SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 00(00), 000–000 (MONTH 1990)

Graph Drawing by Force–Directed Placement

THOMAS M. J. FRUCHTERMAN AND EDWARD M. REINGOLD

Department of Computer Science, University of Illinois at Urbana-Champaign, 1304 W. Springfield Avenue, Urbana, IL 61801-2987, U.S.A.

SUMMARY

We present a modification of the spring-embedder model of Eades[1] for drawing general undirected graphs with straight edges. Our heuristic strives for uniform edge lengths, and we develop it in analogy to forces in natural systems, for a simple, elegant, conceptually–intuitive, and efficient algorithm.

KEY WORDS Graph drawing Force–directed placement Multi–level techniques Simulated annealing

THE GRAPH–DRAWING PROBLEM

A graph $G(V, E)$ is a set V of vertices and a set E of edges, in which an edge joins a pair of vertices.[2] Normally, graphs are depicted with their vertices as points in a plane and their edges as line segments or curves connecting those points. There are different styles of representation, suited to different types of graphs or different purposes of presentation. We concentrate on the most general class of graphs: general, undirected graphs, drawn with straight edges. In this paper, we introduce an algorithm that attempts to produce aesthetically–pleasing, two–dimensional pictures of graphs by doing simplified simulations of physical systems.

We are concerned with drawing general undirected graphs according to some generally–accepted aesthetic criteria:[3]

- Evenly distribute the vertices in the frame.
- Minimize edge crossings.
- Make edge lengths uniform.
- Reflect inherent symmetry.
- Conform to the frame.

Our algorithm does not explicitly strive for these goals, but does best at reflecting symmetry, even distributution, and uniform edge lengths. Our goals for the implementation are speed and simplicity.

0000–0000/00/000000–00\$00.00 *Received 00 Month 1900*
© 1990 by John Wiley & Sons, Ltd.
Revised 00 Month 1900

PREVIOUS WORK

Our algorithm for drawing undirected graphs is based on the work of Eades[1] which, in turn, evolved from a VLSI routing technique called *force–directed placement*.[4] We begin by quoting Eades' explanation of his 'metaphor.'

> The basic idea is as follows. To embed [lay out] a graph we replace the vertices by steel rings and replace each edge with a spring to form a mechanical system, ... The vertices are placed in some initial layout and let go so that the spring forces on the rings move the system to a minimal energy state.

Eades modeled a graph as a physical system of rings and springs, but his implementation did not reflect Hooke's law,* rather, he chose his own formula for the forces exerted by the springs. Another important deviation from the physical reality is the application of the forces: repulsive forces are calculated between every pair of vertices, but attractive forces are calculated only between neighbors. This reduces the time complexity because calculating the attractive forces between neighbors is thus $\Theta(|E|)$, although the repulsive force calculation is still $\Theta(|V|^2)$, a great weakness of these n–body algorithms.[6]

Kamada and Kawai[7, 8] have their own variant on Eades' algorithm. They also modeled a graph as a system of springs, but whereas Eades abandoned Hooke's Law, Kamada and Kawai solved partial differential equations based on it to optimize layout. Eades decided that it was important only for a vertex to be near its immediate neighbors and so calculated attractive forces only between neighbors, but Kamada and Kawai's algorithm adds the powerful concept of an ideal distance between vertices that are not neighbors: the ideal distance between two vertices is proportional to the length of the shortest path between them.

Kamada and Kawai saw the graph drawing problem as a process of reducing the total energy of a system of springs connecting steel rings; by minimizing the sum of compression or tension on all the springs, the rings or vertices would be most nearly at their ideal distances from one another. They formulated the total energy of a graph as

$$E = \frac{1}{2} \sum_{1 \le i < j \le |V|} k_{ij} (|p_i - p_j| - l_{ij})^2,$$

where p_i is the position of the ring corresponding to vertex v_i, k_{ij} is the spring constant for the spring between p_i and p_j, and l_{ij} is the optimum distance between vertices

*Hooke's law is a macroscopic approximation describing the behavior of springs. We quote from an introductory physics textbook:[5]

> If the spring is compressed or extended and released, it returns to its original, or natural, length, provided the displacement is not too great. ... We see that ... that for small Δx the force exerted by the spring is approximately proportional to Δx. This result, known as *Hooke's Law,* can be written
>
> $$F_x = -k(x - x_0) = -k\Delta x$$
>
> where the empirically determined constant k is called the *force constant* of the spring.

v_i and v_j. This energy is reduced by solving a partial differential equation for each vertex to find a new location and then moving the ring whose new location minimizes the energy of the new state. The repositioning of vertices is repeated until the energy goes below a preset threshold. An important difference from Eades' approach is that only one vertex moves in each iteration, so the inner loop only needs to recalculate the contribution of that vertex to the energy of the system, taking $\Theta(|V|)$ time.

Davidson and Harel[9] also try to lay out graphs by reducing the energy of a system, but use a method having older roots in VLSI placement and other optimization problems: simulated annealing.[10, 11] Simulated annealing is a powerful, general, optimization algorithm, but it is computationally costly. Drawing a graph is restated as a problem in minimizing energy and therefore one of optimization. Applying simulated annealing requires choosing an energy function. Davidson and Harel picked a flexible function combining terms for vertex distribution, nearness to borders, edge-lengths, and edge-crossings. The weights on these terms can be varied to emphasize different aesthetic standards.

Davidson and Harel tried to be flexible in meeting different aesthetic standards and in producing the highest quality figures; they have a 'fine-tuning' option that after the basic configuration is found using simulated annealing, adjusts the it a few pixels at a time, forbidding 'up-hill' moves. We will pick up the threads of this idea later in this paper. Despite using updates of only $\Theta(|V|)$ time complexity in its inner loop, simulated annealing is extremely slow, and impractical for interactive display of graphs.

A NEW METHOD

We have only two principles for graph drawing:

- Vertices that are neighbors should be drawn near each other.
- Vertices should not be drawn *too* close to each other.

