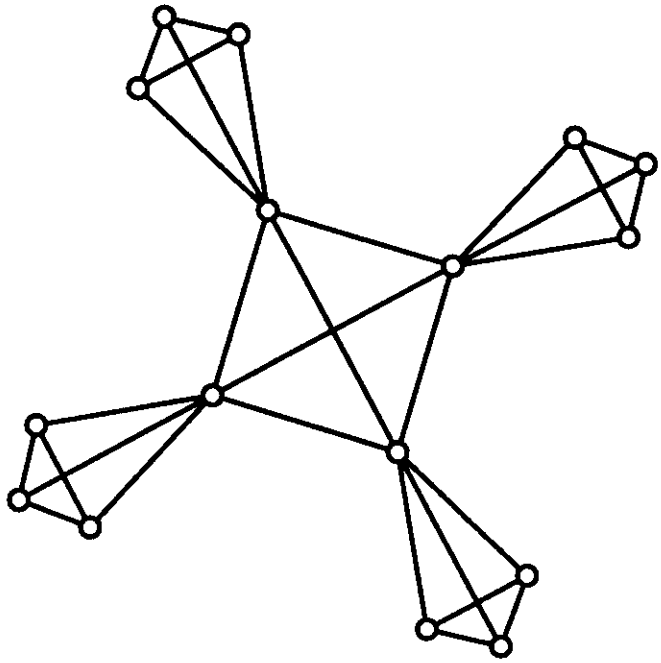

Figure 4.12: Figure 16 from Davidson and Harel (2.2s)

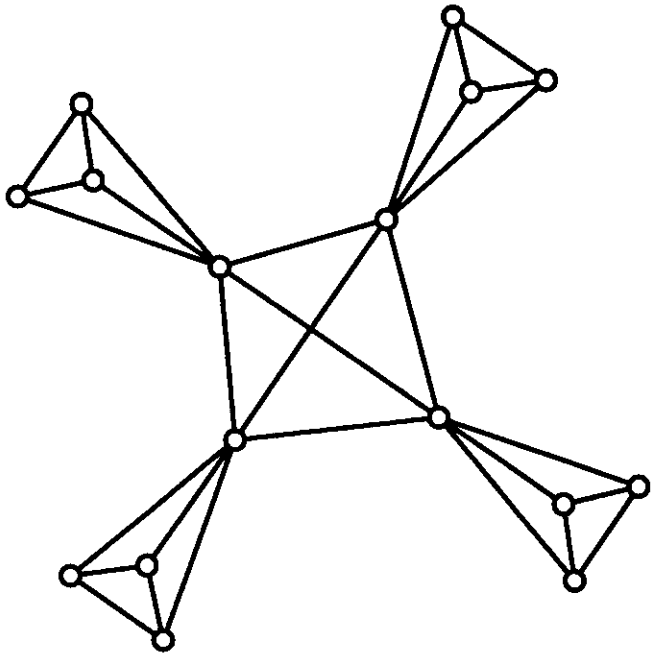

Figure 4.13: Figure 16 from Davidson and Harel, variant (6.9s)

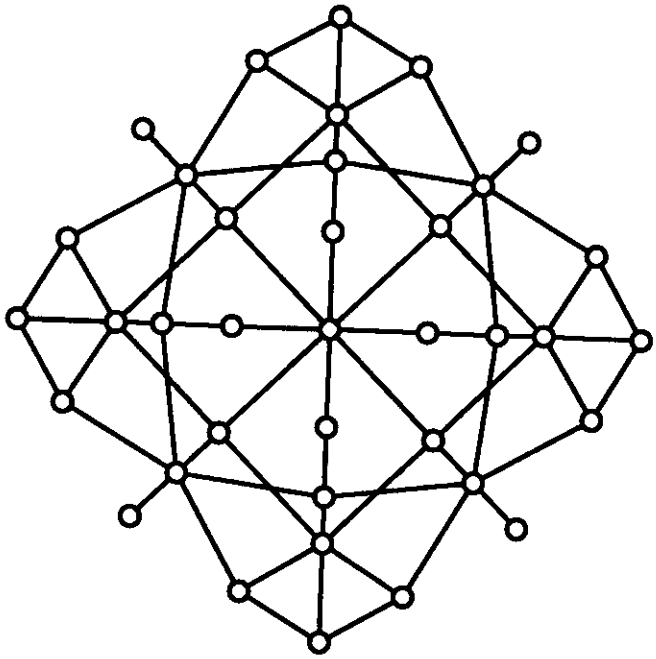

Figure 4.14: Figure 20 from Davidson and Harel (8.9s)

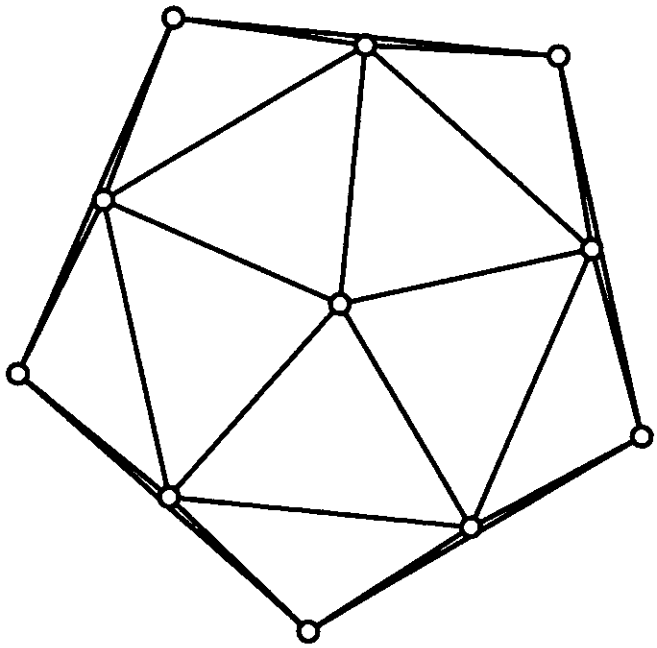

Figure 4.15: Figure 28 from Fruchterman and Reingold (1.1s)

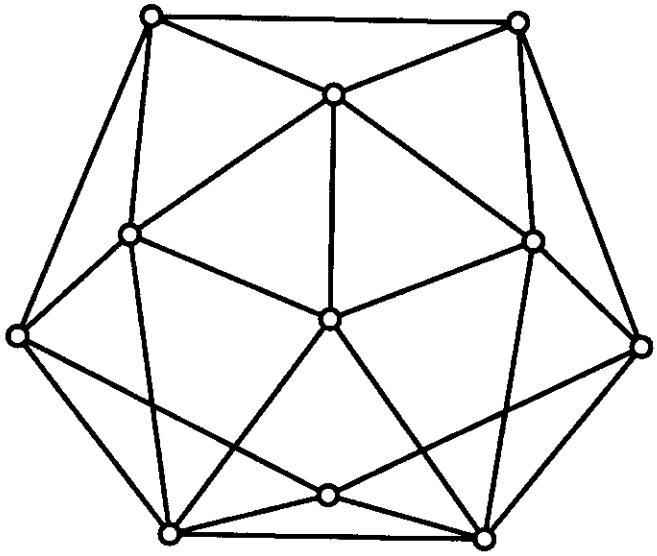

Figure 4.16: Figure 28 from Fruchterman and Reingold, another version (6.4s)

We produced this latter variant by adding the node/edge repulsion aesthetic. It is not entirely clear how [FR91] produce their variations. Presumably, they are either altering cooling parameters, or the variations are a result of the instability of the algorithm (with respect to starting state, etc.) While our method also sometimes suffers from instability, we believe that our ability to control the style of the variations produced is an improvement over their method.

Figure 4.17 shows an icosahedron. We produce the improved layout in Figure 4.18 by replacing node repulsion with centripetal repulsion. Strangely, [FR91] produce the latter version rather than the former. [DH89] are able to produce a planar, symmetric layout of the icosahedron.

In Figures 4.19 through 4.28, we show more simple graphs included in [FR91]. Our layouts are very similar to theirs, which are in turn quite similar to the original

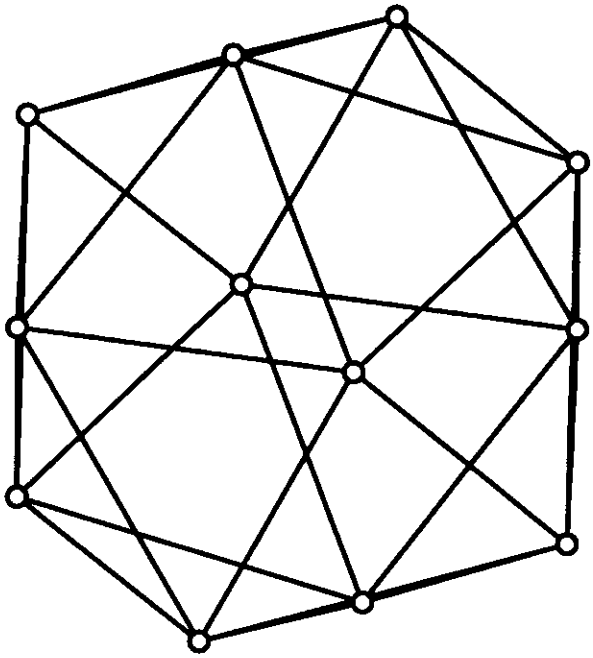

Figure 4.17: Figure 29 from Fruchterman and Reingold (1.3s)

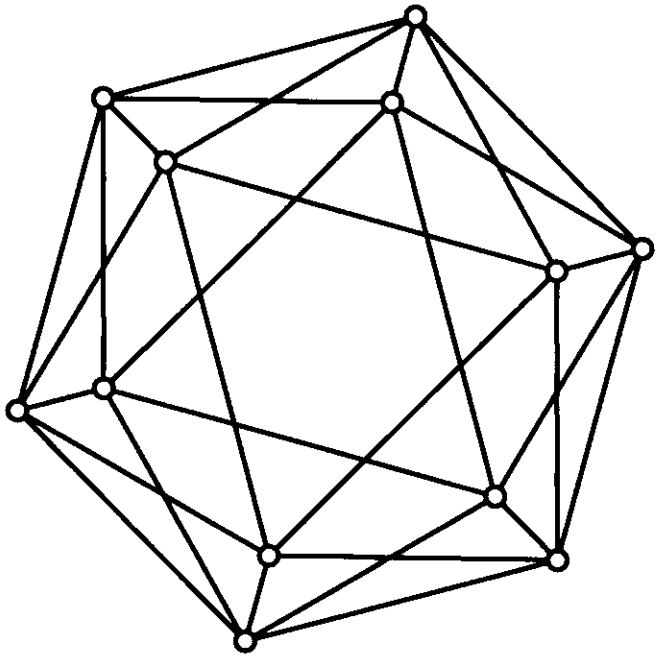

Figure 4.18: Figure 29 from Fruchterman and Reingold, better (0.7s)

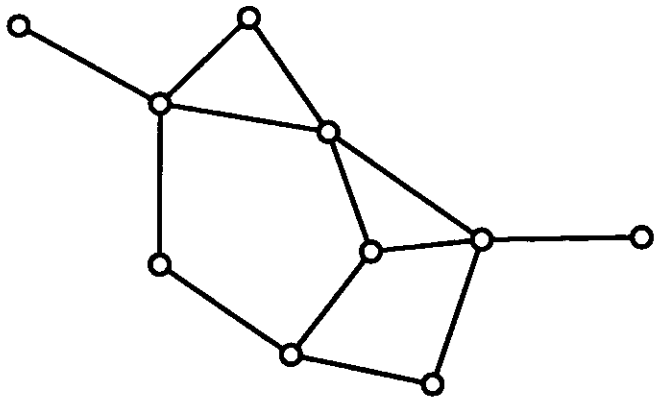

Figure 4.19: Figure 7(c) from Kamada and Kawai (0.8s)

sources.

In Figure 4.27, we are able to produce a planar version of Figure 4.26, which [FR91] do not do, by including the edge crossing minimization aesthetic.

Figure 4.29 shows a pentagonal prism, much as [FR91] render it. They fail to produce a planar version, while [DH89] succeed with a somewhat skewed version. Our planar version (Figure 4.30), generated with the command:

```
gloss -kcp 1 -kmel 1 -kmei 10 -kner 1
```

is clearer and more aesthetically attractive.

Our version of the twin cubes graph is given in Figure 4.31. Though this graph is planar, [FR91] and [DH89] are unable to planarize it, and we fail as well.