How close vertices should be placed depends on how many there are and how much space is available. Some graphs are too complicated to draw attractively at all. Our vague guidelines recall a result from particle physics:[5]

> At a distance of about 1 fm [femto-meter] the strong nuclear force is attractive and about 10 times the electric force between two protons. The force decreases rapidly with increasing distance, becoming completely negligible at about 15 times this separation. When two nucleons are with within about 0.4 fm of each other, the strong nuclear forces become repulsive. Thus nuclei do not collapse.

Consider the following analogy: the vertices behave as subatomic particles or celestial bodies, exerting attractive and repulsive forces on one another; the forces induce movement. Our algorithm will resemble molecular or planetary simulations, sometimes called n–body problems. Following Eades, however, we know we need not have a faithful simulation, and we can apply unrealistic forces in an unrealistic manner. So, like Eades, we make only vertices that are neighbors attract each other, but all vertices repel each other. This is consistent with the asymmetry of our two guidelines above.

We were inspired by natural systems such as springs and macro–cosmic gravity, but must point out that the 'forces' are not correctly named. We use forces to calculate *velocity* for every time quantum (and thus displacement, since the time of a quantum is unity), whereas true forces induce *acceleration*. The distinction is extremely important, because the real definition leads to dynamic equilibria (pendulums, orbits), and we seek static equilibria.

Psuedo-code for the algorithm is given in Figure 1. We have not said anything about the initial configuration, the input, or the output. The initial configuration could be all or partly specified, but normally vertices are placed randomly in the frame. Different functions might have been chosen for f_a and f_r

There are three steps to each iteration: Calculate the effect of attractive forces on each vertex; then the effect of repulsive forces; and finally limit the total displacement by the temperature. A special case occurs when vertices are in the same position: our implentation acts as though the two vertices are a small distance apart in a randomly chosen orientation: this leads to a violent repulsive effect separating them.

Figure 1 omits an explanation of 'temperature' and 'cooling'. The idea is that the displacement of a vertex is limited to some maximum value, and this maximum value decreases over time; so, as the layout becomes better, the amount of adjustment becomes finer and finer. For example, the temperature could start at an initial value (say one tenth the width of the frame) and decay to 0 in an inverse linear fashion. We discuss more efficient cooling functions later.

Modeling the Forces

We calculate k, the optimal distance between vertices as

$$k = C\sqrt{\frac{\text{area}}{\text{number of vertices}}},$$

where the constant C is found experimentally. Intuitively, the farther apart two vertices are, or the closer together, the more overpoweringly intolerable the current layout should be considered and the more violent the correction. If f_a and f_r are the attractive and repulsive forces, respectively, with d the distance between the two vertices then:

$$f_a(d) = d^2/k,$$

$$f_r(d) = -k^2/d.$$

Figure 2 illustrates these forces and their sum versus distance. The point where the sum of the attractive and repulsive force crosses the x–axis is where the two forces would exactly cancel each other out and this is at k, the ideal distance between vertices.

We chose the two functions above after some experimentation; they naturally suggested themselves as implementing our objectives and they resembled Hooke's Law. We experimented with several other functions that seemed to meet our guidelines, for example

```
$area$ := $W * L$; { W and L are the width and length of the frame }
$G$ := $(V, E)$; { the vertices are assigned random initial positions }
$k$ := $\sqrt{area/|V|}$;
**function** $f_a(x)$ := **begin return** $x^2/k$ **end**;
**function** $f_r(x)$ := **begin return** $k^2/x$ **end**;

**for** $i$ := 1 **to** $iterations$ **do begin**
    { calculate repulsive forces }
    **for** $v$ **in** $V$ **do begin**
        { each vertex has two vectors: $.pos$ and $.disp$ }
        $v.disp := 0$;
        **for** $u$ **in** $V$ **do**
            **if** $(u \neq v)$ **then begin**
              { $\Delta$ is short hand for the difference}
              } vector between the positions of the two vertices }
              $\Delta := v.pos - u.pos$;
              $v.disp := v.disp + (\Delta/|\Delta|) * f_r(|\Delta|)$
            **end**
    **end**

    { calculate attractive forces }
    **for** $e$ **in** $E$ **do begin**
        { each edge is an ordered pair of vertices $.v$ and $.u$ }
        $\Delta := e.v.pos - e.u.pos$;
        $e.v.disp := e.v.disp - (\Delta/|\Delta|) * f_a(|\Delta|)$;
        $e.u.disp := e.u.disp + (\Delta/|\Delta|) * f_a(|\Delta|)$
    **end**

    { limit the maximum displacement to the temperature $t$ }
    { and then prevent from being displaced outside frame }
    **for** $v$ **in** $V$ **do begin**
        $v.pos := v.pos + (v.disp/|v.disp|) * min(v.disp, t)$ ;
        $v.pos.x := min(W/2, max(-W/2, v.pos.x))$;
        $v.pos.y := min(L/2, max(-L/2, v.pos.y))$
    **end**
    { reduce the temperature as the layout approaches a better configuration }
    $t := cool(t)$
**end**
```

Figure 1. Force–directed placement

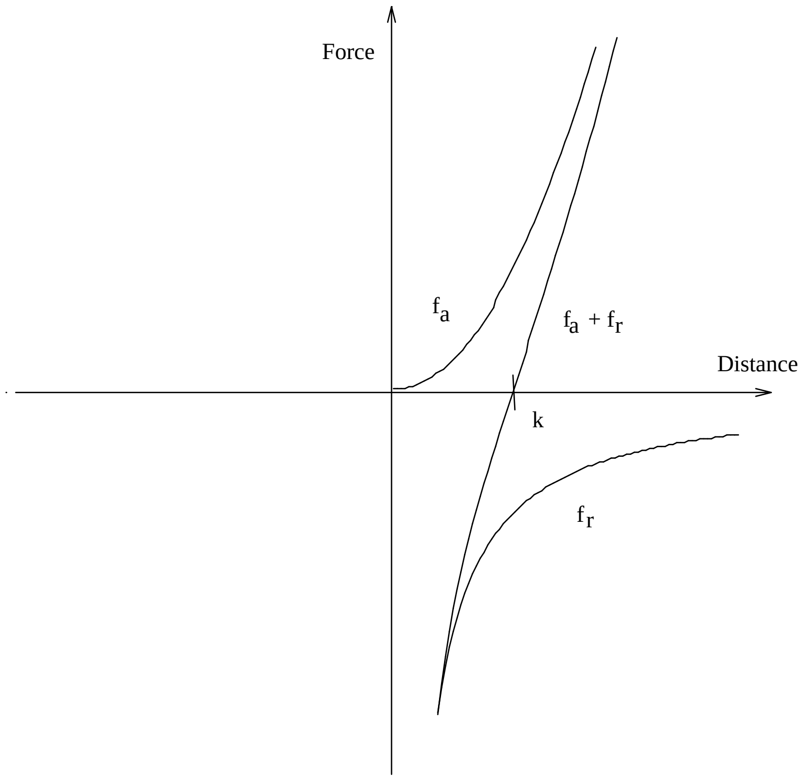

Figure 2. Forces versus distance

$$f_a(d) = d/k,$$

$$f_r(d) = -k/d.$$

This latter pair of functions, however, worked poorly for more complex graphs because it seemed unable to overcome what in simulated annealing would be called *local minima*; a vertex was less likely to move past another vertex that a bad initial placement had put in its way. Overcoming a bad configuration by moving through a yet worse configuration before reaching a better one is known in simulated annealing as *hill–climbing.* Simulated annealing probabilistically decides whether to accept a transition to an inferior configuration, but our force–directed placement deterministically always seeks the lowest ground. This would lead to getting stuck in local minima or 'valleys' more often, except that our simulation is discrete, and a vertex can shoot past another in a single time step without having to face its repulsive effects at short range, where they would be inexorable. Force laws that use higher order powers tend to give results similar to those of the quadratic functions, so we chose the easier to compute.

Eades' equations were

$$f_a(d) = k_a \log d,$$

$$f_r(d) = k_r/d^2.$$

As we will show, we achieved results similar to those of Eades, but we rejected his formula for f_a since it was inefficient to compute.

The Frame

We have to confine the graph to the frame specified by the user. Originally, we placed dummy vertices around the perimeter of the graph that exerted repulsive forces but could not move themselves. Exactly the same strategy occurred at first to Davidson and Harel. They then modeled the walls as a sloping potential barrier, by putting a term in their energy function causing the cost for a vertex being near a border to increase inverse quadratically. We chose to consider the frame an immovable object, modeling it as four 'walls', each of which exerts a 'normal' force exactly equal to the force pushing any vertex beyond it, thus stopping it like a real wall.

We have yet to implement satisfactorily an extension of this concept to borders constructed of arbitrary line segments. In drawing a graph within a concave regions, we must prevent edges as well as vertices from crossing borders. This adds another term of $\Theta(|V||E|)$ to the time complexity of our algorithm, but such arbitrary regions would be useful for reserving an island for text in the middle of a graph. For example one might try to draw a tree inside a wedge–shaped area by fixing the root of the tree at the apex of the wedge.

There are several approaches for modeling the frame after physical walls. The easiest is the 'sticky' vertex that adheres to the spot on the wall where it first strikes (an inelastic collision) and stays there until the force vector on it has only components moving away from the wall (see Figure 3). Another approach is to model striking the wall as an elastic collision but then we must determine the order in which the