Our first version of the 24-node mesh (Figure 4.32) is essentially the same as that of [FR91]. By adding more aesthetics,

```
gloss -knr 1 -kmel 1 -kmei2 10 -kner 1 -kcp 1 -it 2000
```

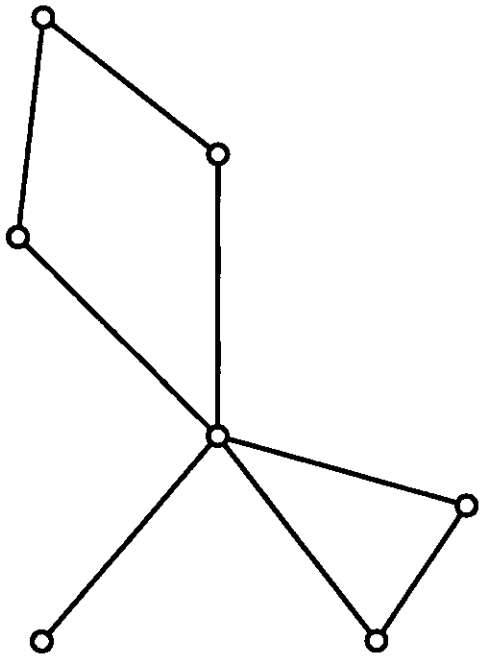

Figure 4.20: Figure 7(a) from Kamada and Kawai (0.4s)

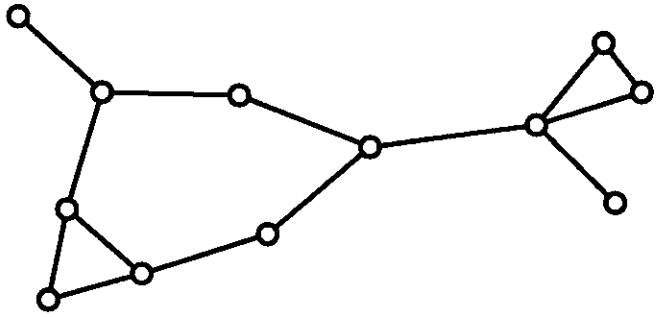

Figure 4.21: Figure 7(d) from Kamada and Kawai (1.2s)

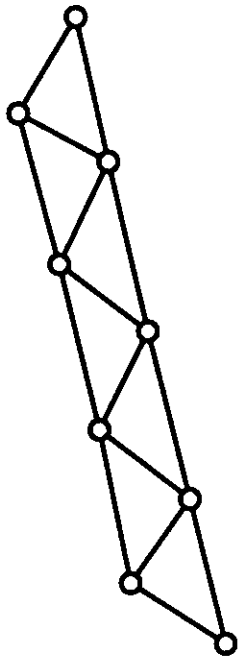

Figure 4.22: Figure 2(b) from Eades (0.7s)

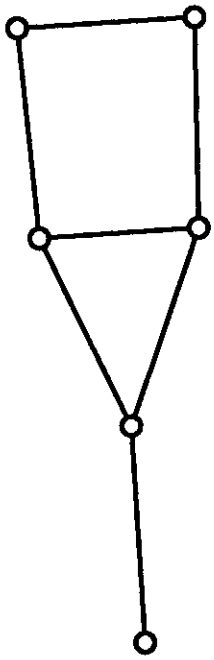

Figure 4.23: Figure 5(c) from Eades (0.3s)

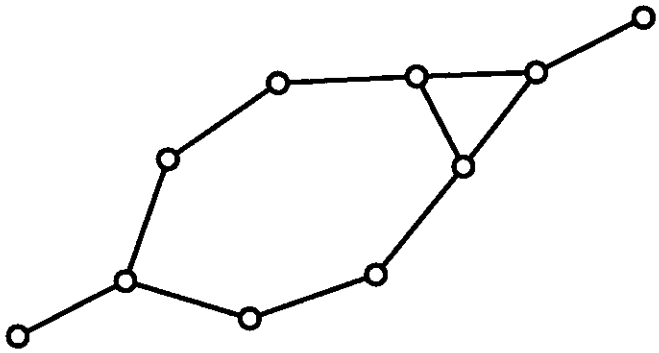

Figure 4.24: Figure 5(b) from Eades (0.8s)

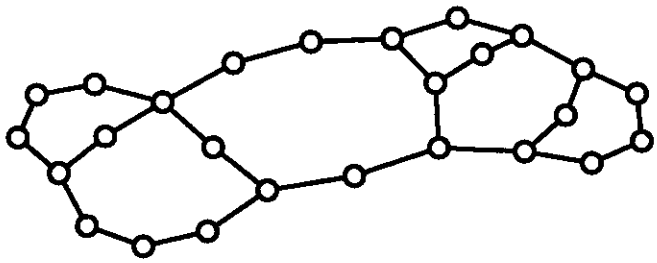

Figure 4.25: Figure 5(a) from Eades (4.4s)

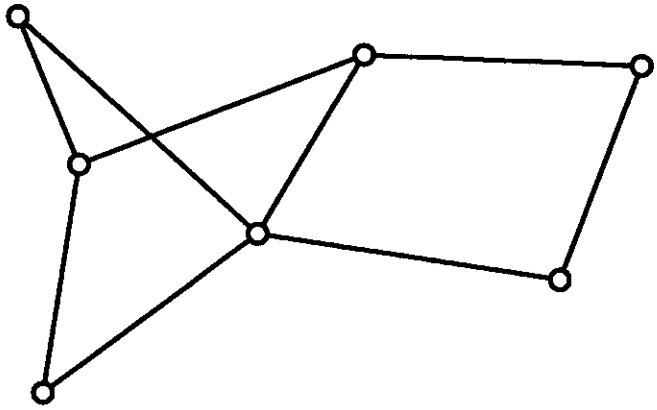

Figure 4.26: Figure 6(c) from Eades (0.5s)

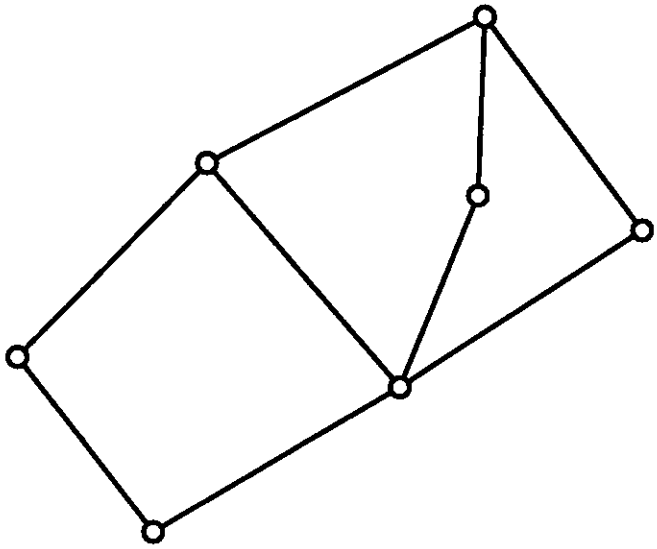

Figure 4.27: Figure 6(c) from Eades, planar (2.0s)

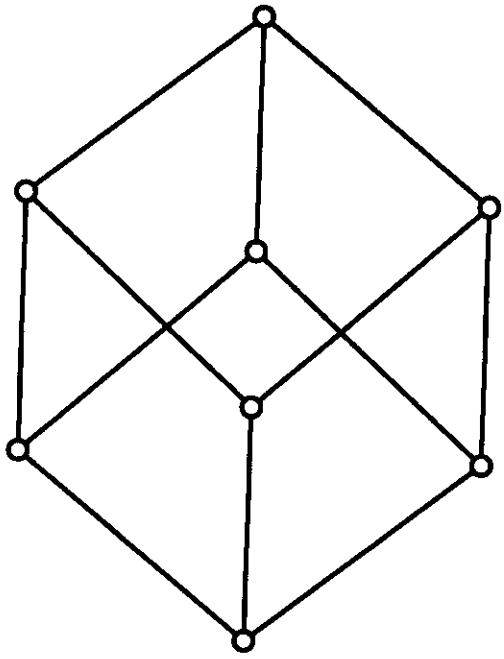

Figure 4.28: Figure 47 from Fruchterman and Reingold (0.6s)

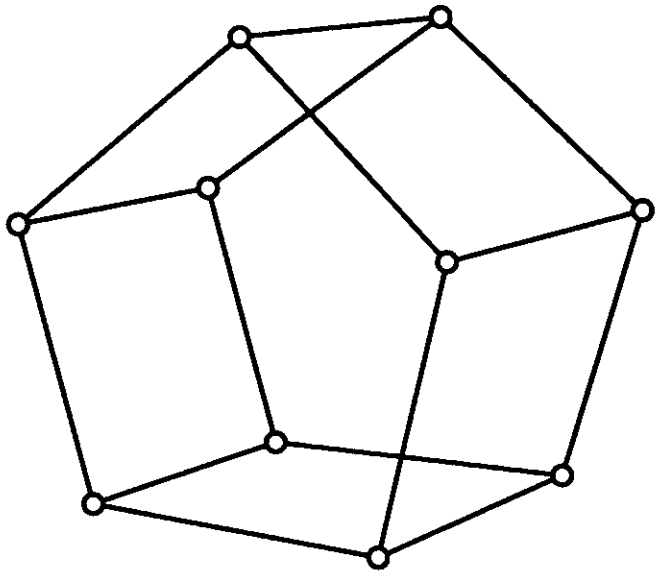

Figure 4.29: Figure 49 from Fruchterman and Reingold (0.9s)

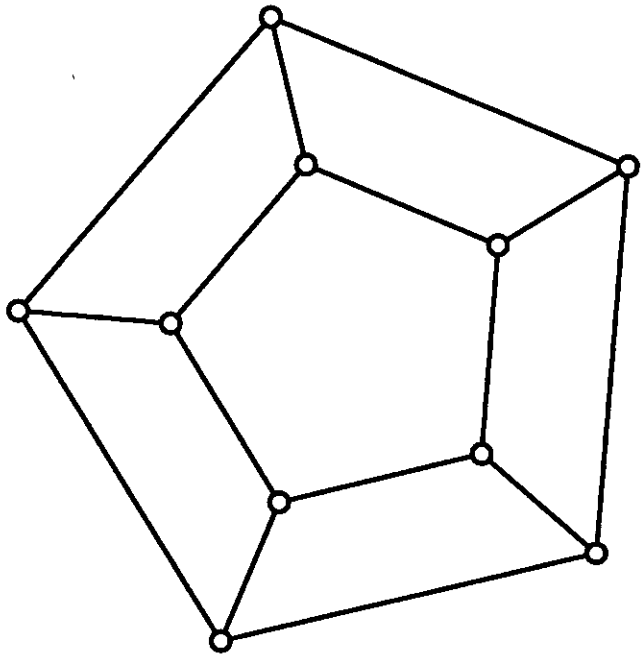

Figure 4.30: Figure 49 from Fruchterman and Reingold, planar (4.5s)

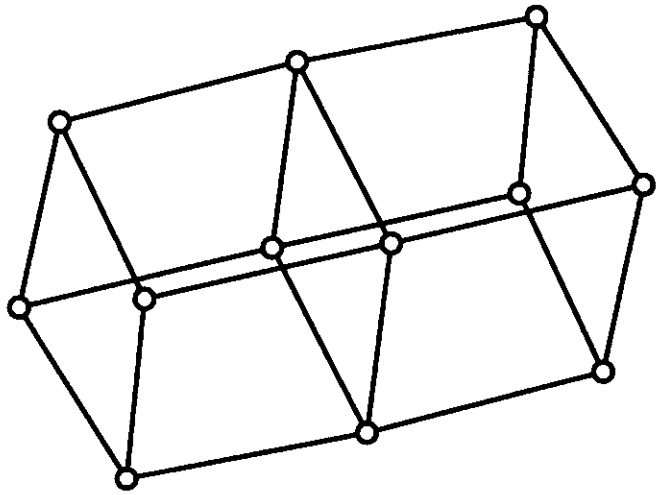

Figure 4.31: Figure 11 from Davidson and Harel (1.2s)

we produce the planar layout in Figure 4.33, which is very close to that of [DH89]. The planar and nonplanar versions are arguably approximately equivalent in clarity.