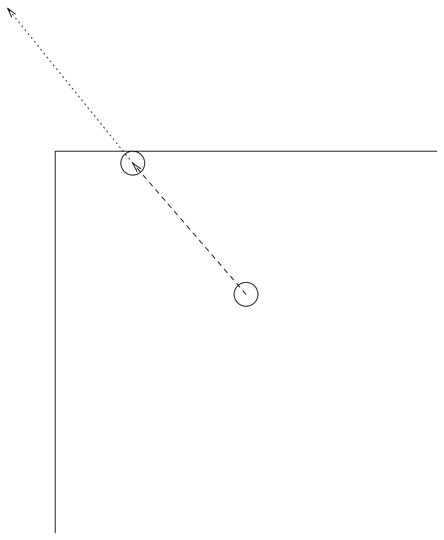
Figure 3. Inelastic collision

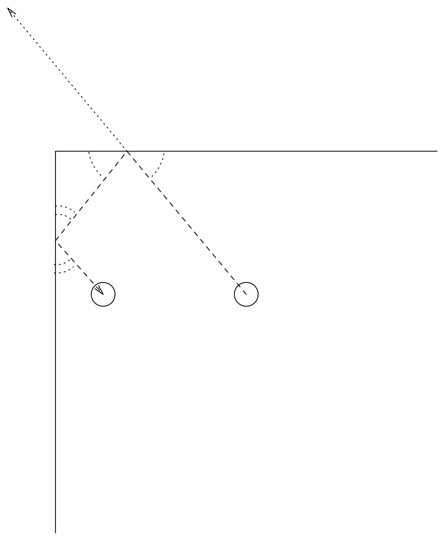
Figure 4. Elastic collision

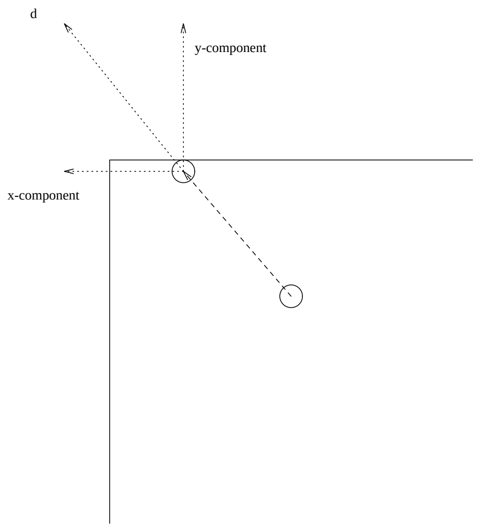

Figure 5. Halted in upward direction

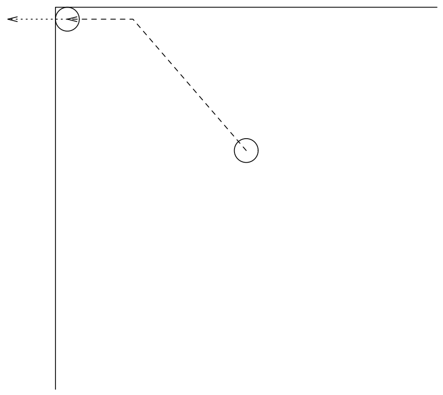
Figure 6. Stopped altogether

walls are struck—and perhaps a vertex would 'bounce' many times, entailing extensive computation. This approach is reminiscent of ray–tracing,[12] a realistic but computation–intensive method used for shading in computer graphics. See Figure 4.)

We chose to have a wall stop that component of displacement *normal* to it. This was easy and efficient to implement for a box around the graph because all the walls are orthogonal to the coordinate system. Figure 5 shows a vertex stopped in its proposed trajectory outside the frame by the top border, but in Figure 6 the vertex is allowed to slide along to the left until it reaches the left border and lodges in the upper left corner. (For borders made of arbitrary lines, we would have to compute dot products to find the component of the trajectory normal to the border. In practice, we often choose the strength of the forces so that the resulting graph is small enough that it never nears the borders, then apply a filter to enlarge the graph to fill the frame.

Speeding up the Algorithm

A time-honored improvement on simple n-body simulation is to approximate the effect of distant bodies as a single pole,[13] Doing this reduces the n-body simulation from $\Theta(n^2)$ complexity to $\Theta(n \log n)$. We need not faithfully imitate a celestial or chemical or sub–atomic system—we desire only that the results be pleasing. This allowed us make one time-saving adjustment already described: vertices are only attracted to their neighbors—this has $\Theta(|E|)$ complexity. Still, we have to compute the repulsion of every vertex from every other; this has $\Theta(|V|^2)$ complexity. The repulsive force decreases as the inverse square of the distance. Can we neglect the contribution of the more distant vertices?

In this paper, we will discuss results from applying the basic algorithm described in the previous sections, and what we will call henceforth the grid–variant algorithm, because to compute repulsion only between those vertices that are near to each other we rewrote our algorithm to divide the screen into a grid of squares, and at each iteration, each vertex is placed in its grid square and then compute the repulsive forces between it and only the vertices in nearby squares of the grid; we compute the attractive forces as usual. This is nearly equivalent (in the resulting output if not the time complexity) to applying the repulsive force to all vertices, but computing it in this way:

$$f_r = \frac{k^2}{d} u(2k - d)$$

where

$$u(x) = \begin{cases} 1 & \text{if } x > 0, \\ 0 & \text{otherwise.} \end{cases}$$

In practice, we found that the square shape of the grid boxes caused distortion and it was necessary to make the fall–off distance the same in all directions (the repulsive force is applied from all vertices within the circular area of radius $2k$ centered at the center of the vertex in question). In Figure 7, vertex v is repelled from vertex q, but not from r, which is outside the immediately neighboring squares of the grid, or s, which is considered but then rejected for being too far away anyway.

If the frame has width W and length L, then the area per vertex,

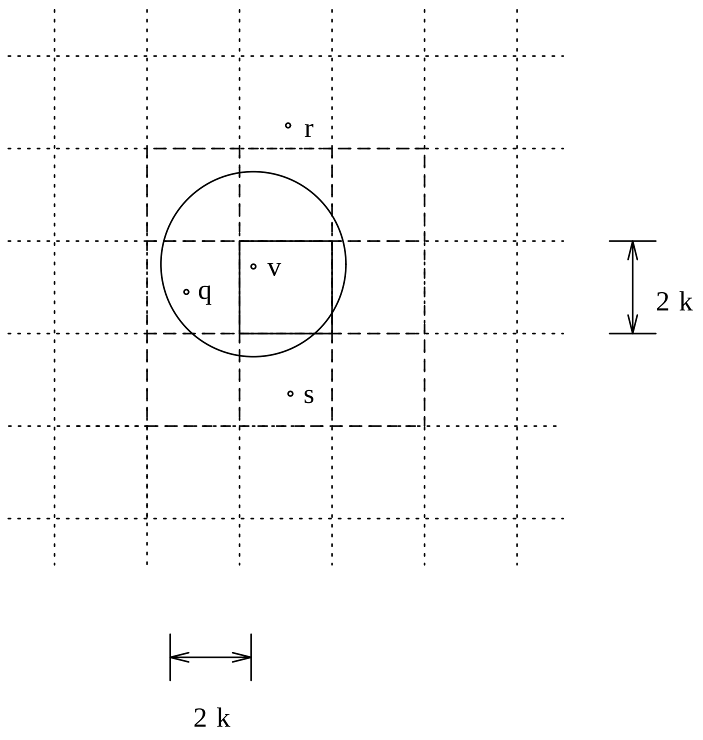

Figure 7. Calculating repulsive forces using the grid square algorithm

$$A = \frac{WL}{|V|},$$

and again

$$k = \sqrt{A}.$$

Because we ignore the repulsion of vertices beyond a distance of $2k$, $2k$ is the length of a side of a grid box.

$$\text{Number of grid boxes} = \frac{W}{2k}\frac{L}{2k} = WL\frac{|V|}{4WL} = \frac{|V|}{4}.$$

Boldly assuming the distribution of vertices is approximately uniform, then the calculation of the repulsive force becomes $\Theta(|V|)$.

An important considerations of simulated annealing is choosing a cooling schedule for the temperature T, balancing the ability to escape local minima and find the best layout with the desire for quicker termination. Our analogue to simulated annealing's temperature is limiting the maximum displacement of every vertex during an iteration, and changing that maximum displacement from iteration to iteration.

Much research in simulated annealing is devoted to speeding it up, much of the effort goes into developing better and more general cooling schedules. *Simulated sintering* is a variant on simulated annealing that is based on the hypothesis that simulated annealing wastes effort in moving from a random placement to a reasonable one, and that it would be better to generate a good initial placement by applying a more computationally efficient method and then apply simulated annealing at a low temperature to make more subtle adjustments.[14, 11] Indeed, this resembles what Davidson and Harel have done, with their computation–intensive final adjustment phase. What Grover[14] suggests doing for initial VLSI layout is Min-Cut Placement.[15] Another alternative for quickly generating an initial layout is *quenching*, simulated annealing rapidly cooling over a few iterations. To keep our system conceptually simple, we do something analogous to simulated sintering, but we will use the force–directed analogy to quenching to avoid introducing another algorithm.

We conducted some subjective experiments wherein we compared the results of drawing different graphs under different cooling schedules and using different numbers of iterations. Primarily, we compared a steady decrease in temperature with a combination of 'quenching' and 'simmering': the first phase starts at a high temperature and cools steadily and rapidly, and the second is at a constant, low temperature. The results from the latter schedule were better and required fewer iterations.

Time Complexity

In introducing our algorithm, we have shown that an individual iteration has a time complexity $\Theta(|V|^2 + |E|)$ for the basic algorithm and propose that the grid variant is $\Theta(|V| + |E|)$, but exactly how many iterations are necessary to lay out a graph attractively. There is much *ad hoc* argument in the literature when the authors try to explain when the algorithm terminates. Eades simply asserted that "Almost all graphs reach a minimal energy state after the simulation step is run 100 times . . . ." Davidson and Harel, and Kamada and Kawai based conditions for termination on their explicit representations of state energy, but they could offer little analysis how seen those conditions would be met; indeed Kamada and Kawai did not include the number of iterations in their outermost loop in doing their time complexity analysis! We too can offer little justification for the number of iterations, although we experimented with making it a function of $|V|$ or $|E|$. We did suspect, as did Davidson and Harel, that choosing a better cooling schedule could make a dramatic difference, and we discussed this earlier.

The algorithms applied a conscious concept of state energy terminated when an energy threshold is reached and presumably the graph was in a final state. In contrast, our algorithm's termination is guesswork, as was Eades'. However, the intermediate output of the program can be saved and used as an initial configuration if the layout comes out "half-baked", and then more simmering can be used to fix the layout for less cost than starting from scratch. Also, saving the intermediate layout facilitates other forms of incremental layout, such as adding edges or vertices to a graph already laid out.

IMPLEMENTATION

We wanted to have a simple, coherent method, not a "bag–of–tricks." For example, we rejected planarity testing and planarization even though they are efficient. We avoided slow, complicated techniques such as simulated annealing, and we tried to make our implementation fast enough to be interactive for graphs of moderate size.

We constructed a flexible experimental framework, to let us test many different models of forces easily. We could specify arbitrarily complex functions using addition, subtraction, multiplication, division, mod, log, square root, exponentiation, and the unit step function in an **awk**–like syntax[16] on the command line.

We implemented our algorithm in C++.[17, 18] We decided early on to use integer arithmetic for speed. This often required that expressions be carefully ordered to preserve significance, but it paid off. Following the advice of Jon Bentley,[6] we kept expensive operations such as square roots to a minimum; often a square will do in the place of a square root. Experimentation was aided by the 'little language' used to specify the forces and other parameters. Many an idea was evaluated this way, and then hard–coded if it proved its worth (such as the grid approximation and sintering).

There are several of variants of the lay out program. The basic version is called **Nature**, because of the analogy between our layout algorithm and the forces of nature. We have a variant that lays out in three dimensions called **Nature3d**, and the one that implements the Order($|V|$) inner loop is called **Naturev.** All three take text input and produce text output. The input file format is the same as that of the output, except that the input will not necessarily have coordinates for the points. If it does, those specify the initial configuration, otherwise vertices are placed randomly. So if one pass through **Nature** fails to result in a pleasing lay–out, one can resubmit the graph, but now some computation is saved because one can reuse the output of the first attempt. The difference between the variants is a matter of linking to different object files.

AN EXTENDED EXAMPLE — THE PENTAGONAL PRISM

To demonstrate our algorithm in action, we show in Figures 8 and 9 every step in laying out a pentagonal prism. The first frame shows all the vertices in their random initial positions. The first twelve frames show quenching; the second twelve, simmering. If a figure in this paper has a box it, it represents the border is part of our algorithm. Most others have no box, which means they have been scaled by a filter.

SYMMETRIC GRAPHS

The most natural and pleasing results from our algorithm may be those from highly symmetrical graphs in which the underlying force law paradigm is most evident. One graph that appears to be ubiquitous in papers on graph layout is K_5 (Figure 10). Our format will be identify all graphs in the figure captions by giving the figure number in the original paper and the citation. Some are specially identified with 'as drawn by" or 'as proposed by', these are reproductions of those figures exactly as they appeared in the cited paper; the former type of caption identifies those representations produced by the algorithm discussed in the cited paper, while the latter those which were merely proposed by the authors, but not generated by their algorithm.

K_5 and the other complete graphs K_4 (Figure 11), K_6 (Figures 12 and 13), and K_8 (Figure 14) illustrate the balance of attractive and repulsive forces in equilibrium. Note that K_4 is planar but has not been represented so; the planar representation requires a curved edge and it is arguably not as natural to humans as the representation generated here. The complete graphs get smaller as the degree gets higher: the higher the density of edges, the shorter the distances at which equilibrium is reached. Our calculation of the ideal distance between vertices works as we ideally described it only in the simple case of K_2 Figure 15, but since all our figures have been through the enlarger filter, they all are of about the same size.