Three versions of a dodecahedron are shown in Figures 4.34 through 4.36. (Figures 4.35 and 4.36 both use the node/edge repulsion and edge crossing minimization aesthetics, but the latter uses the stronger edge crossing aesthetic mentioned in Section 3.2.4.) [FR91] produce drawings like Figures 4.34 and 4.36, while those of [DH89] are like the layouts in Figures 4.34 and 4.35. Neither successfully planarizes the dodecahedron, nor do we.

We show a series of complete graphs in Figures 4.37 through 4.47. Figure 4.42, produced with the command:

```
gloss -kcp 1 -kmel 1 -kmei2 10 -kner 1
```

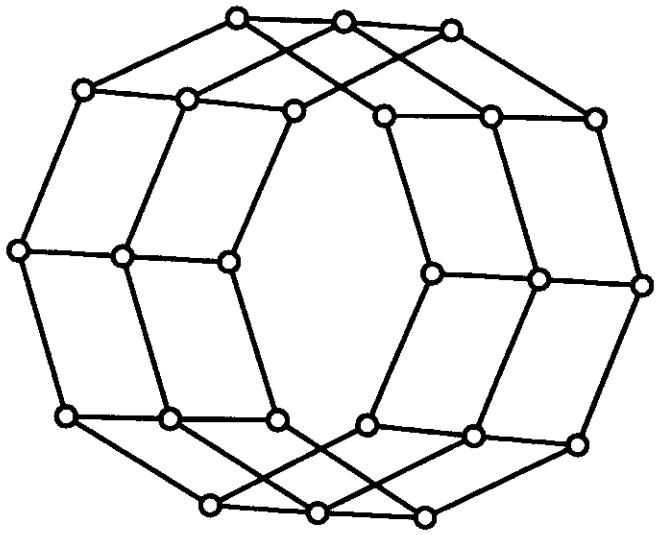

Figure 4.32: Figure 2 from Davidson and Harel (4.0s)

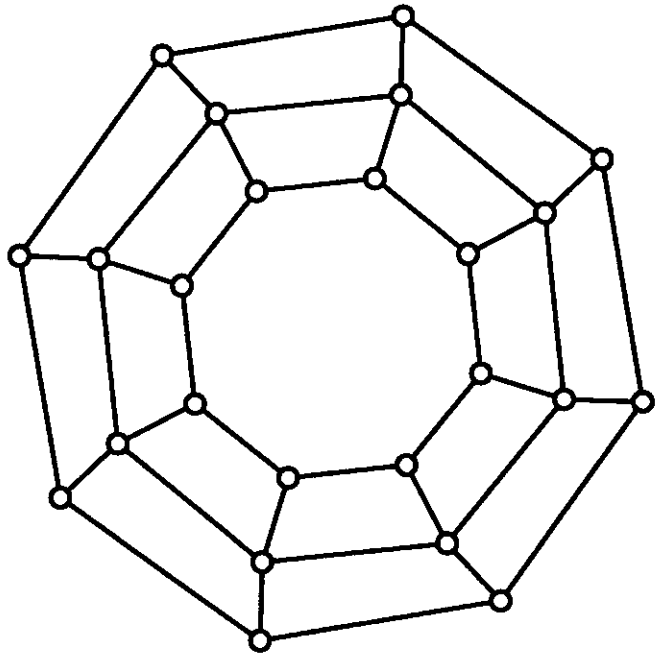

Figure 4.33: Figure 2 from Davidson and Harel, planar (60.1s)

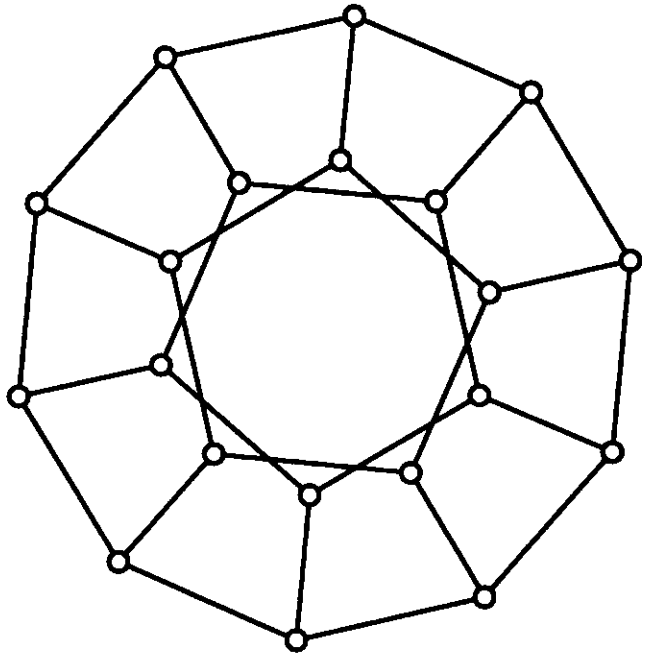

Figure 4.34: Figure 57 from Fruchterman and Reingold, version 1 (2.8s)

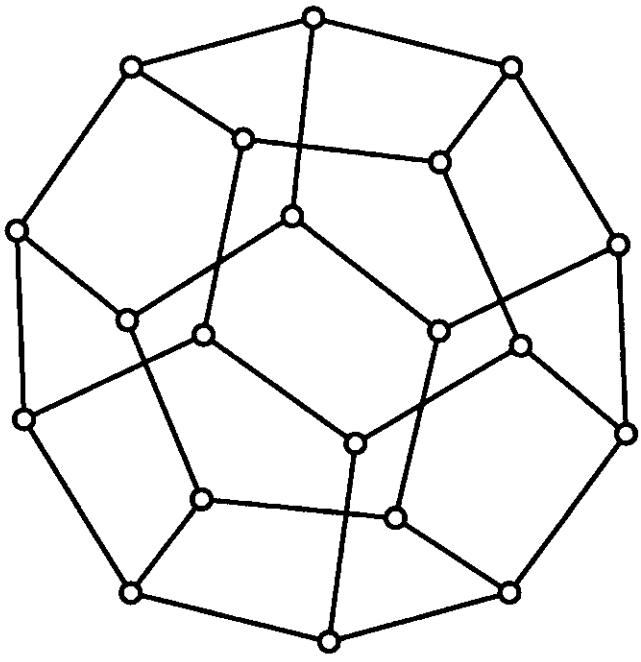

Figure 4.35: Figure 57 from Fruchterman and Reingold, version 2 (20.0s)

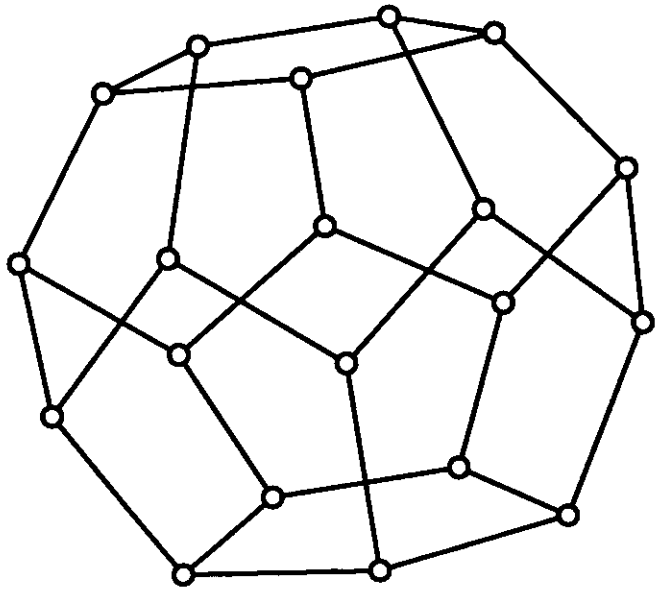

Figure 4.36: Figure 57 from Fruchterman and Reingold, version 3 (19.8s)

is a variant of Figure 4.41 that has fewer edge crossings. [DH89] also produce both versions.

The larger graphs can generally be rendered either with or without a central vertex (cf. Figures 4.46 and 4.47). We can control this with the use of the centripetal aesthetic.

Both [DH89] and [FR91] attempt to produce a symmetric layout for the Heywood graph, but are unable to do so, producing graphs that are fairly similar to our Figure 4.48. Using our centripetal aesthetic, we are able to produce the layout in 4.49, using only 0.7 seconds of CPU time! This placement differs from the ideal suggested by [DH89] (Figure 4.50), which we were unable to achieve. The ideal has fewer edge crossings, but our layout is just as readable, possibly more so.

Figure 4.37: K_2 (0.1s)

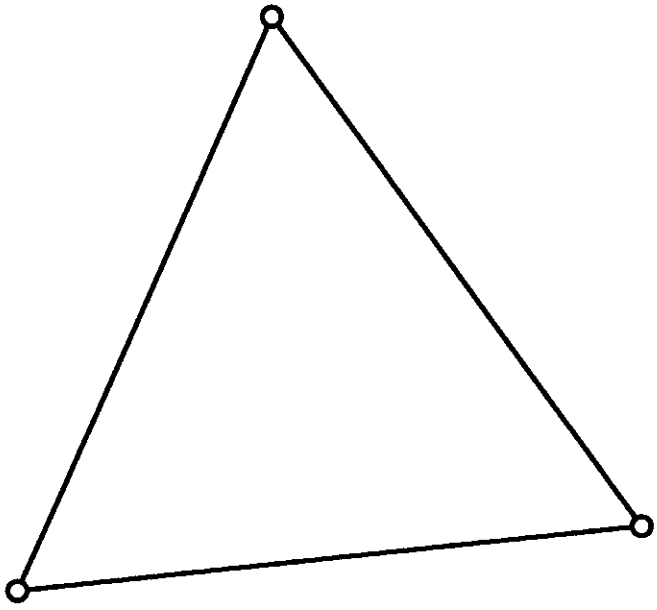

Figure 4.38: K_3 (0.1s)

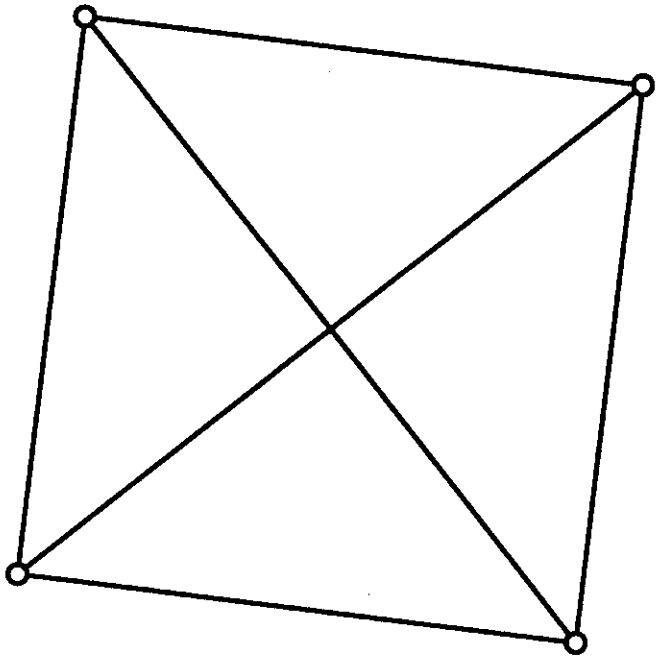

Figure 4.39: K_4 (0.2s)

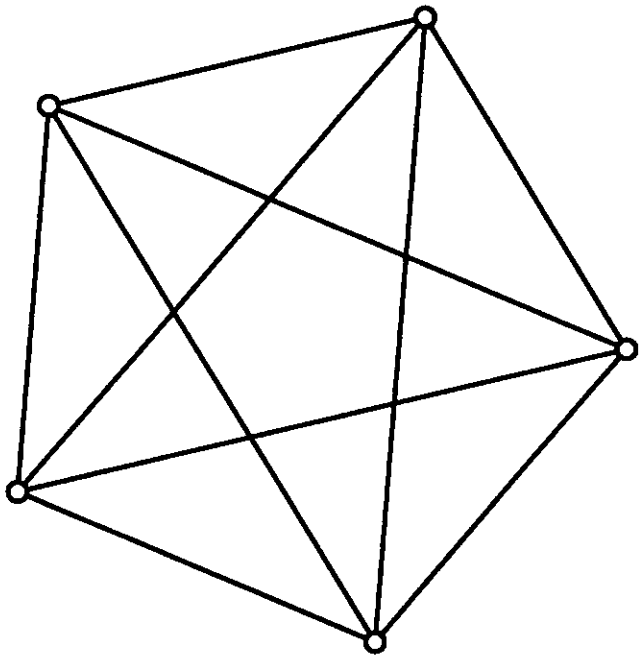

Figure 4.40: K_5 (0.3s)