Some more examples of simple, planar or nearly planar, symmetric graphs are in the three in Figure 16 and the one in Figure 17, which first appeared in Kamada and Kawai,[7] and Figure 18, which appears in Davidson and Harel.[9] The graphs presented by Kamada and Kawai are especially easy and our renderings are essentially the same as theirs.

Another highly symmetric graph is $K_{3,3}$ (Figures 19 and 20). The first representation, while acceptable, is arguably not as good as the other ones because one dimension of symmetry is missing. The Heywood Graph (Figure 21), was taken from Davidson and Harel, but they did not achieve this with their algorithm: their layout is Figure 22. Still, it is probably preferable to our Figure 12. The failure of both methods is probably because the symmetric rendering requires long distances between neighbors.

Other problems that our algorithm encounters, even with symmetric graphs, are illustrated by Figure 24 which was the first figure of Davidson and Harel.[9] The original, Figure 25 makes the problem a bit clearer. The graph *is* planar and Davidson and Harel generated a planar representation, but the repulsive forces from the inner vertices prevent us from achieving the more attractive, yet equally stable, configuration.

The centerpiece of Davidson and Harel's paper was Figure 26. Our rendition, Figure 27, is faulty in one respect: Davidson and Harel drew the outermost vertices p, q, n, and o between their two neighbors to make the graph perfectly planar. As a comparison, using the time complexity formula Davidson and Harel provide, they