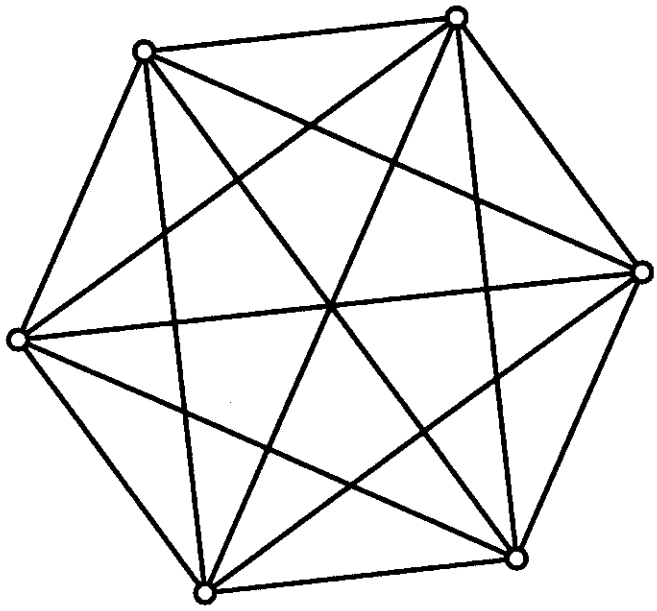

Figure 4.41: K_6 (0.5s)

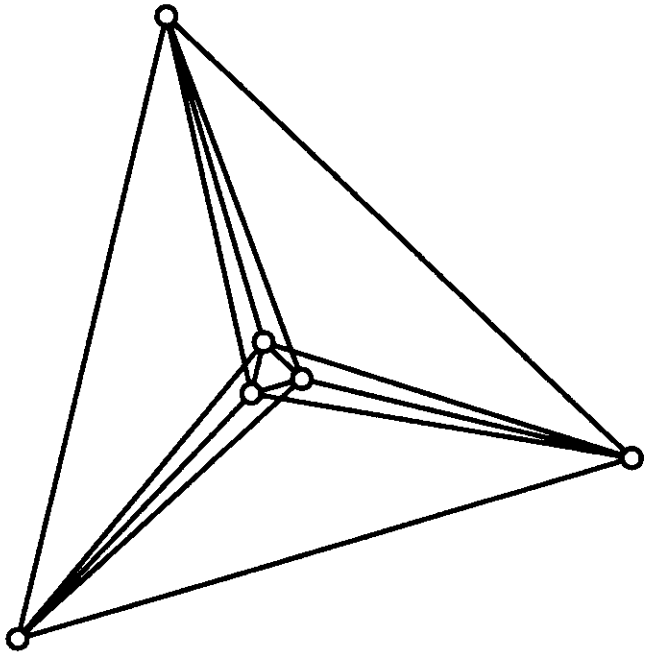

Figure 4.42: K_6, variant (2.9s)

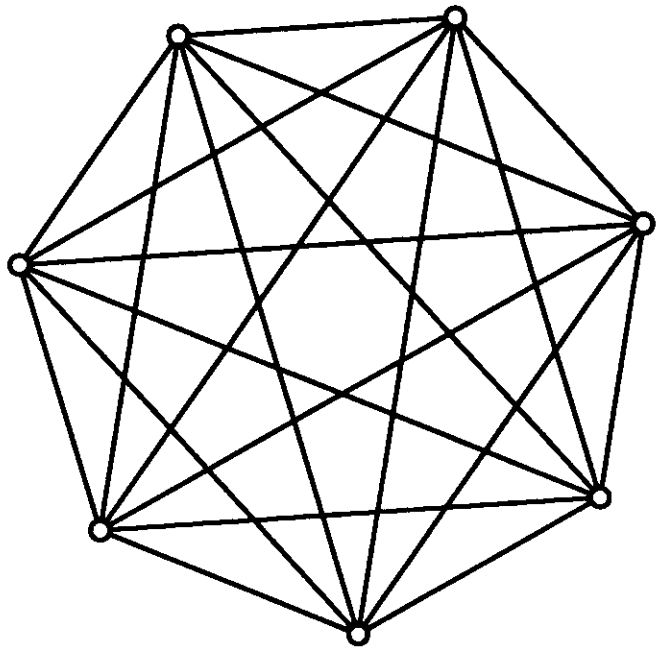

Figure 4.43: K_7 (0.7s)

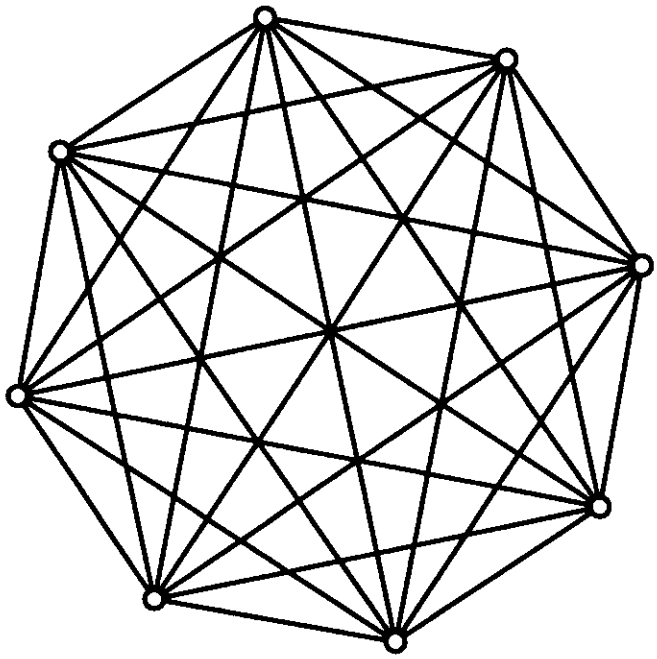

Figure 4.44: K_8 (0.8s)

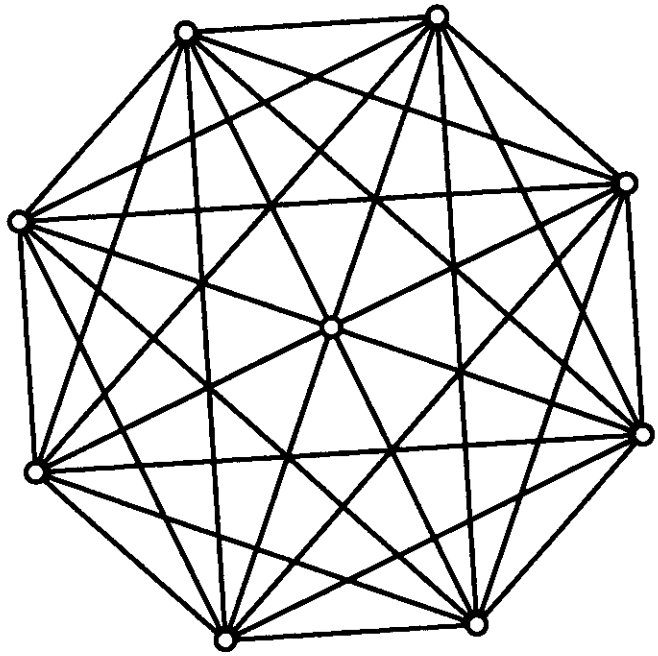

Figure 4.45: K_9 (1.0s)

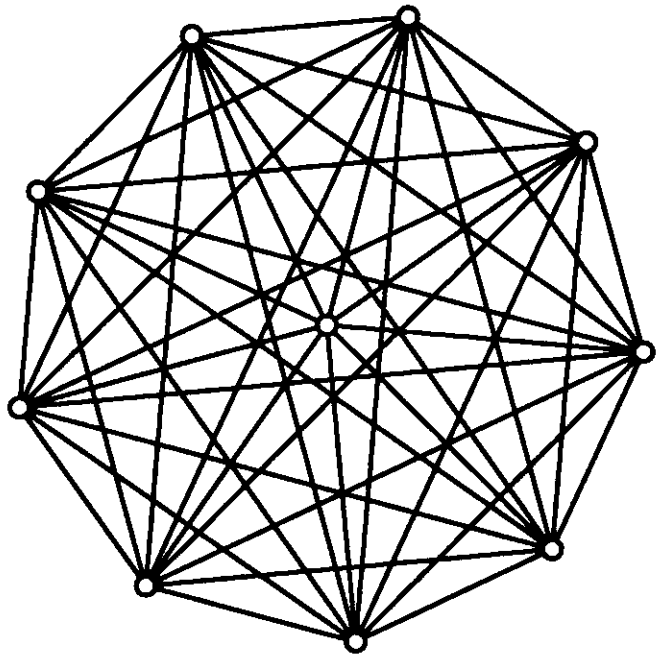

Figure 4.46: K_{10} (1.3s)

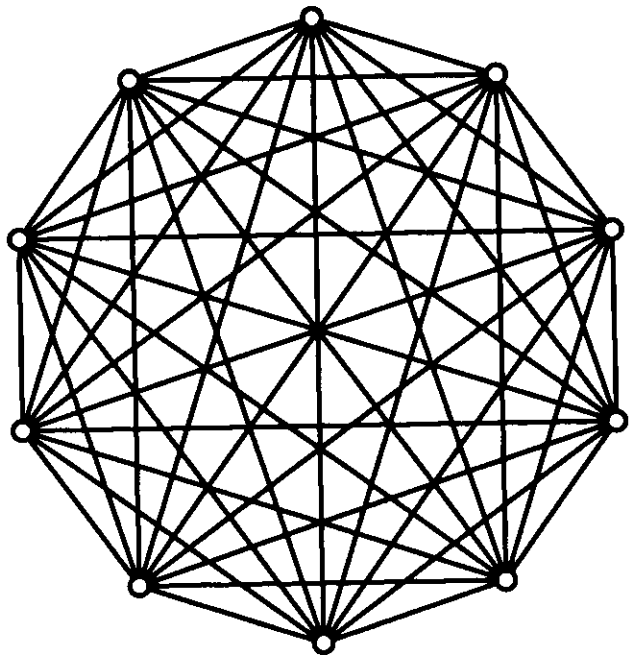

Figure 4.47: K_{10}, another variant (0.9s)

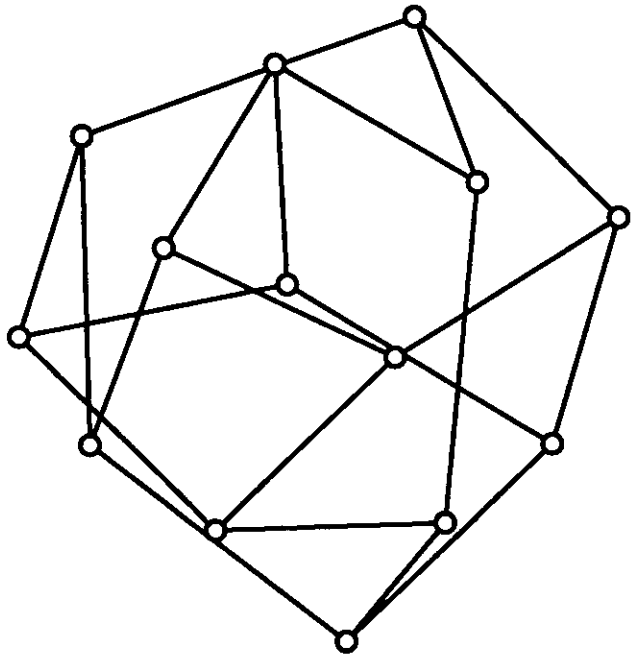

Figure 4.48: Figure 18 from Davidson and Harel (1.5s)