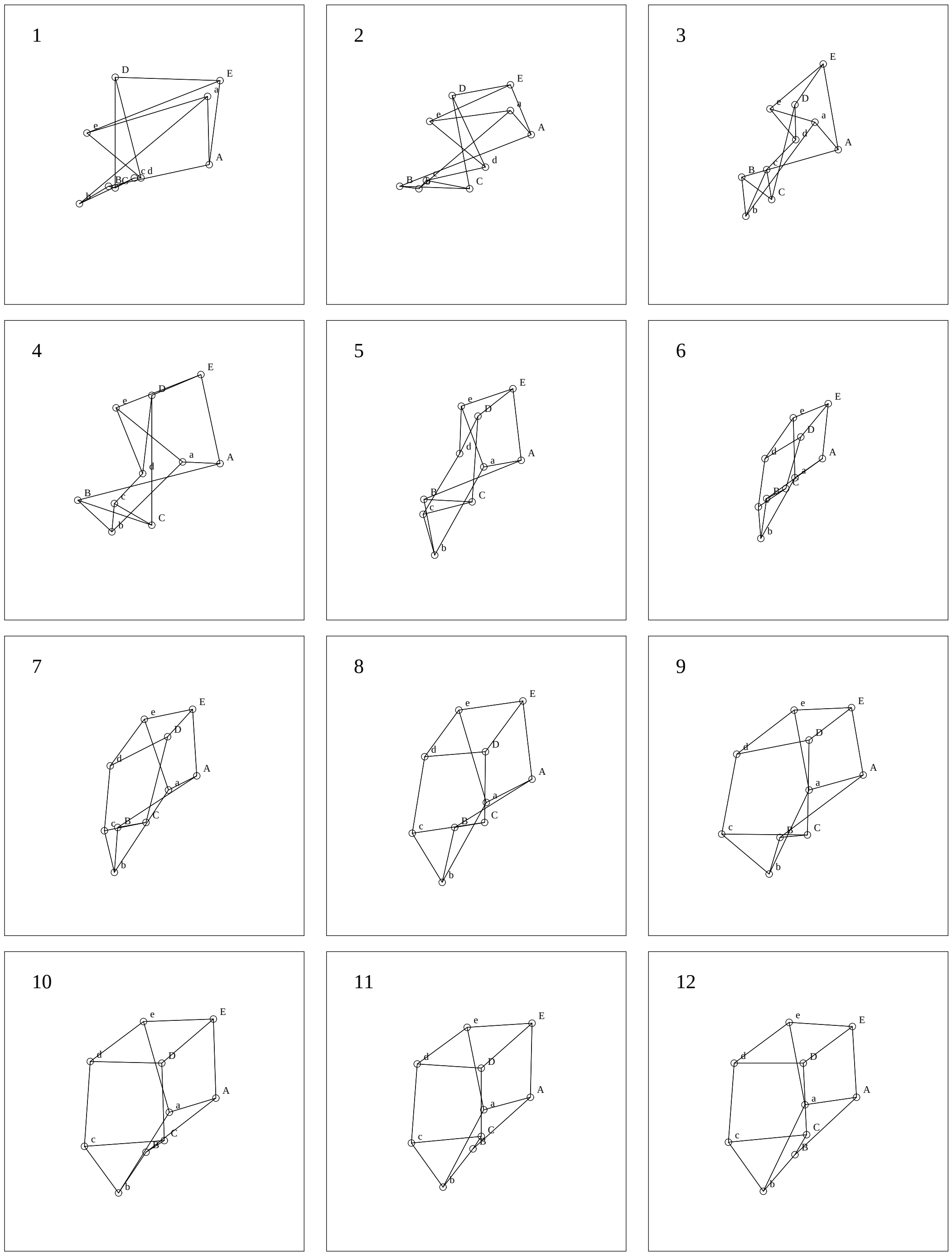

Figure 8. Quenching

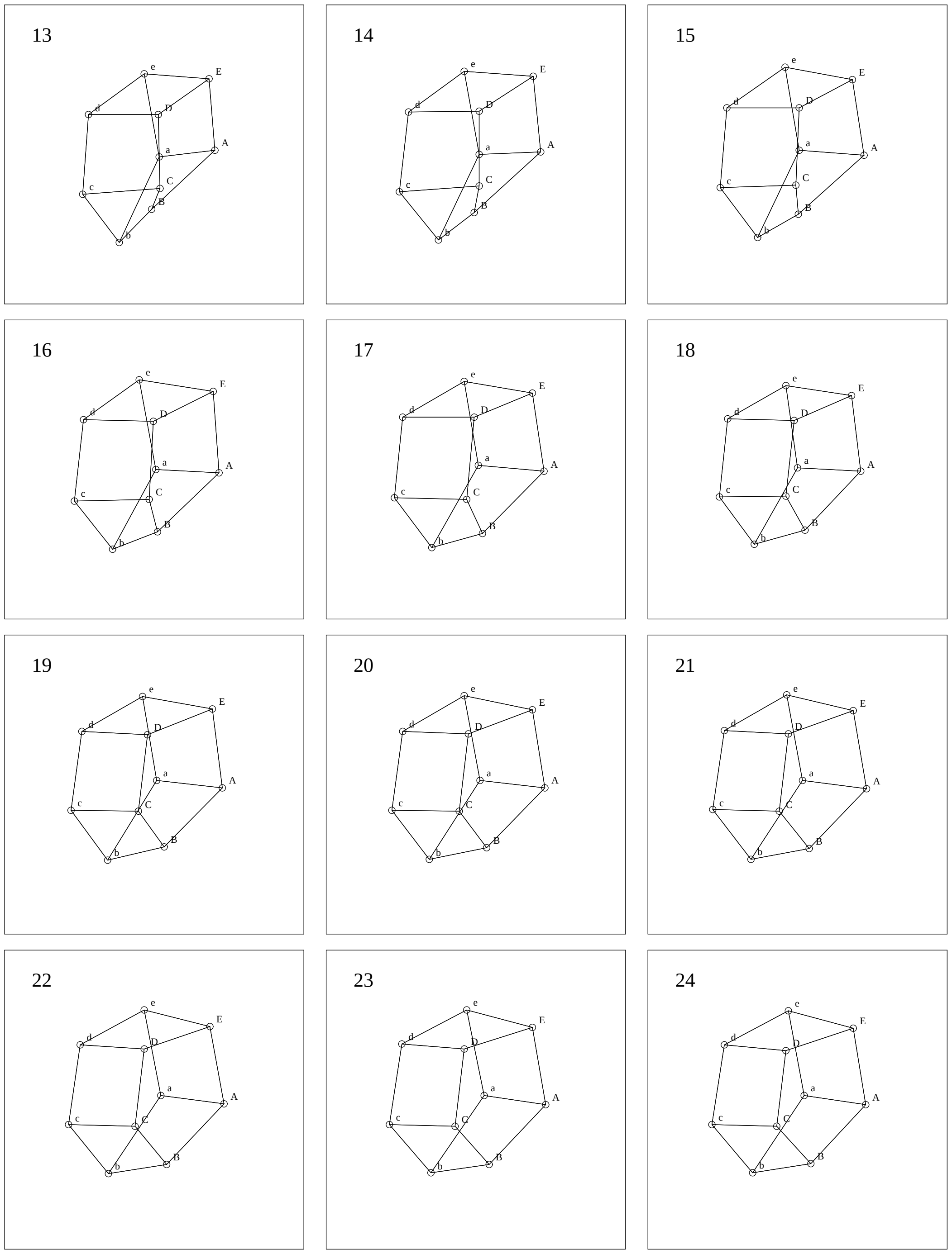

Figure 9. Simmering

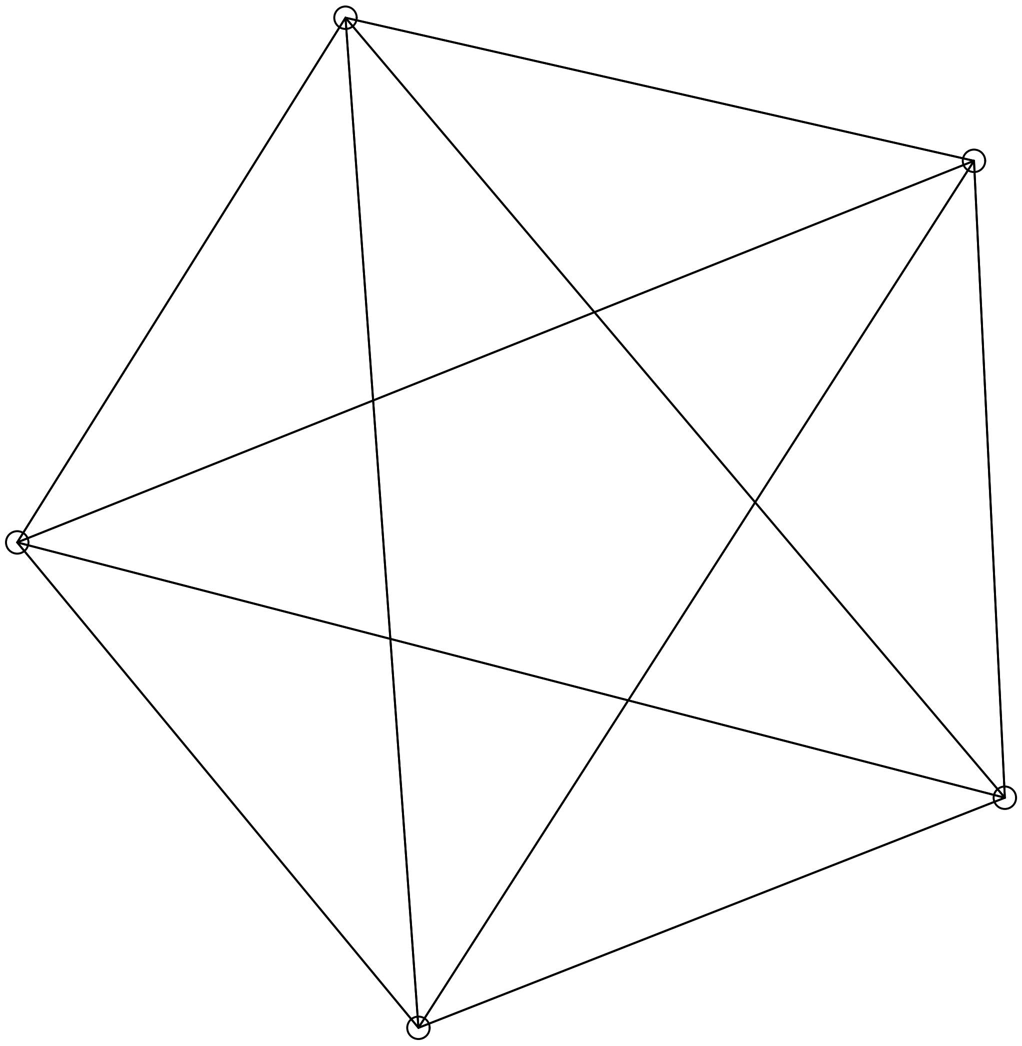

Figure 10. K_5

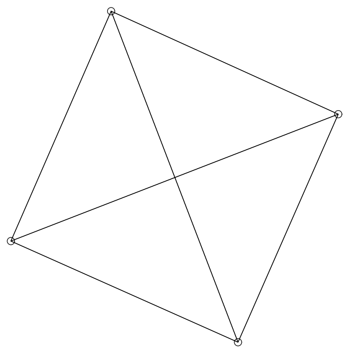
Figure 11. K_4

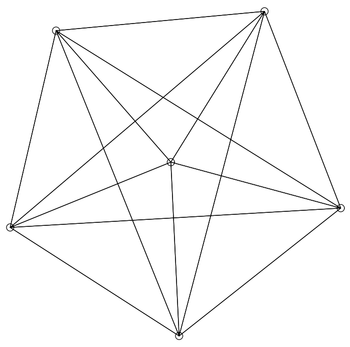
Figure 12. K_6

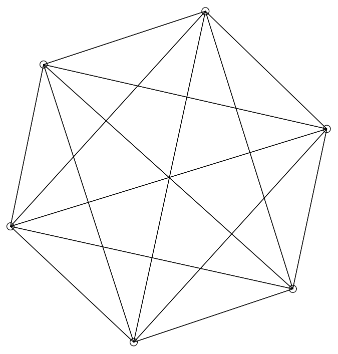
Figure 13. K_6