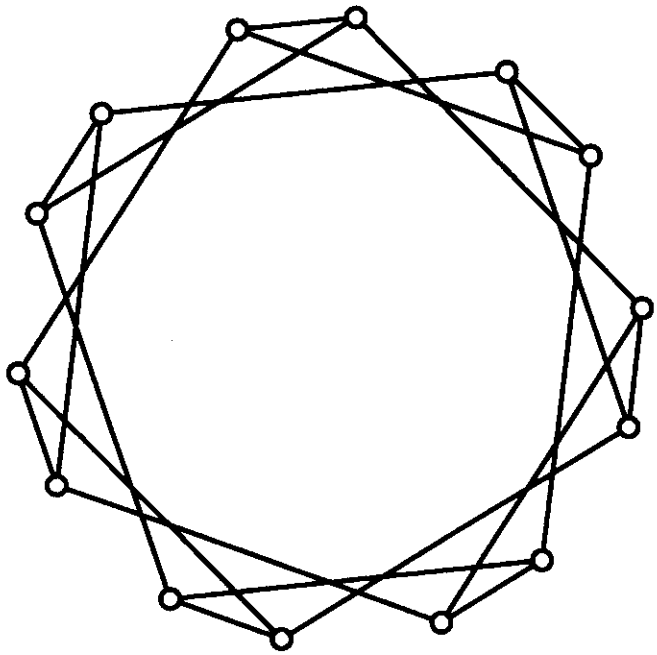

Figure 4.49: Figure 18 from Davidson and Harel, better (0.7s)

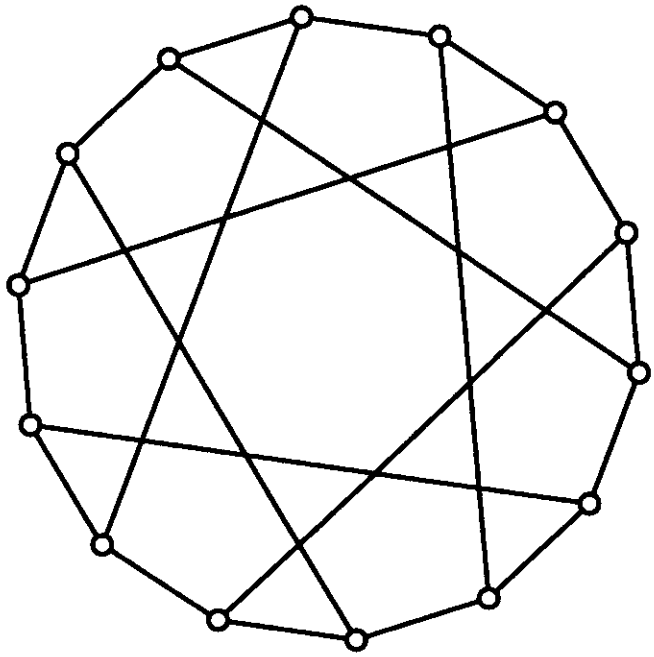

Figure 4.50: Figure 18 from Davidson and Harel, their proposed ideal

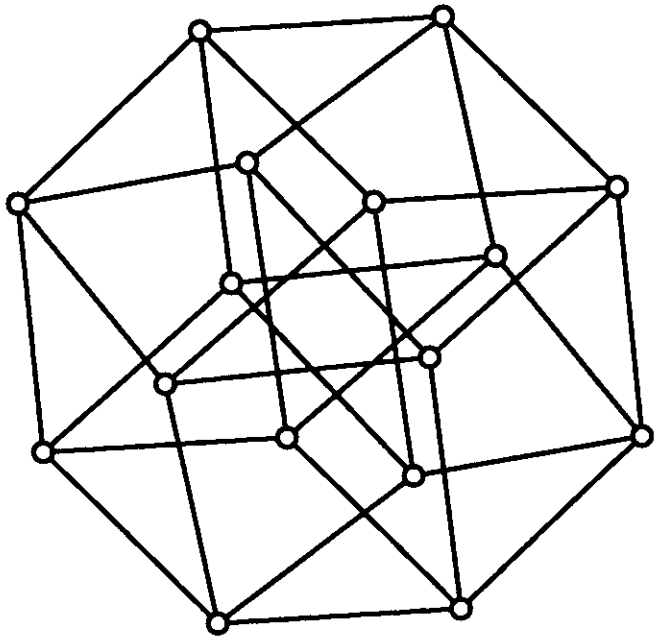

Figure 4.51: Figure 48 from Fruchterman and Reingold (2.0s)

Figures 4.51 through 4.53 are all renderings of the four-dimensional hypercube. Figure 4.51 is fairly close to the version in Figure 48 of [FR91], though ours achieves better separation between vertices and nonincident edges via the node/edge repulsion aesthetic. Adding the centripetal aesthetic gives Figure 4.52, which is better than the first version because it shows the symmetry of the hypercube. Using only the basic two aesthetics, we obtain Figure 4.53. Although this layout is neither symmetric nor particularly attractive in an "artistic" sense, it has an interesting feature that the other two versions lack: if one looks carefully, it is possible to see two coaxial tubes (each being a twin cube, as in Figure 4.31). This "mnemonic" makes the structure of the hypercube easier to visualize and think about, albeit hiding some of its symmetry.

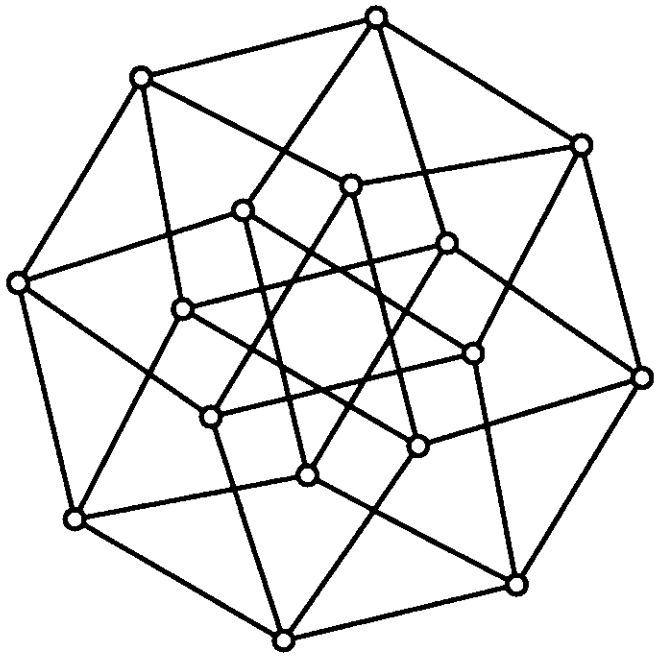

Figure 4.52: Figure 48 from Fruchterman and Reingold, symmetric (13.3s)

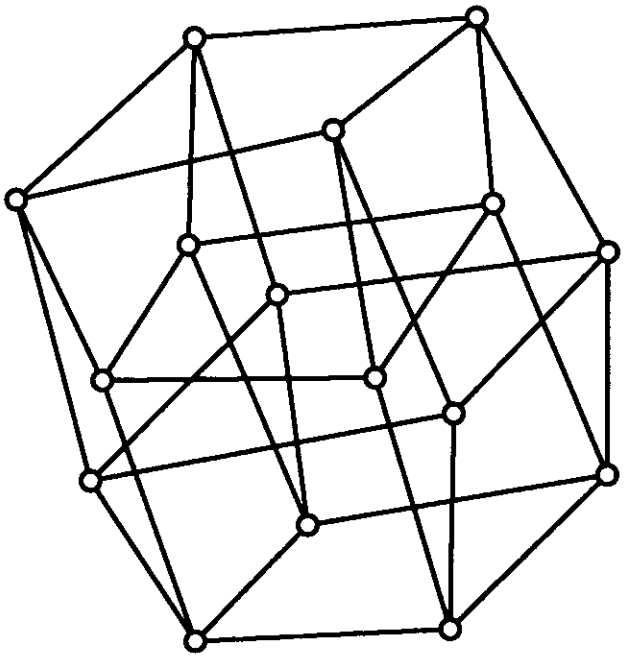

Figure 4.53: Figure 48 from Fruchterman and Reingold, another variant (13.1s)

4.2.2 Directed Acyclic Graphs

In this section, we show how directed acyclic graphs (DAGs) can be rendered using **aglo**. The same algorithm used to do layout of general graphs in the previous section is applied here to DAGs, showing the generality of our method.

Though arrowheads are not shown in these figures, all edges are directed and run from left to right. Except as mentioned, all figures in this section were produced by the command

```
gloss -knr 1 -kmel 1 -kner 1 -kpl 1 -kmlv 100
```

The placements in Figures 4.54 through 4.56 seem reasonable. They appear somewhat amorphous, lacking the "graph paper" look of discrete algorithms, such as that in [RT81]. (It is not wholly clear that this is a disadvantage.) Figure 4.55 includes a light dose of the edge crossing minimization aesthetic to ensure planarity.

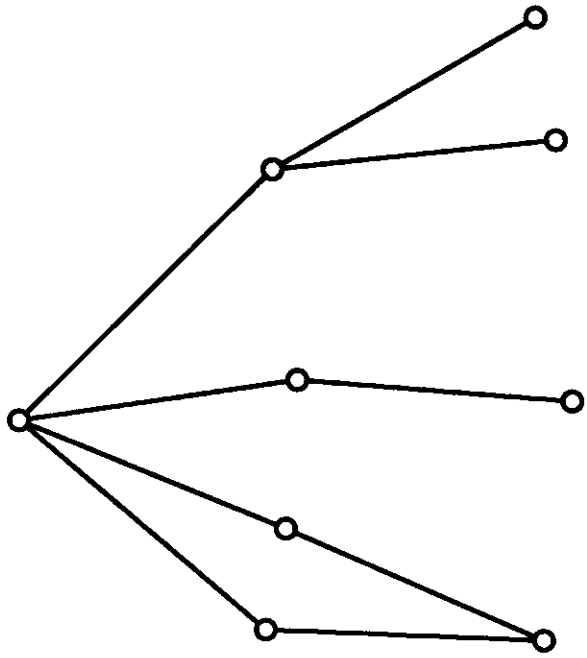

Figure 4.54: A small DAG (2.9s)

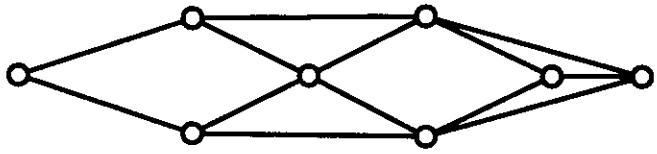

Figure 4.55: A small, symmetric DAG (3.4s)

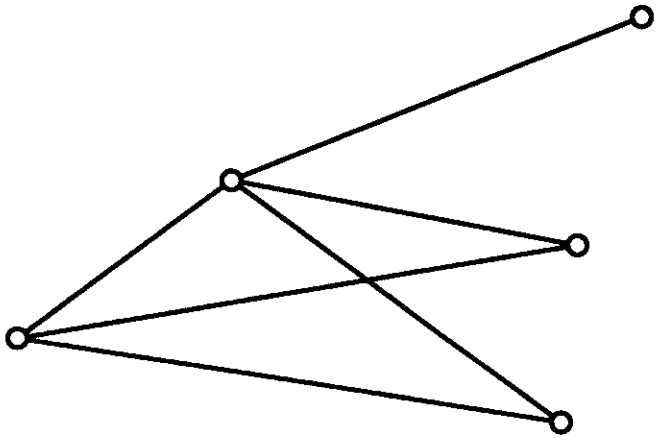

Figure 4.56: A small, nonplanar DAG (1.2s)

4.2.3 Trees

In this section, we explore tree drawing with **aglo**, further demonstrating the generality of the AGLO algorithm. (Again, arrowheads are not shown in these figures, but all edges are directed.)

We begin with a small binary tree (Figure 4.57). This layout utilizes both tree aesthetics (parent left and level variance minimization) as well as edge crossing minimization, node/edge repulsion, and the basic two (node repulsion and edge length minimization).

Figure 4.58 shows a binary tree that the current **aglo** aesthetics handle poorly. Since order of siblings is unspecified, layout is arbitrary in this respect, but we

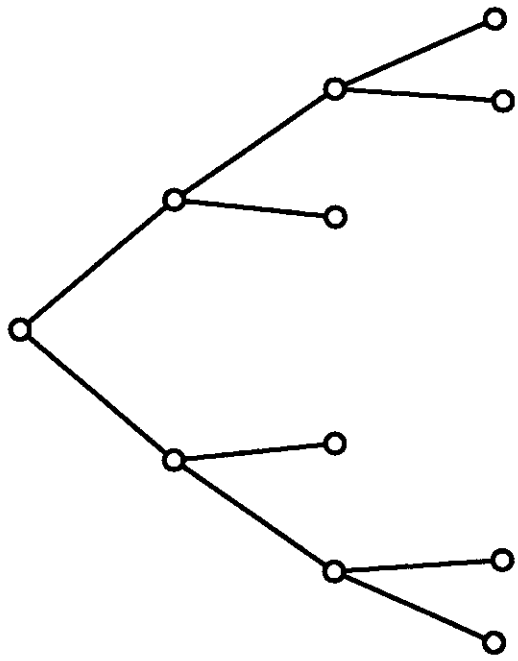

Figure 4.57: A small binary tree (3.9s)

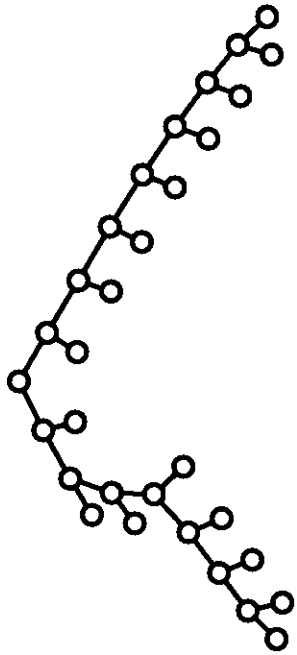

Figure 4.58: A binary tree conundrum (26.1s)

would really much rather see a symmetric display in this case.

Figures 4.59 and 4.60 show complete binary trees of height two and four, respectively, drawn as general graphs. Figure 4.61 is the latter graph drawn as a tree. The levels of the tree have a slight curve, but the layout as a whole is quite adequate.

Figures 4.62 through 4.65 are simple trees that appear in [FR91]. Our versions differ only slightly.

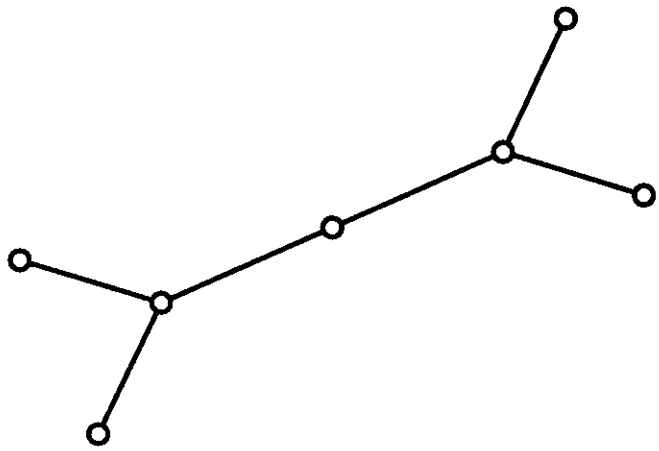

Figure 4.59: Figure 40 from Fruchterman and Reingold (0.4s)

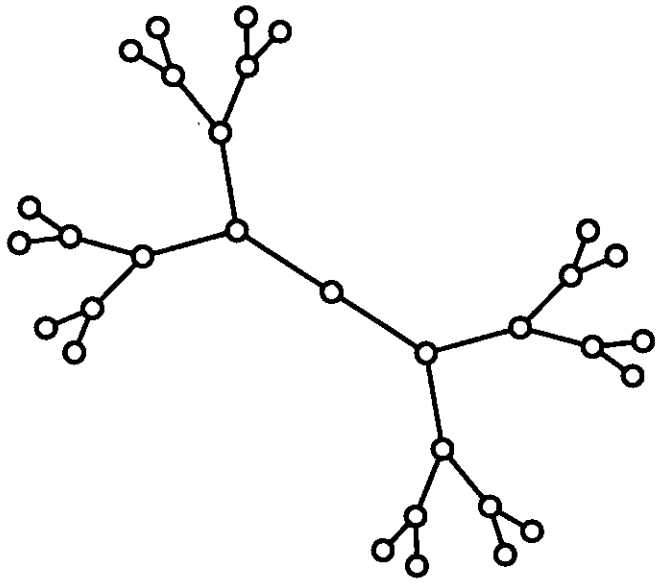

Figure 4.60: Figure 14(a) from Davidson and Harel (6.1s)

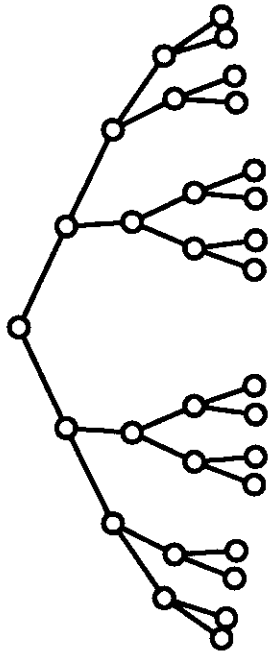

Figure 4.61: Figure 14(a) from Davidson and Harel, as a tree (28.0s)

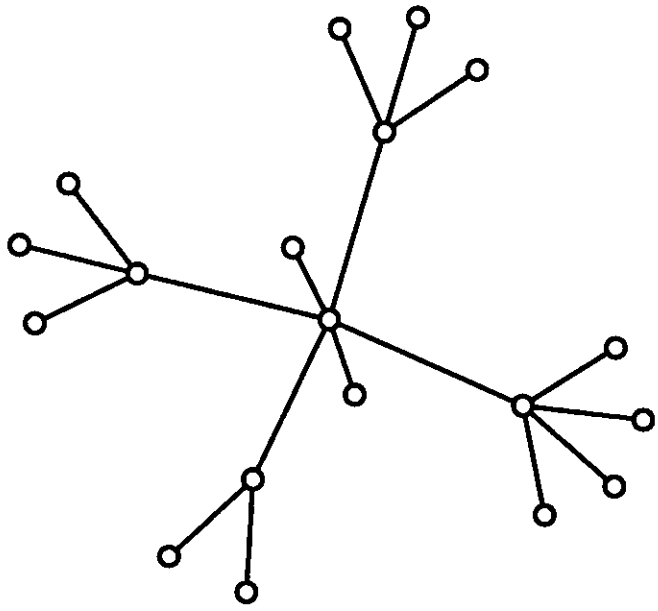

Figure 4.62: Figure 4(a) from Eades (2.4s)

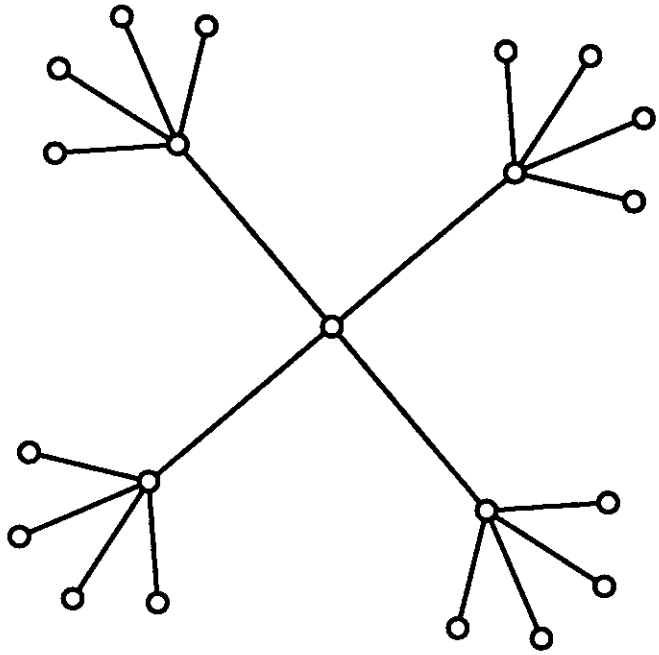

Figure 4.63: Figure 14(b) from Davidson and Harel (2.9s)

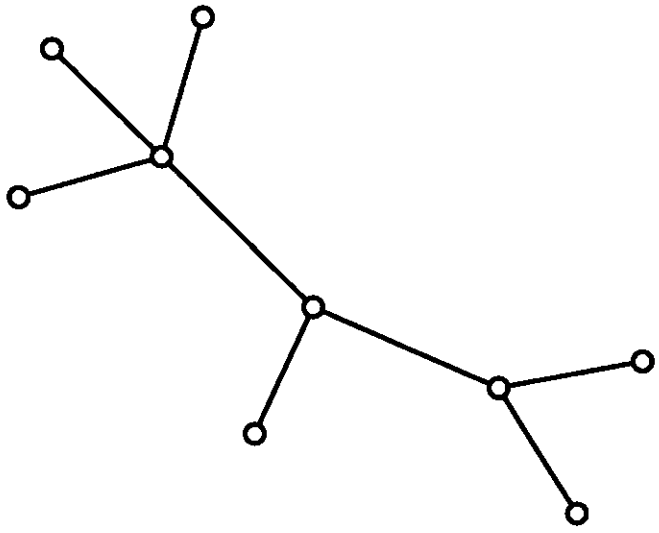

Figure 4.64: Figure 7(b) from Kamada and Kawai (0.6s)

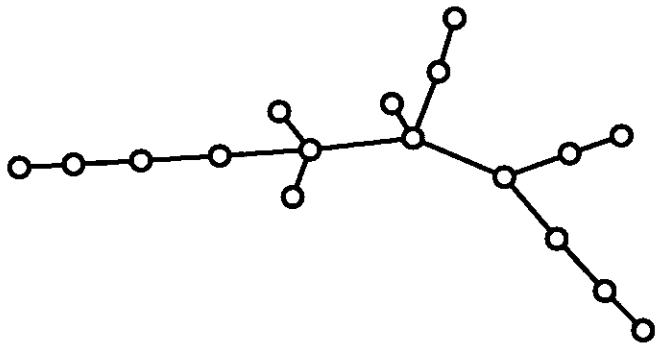

Figure 4.65: Figure 4(b) from Eades (1.9s)

4.2.4 Large-scale Examples

In this section, we apply **aglo** to several large graphs. The results are mixed, though much better than we expected, and show the competence of the AGLO algorithm on larger examples.

One real-world application of graph layout is the visualization of call graphs. In Figure 4.66, we render the call graph of a large module (around 90 vertices and 460 edges) as a general graph.[3] The resulting layout does show some gross features of the call graph, such as node clusters and outliers, but the layout could be better. There is really too much information for full comprehension in any case.

[3]The data for Figure 4.66 courtesy of Twin Sun, Inc.

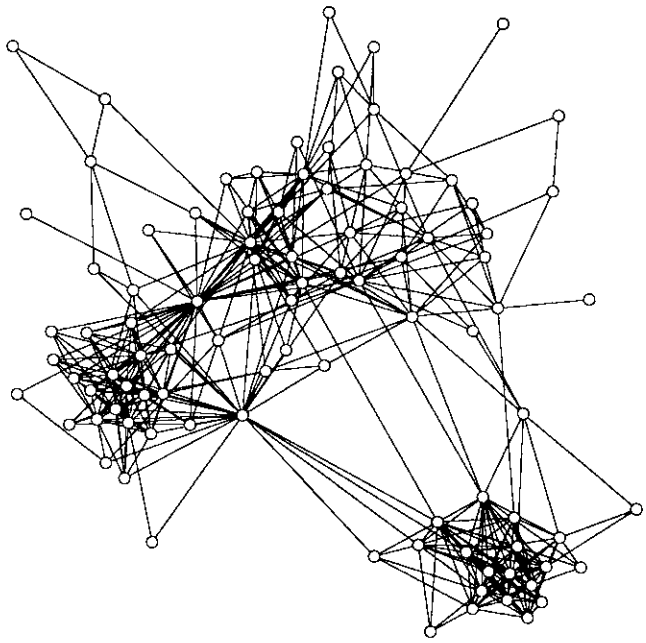

Figure 4.66: A large call graph (55.8s)

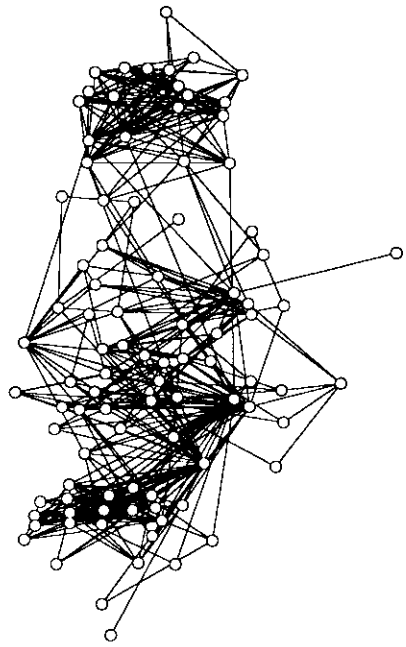

Figure 4.67: A large call graph, variant (57.8s)

The same call graph is rendered with the parent left aesthetic in Figure 4.67. This shows the direction of the calls better, of course, but it is inferior to the previous version.

In Figure 4.68, we have a large regular mesh that describes the surface of an ellipsoid.[4] The resulting layout is quite attractive considering the size of the mesh (around 400 vertices and 800 edges). As the figure shows, this is really too much information for a single visual—we can see the overall structure of the graph, but we would be hard pressed to notice a missing edge (Figure 4.69).

Figure 4.70 shows a toroidal mesh of a similar size. Again, the layout produced is surprisingly attractive, albeit somewhat overwhelming.

[4]Thanks to Junio Hamano of Twin Sun, Inc., who created the data for Figures 4.68 and 4.70.

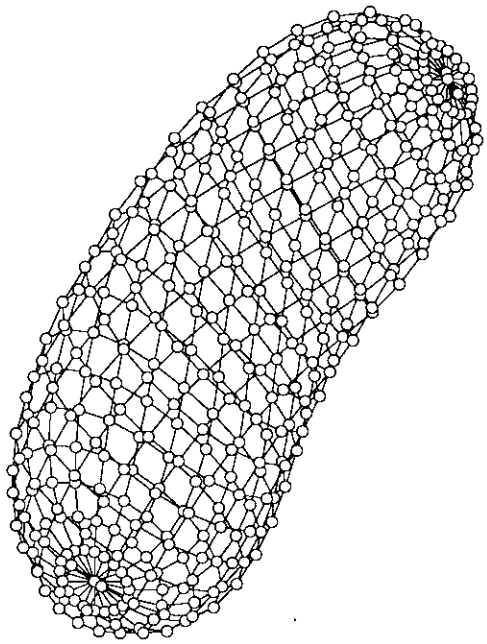

Figure 4.68: Ellipsoid mesh (917.8s)

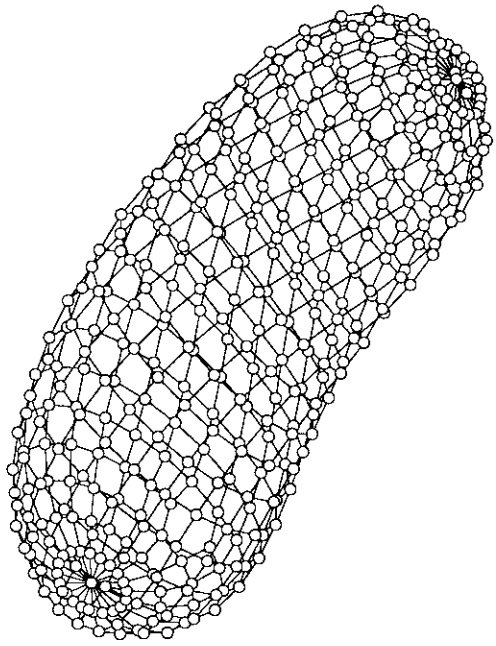

Figure 4.69: Ellipsoid mesh, minus one edge (917.7s)

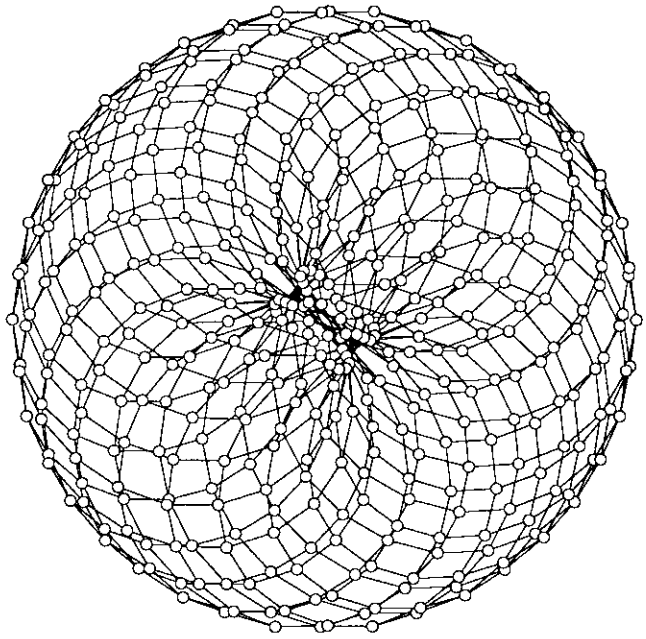

Figure 4.70: Toroidal mesh (909.5s)

Figure 4.71 is a nice layout of a 200-node Möbius strip mesh. Just for fun, Figure 4.72 shows a 400-node Klein bottle.[5]

4.2.5 Summary

Tables 4.2 and 4.3 summarize execution times for the figures in this chapter, and give the exact commands used to produced them.

[5]To construct a Klein bottle, take two Möbius strips and glue their edges together (each strip has only one edge). Now "inflate" the resulting "tube". Notice how difficult this is. (A Klein bottle is not realizable in three-dimensional space and with only one side and no interior, would not be inflatable in any case.)

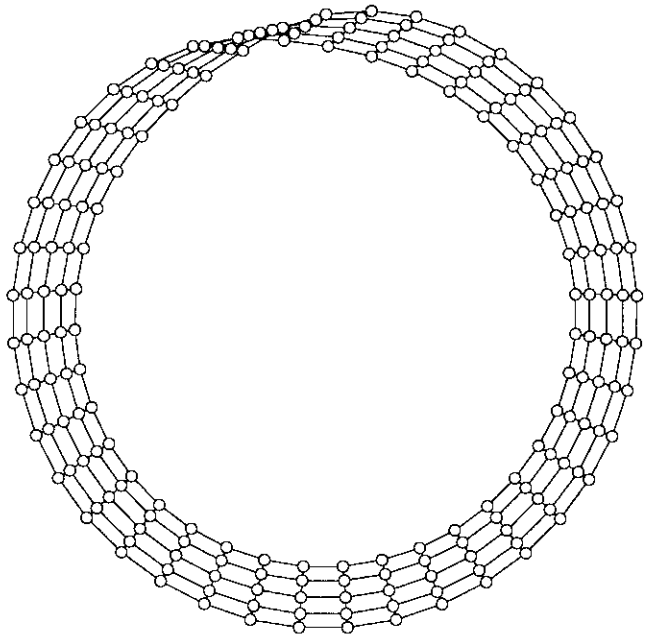

Figure 4.71: Möbius strip mesh (229.4s)

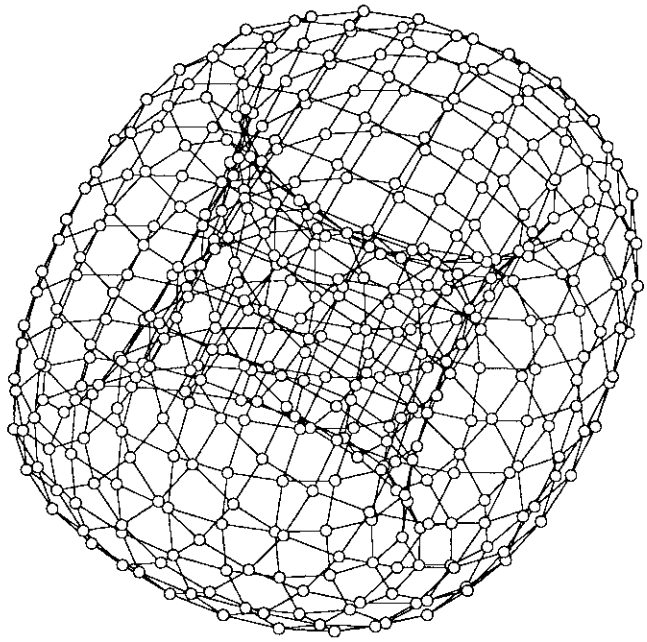

Figure 4.72: Klein bottle mesh (908.5s)

Fig.	*Pg.*	*V*	*E*	*Time*	*Command*
4.1	39	19	45	2.8	-knr 1 -kmel 1 -bt 25
4.2	41	19	45	13.4	-knr 1 -kmel 1 -kmei 1 -bt 25
4.3	42	19	45	29.3	-knr 1 -kmel 1 -kmei 1 -kner 1 -bt 25
4.4	43	19	45	29.1	-knr 1 -kmel 1 -kmei 10 -kner 1 -bt 25
4.5	43	19	45	27.3	-kcp 1 -kmel 1 -kmei 10 -kner 1 -bt 25
4.6	46	8	8	0.5	-knr 1 -kmel 1
4.7	46	8	8	0.8	-knr 1 -kmel 1 -kmei 1
4.8	47	6	7	0.3	-knr 1 -kmel 1
4.9	47	6	6	0.3	-knr 1 -kmel 1
4.10	48	6	9	0.4	-knr 1 -kmel 1
4.11	48	10	18	0.9	-knr 1 -kmel 1
4.12	50	16	30	2.2	-knr 1 -kmel 1 -kcp 1
4.13	50	16	30	6.9	-knr 1 -kmel 1 -kmei 1
4.14	51	37	68	8.9	-knr 1 -kmel 1
4.15	51	11	25	1.1	-knr 1 -kmel 1
4.16	52	11	25	6.4	-knr 1 -kmel 1 -kner 1
4.17	53	12	30	1.3	-knr 1 -kmel 1
4.18	53	12	30	0.7	-kcp 1 -kmel 1
4.19	54	10	13	0.8	-knr 1 -kmel 1
4.20	55	7	8	0.4	-knr 1 -kmel 1
4.21	55	12	14	1.2	-knr 1 -kmel 1
4.22	56	9	15	0.7	-knr 1 -kmel 1
4.23	56	6	7	0.3	-knr 1 -kmel 1
4.24	57	10	11	0.8	-knr 1 -kmel 1
4.25	57	26	31	4.4	-knr 1 -kmel 1
4.26	57	7	9	0.5	-knr 1 -kmel 1
4.27	58	7	9	2.0	-knr 1 -kmel 1 -kmei 1 -kner 1
4.28	58	8	12	0.6	-knr 1 -kmel 1
4.29	59	10	15	0.9	-knr 1 -kmel 1
4.30	59	10	15	4.5	-kcp 1 -kmel 1 -kmei 10 -kner 1
4.31	60	12	20	1.2	-knr 1 -kmel 1
4.32	61	24	40	4.0	-knr 1 -kmel 1
4.33	61	24	40	60.1	-knr 1 -kmel 1 -kmei2 10 -kner 1 -kcp 1 -it 2000
4.34	62	20	30	2.8	-knr 1 -kmel 1
4.35	62	20	30	20.0	-knr 1 -kmel 1 -kmei 1 -kner 1
4.36	63	20	30	19.8	-knr 1 -kmel 1 -kmei2 1 -kner 1

Table 4.2: Summary of Chapter 4 figures

Fig.	*Pg.*	*V*	*E*	*Time*	*Command*
4.37	64	2	1	0.1	-knr 1 -kmel 1
4.38	64	3	3	0.1	-knr 1 -kmel 1
4.39	65	4	6	0.2	-knr 1 -kmel 1
4.40	65	5	10	0.3	-knr 1 -kmel 1
4.41	66	6	15	0.5	-knr 1 -kmel 1
4.42	66	6	15	2.9	-kcp 1 -kmel 1 -kmei2 10 -kner 1
4.43	67	7	21	0.7	-knr 1 -kmel 1
4.44	67	8	28	0.8	-knr 1 -kmel 1
4.45	68	9	36	1.0	-knr 1 -kmel 1
4.46	68	10	45	1.3	-knr 1 -kmel 1
4.47	69	10	45	0.9	-kcp 1 -kmel 1
4.48	69	14	21	1.5	-knr 1 -kmel 1
4.49	70	14	21	0.7	-kmel 1 -kcp 1
4.51	71	16	32	2.0	-knr 1 -kmel 1
4.52	72	16	32	13.3	-knr 1 -kmel 1 -kner 1
4.53	72	16	32	13.1	-knr 1 -kmel 1 -kner 1 -kcp 1
4.54	74	9	9	2.9	-knr 1 -kmel 1 -kner 1 -kpl 1 -kmlv 100
4.55	74	8	13	3.4	-knr 1 -kmel 1 -kner 1 -kpl 1 -kmlv 100 -kmei 0.1
4.56	74	5	6	1.2	-knr 1 -kmel 1 -kner 1 -kpl 1 -kmlv 100
4.57	75	11	10	3.9	-knr 1 -kmel 1 -kner 1 -kpl 1 -kmlv 100 -kmei 1
4.58	76	31	30	26.1	-knr 1 -kmel 1 -kner 10 -kpl 1 -kmlv 2 -kmei 10
4.59	77	7	6	0.4	-knr 1 -kmel 1
4.60	77	31	30	6.1	-knr 1 -kmel 1
4.61	78	31	30	28.0	-knr 1 -kmel 1 -kpl 1 -kmlv 2 -kmei 1000 -kner 1
4.62	78	19	18	2.4	-knr 1 -kmel 1
4.63	79	21	20	2.9	-knr 1 -kmel 1
4.64	79	9	8	0.6	-knr 1 -kmel 1
4.65	80	17	16	1.9	-knr 1 -kmel 1
4.66	81	93	461	55.8	-knr 1 -kmel 1
4.67	82	93	461	57.8	-knr 1 -kmel 1 -kpl 1
4.68	83	402	820	917.8	-knr 1 -kmel 1
4.69	83	402	819	917.7	-knr 1 -kmel 1
4.70	84	400	800	909.5	-knr 1 -kmel 1
4.71	85	200	360	229.4	-knr 1 -kmel 1
4.72	85	400	800	908.5	-knr 1 -kmel 1

Table 4.3: Summary of Chapter 4 figures (cont.)

4.3 Discussion and Comparison

In this section, we discuss the speed and robustness of AGLO and the quality of the layouts generated and compare its performance to other layout algorithms (primarily [FR91] and [DH89]).

4.3.1 Speed

Layout speed is very important if the layout is to be done interactively. Except for our large-scale examples, all of the layouts in this chapter were performed in a minute or less, most in substantially less. With a little bit of work tuning the library, better understanding of the cooling schedule, and faster hardware, we are quite confident that all of the times for these examples will drop into the interactive range (five to ten seconds).

Fruchterman and Reingold indicate that all of their examples were drawn in under ten seconds on a SPARCstation 1 using **fdp**. In the examples where we use our two **fdp**-style aesthetics (node/node repulsion, or our centripetal repulsion, together with edge length repulsion), we get similar times. The timing data is not strictly comparable because we use a slightly faster machine (a SPARCstation ELC) and our library does not incorporate the distal-force speedup they use (see Section 3.4.2).

When we use more layout aesthetics, layout takes longer, but we can produce different styles of aesthetics that **fdp** cannot generate at all.

Davidson and Harel give an equation to estimate the run time of their algorithm, but no explicit times for their drawings. All but the smallest graphs take at least several minutes to lay out, and the authors concede that their implementation could not be considered to be of interactive speed. They suggest that the speed of their algorithm could be significantly increased by rewriting parts of the code and translating portions into assembler, but we feel certain that their method will not be competitive with the **aglo** library until they make use of gradient information.

We believe that it is important to include some very ambitious test cases to show how **aglo** would perform when stressed. At around 15 minutes, the layout time of the 400-node ellipsoid in Figure 4.68 cannot be considered interactive, but it is respectable, and it would be quite reasonable to use **aglo** to lay out such huge graphs for typesetting purposes, for example.

Our times for trees and DAGs are respectable, but definitely slower than special purpose procedural algorithms like [RT81]. It may be possible to improve the relative performance, but there will probably always be a performance penalty associated with the substantially increased generality of our method.

4.3.2 Layout Quality

The layout quality of **aglo** is similar to that of comparable layouts in [FR91] and [DH89]. Our layouts of the former's examples are quite similar to theirs, and our layouts of the latter's are of comparable quality.

We also do layout using aesthetics (such as our centripetal repulsion and "tree" aesthetics) that neither of these methods uses; there is no basis for comparison in this case. Our tree layouts still need some work because they are not yet quite as good as those of traditional tree layout algorithms, such as [RT81].

4.3.3 Robustness

If a graph layout algorithm is to be used in real applications, it must be reasonably robust, meaning that it should by default produce reasonably good layouts of whatever graphs it is given (i.e., without the user having to twiddle parameters).

An important robustness quality is *stability*. For our purposes, stability means that

1. Layout quality is insensitive to the choice of the initial state (placement).

2. Layout quality is insensitive to minor rounding errors which may exist on the machine on which the **aglo** library is run.

3. Layout quality varies in a gradual and continuous way as the layout weights and parameters are changed. Small changes in weights should not produce large changes in layout quality.

Neither [FR91] nor [DH89] give the specific algorithm parameters used to produce each of their figures, so it is difficult to say how much adjustment was necessary to generate them (i.e., to estimate the robustness of their methods). Stability is not directly addressed in either paper.

The default behavior of the **aglo** library is fairly robust but varies some with the aesthetics chosen and the complexity of the input graph. If inappropriate aesthetics are chosen (e.g., applying "tree" aesthetics to a cyclic graph), the results will be poor. For larger graphs and larger sets of aesthetics, starting temperature and cooling time need to be increased beyond the default in order to ensure good layout quality.

The stability of AGLO is excellent with respect to items 1 and 2 above, and quite good with respect to the third. (Not surprisingly, the algorithm does become unstable when cooling is done too quickly.)

In this chapter, we have presented and discussed numerous examples of the AGLO algorithm in use. Now we consider the conclusions to be drawn from this work.

CHAPTER 5

Conclusion

Panglossian—*adj.* Given to extreme optimism.

5.1 So What?

Existing methods are fast and produce reasonably good-looking layouts, so why should anyone be interested in our work?

- Our method has a stronger theoretical foundation. We present an optimization model based on utility theory that explains how graph layout aesthetics may be derived and combined to produce a quantitative evaluation of graph layouts. Furthermore, our theory suggests an objective way of evaluating the layouts produced by different algorithms.

- Our method is more general and uniform. By choosing different combinations of aesthetic functions, we can handle different kinds of graphs and do layout in many different styles, all within a single framework.

- Our method is more malleable. It provides a means for trading off between conflicting aesthetics and allows the user to control the style of layout.

- Our method is fast, at least an order of magnitude faster than [DH89], which is the only previous algorithm with the flexibility of AGLO.

In a nutshell, our approach combines the power and flexibility of [DH89] with the relative speed of [FR91] and provides robustness and a better theoretical foundation for these methods. In addition, we have developed several new layout aesthetics to support new layout styles. Using these aesthetics, we are able to produce pleasing displays for graphs where these other methods flounder.

5.2 Future Directions

There are several avenues of attack for improving the capability, robustness, and speed of the **aglo** library:

- *Better cooling schedule*: Cooling in **aglo** is currently quite naive (see Section 3.4.1). Rather than iterating for a fixed count, we should stop early when a solution is discovered. It would also be interesting to see if some of the advanced cooling schedules used in simulated annealing could be adapted for our purposes.

- *Better optimization methods*: This includes both improvements to the current algorithm, as discussed in Section 3.4.2, and altogether different algorithms, as discussed in Section 2.3.3.

- *Better aesthetics*: The **aglo** library includes a basic set of aesthetics, but it is the intention of the author that many different aesthetics supporting different layout styles, as well as more efficient versions of current aesthetics, be discovered and implemented.

- *Better understanding of aesthetic composition*: Some of the issues involved in the composition of aesthetic functions were discussed in section 2.2.3, but more work is needed. Problems in this area can cause unintuitive layout results.

- *Better handling of larger graphs*: Performance of **aglo** drops off for larger input graphs. It is possible that a divide-and-conquer approach could speed things up without unduly compromising layout quality.[1]

Up to this point, we have considered layout aesthetics to be a subjective judgement of a user or a collection of users, but there is some reason to think that some of these aesthetics are actually a result of the structure and function of the human visual system (and thus not entirely subjective). For example, Marr [Mar82] posits some low-level pattern-matching capabilities as part of the visual system, which suggests that certain layout aesthetics may be desirable because they increase the efficiency of the visual communication channel. One could imagine that in the future, when the mechanisms of vision are clearer, one could supplement user-supplied preference information with layout aesthetics specifically designed to

[1]This suggestion due to Paul Eggert.

improve the movement of information through that visual channel.

Bibliography

[Bra88] Franz J. Brandenburg. Nice drawings of graphs are computationally hard. In P. Gorny and M. J. Tauber, editors, *Visualization in Human-Computer Interaction*, pages 1–15. Springer-Verlag, May 1988.

[CE89] Mike Coleman and Paul Eggert. Language-independent software visualization tools. Unpublished, November 1989.

[CH83] Vira Chankong and Yacov Y. Haimes. *Multiobjective Decision Making.* Elsevier Science Publishing, New York, 1983.

[CH88] G. M. Crippen and T. F. Havel. *Distance Geometry and Molecular Conformation.* Research Studies Press, 1988.

[Dav87] Lawrence Davis, editor. *Genetic Algorithms and Simulated Annealing.* Research Notes in Artificial Intelligence. Morgan Kaufman, 1987.

[DH89] R. Davidson and D. Harel. Drawing graphs nicely using simulated annealing. Technical Report CS89–13, Department of Applied Mathematics and Computer Science, Weizmann Institute of Science, July 1989.

[Ead84] Peter Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[ET89] Peter Eades and Roberto Tamassia. Algorithms for drawing graphs: An annotated bibliography. Technical Report CS-89-09, Brown University DCS, 1989.

[FR91] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Software–Practice and Experience*, 21(11):1129–1164, November 1991.

[GMW81] Philip E. Gill, Walter Murray, and Margaret H. Wright. *Practical Optimization.* Academic Press, 1981.

[GNV88] E. R. Gansner, S. C. North, and K. P. Vo. DAG—A program that draws directed graphs. *Software–Practice and Experience*, 18(11):1047–1062, November 1988.

[Gre88] Leslie Greengard. *The Rapid Evaluation of Potential Fields in Particle Systems.* MIT Press, 1988.

[Gre89] Daniel R. Greening. A taxonomy of parallel simulated annealing techniques. Technical Report CSD-890050, UCLA CSD, August 1989.

[HH74] Y. Y. Haimes and W. A. Hall. Multiobjectives in water resources systems analysis: The surrogate worth trade-off method. *Water Resources Research*, 10:614–624, 1974.

[KGV82] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi. Optimization by simulated annealing. Research Report RC 9355, IBM T. J. Watson Research Center, Yorktown Heights, NY, April 1982.

[KK88] Tomihisa Kamada and Satoru Kawai. Automatic display of network structures for human understanding. Technical Report 88–007, Department of Information Science, University of Tokyo, February 1988.

[KK89] Tomihisa Kamada and Satoru Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31:7–15, April 1989.

[Lam88] J. Lam. An efficient simulated annealing schedule. Technical Report 8818, Department of Electrical Engineering, Yale University, 1988.

[Mac86] Jock Mackinlay. Automating the design of graphical presentations of relational information. *ACM Transactions on Graphics*, 5(2):110–141, April 1986.

[Mar82] D. Marr. *Vision.* W. H. Freeman, 1982.

[PFTV89] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in Pascal: The Art of Scientific Computing.* Cambridge University Press, 1989.

[QB79] Neil R. Quinn, Jr. and Melvin A. Breuer. A forced directed component placement procedure for printed circuit boards. *IEEE Trans. on Circuits and Systems*, CAS-26(6):377–388, June 1979.

[Rob87] Gabriel Robins. The ISI grapher: A portable tool for displaying graphs pictorially. Research Report ISI/RS-87-196, USC/ISI, September 1987.

[RT81] Edward M. Reingold and John S. Tilford. Tidier drawings of trees. *IEEE Trans. on Software Engineering*, SE-7(2):223–228, March 1981.

[Sor88] Gregory B. Sorkin. Combinatorial optimization, simulated annealing, and fractals. Research Report RC 13674, IBM T. J. Watson Research Center, Yorktown Heights, NY, 1988.

[Sut63] Ivan Sutherland. SKETCHPAD: A man-machine graphical communication system. *IFIPS Proceedings of the Spring Joint Computer Conference*, 1963.

[Tri88] Howard Trickey. Drag: A graph drawing system. In *Proc. of the International Conf. on Electronic Publishing, Document Manipulation, and Typography*, pages 171–182, 1988.

[Vau80] Jean G. Vaucher. Pretty-printing of trees. *Software–Practice and Experience*, 10:553–561, 1980.

[WS79] Charles Wetherell and Alfred Shannon. Tidy drawings of trees. *IEEE Transactions on Software Engineering*, SE–5(5):514–520, September 1979.

[Zel82] Milan Zeleny. *Multiple Criteria Decision Making.* McGraw-Hill, 1982.