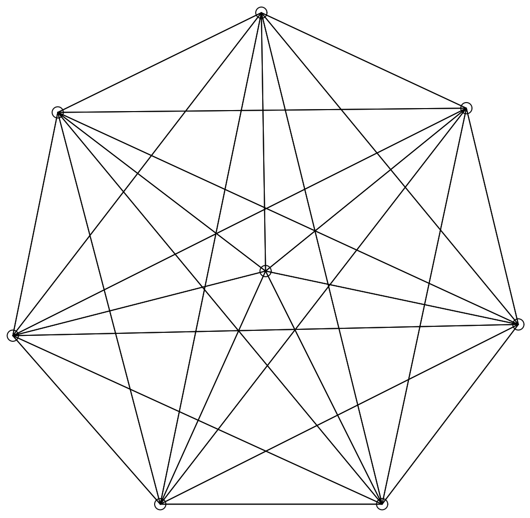
Figure 14. K_8

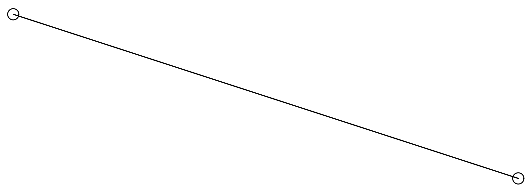
Figure 15. K_2

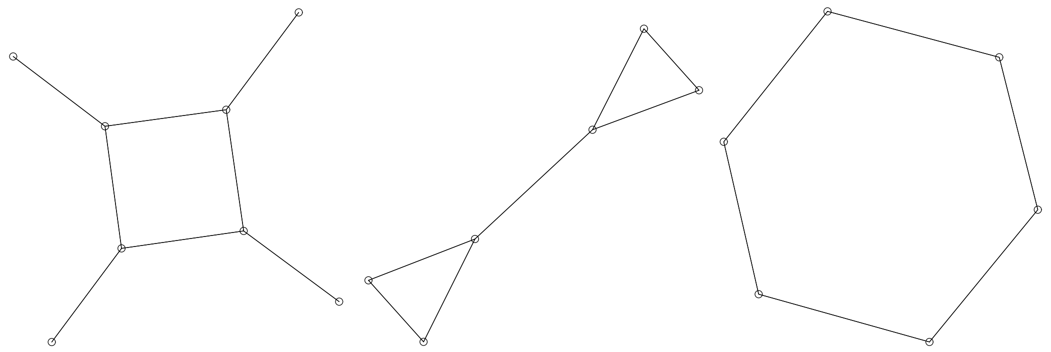
Figure 16. Graphs in Figures 6(a), 4, and 3 from Kamada and Kawai[7]

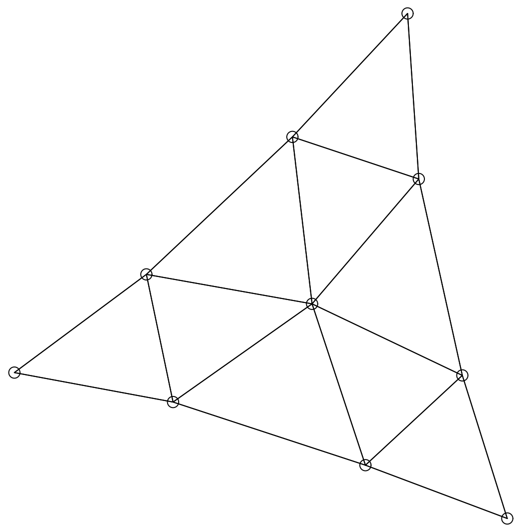
Figure 17. Triangulated Triangle (Graph in Figure 6(c) from Kamada and Kawai[7])

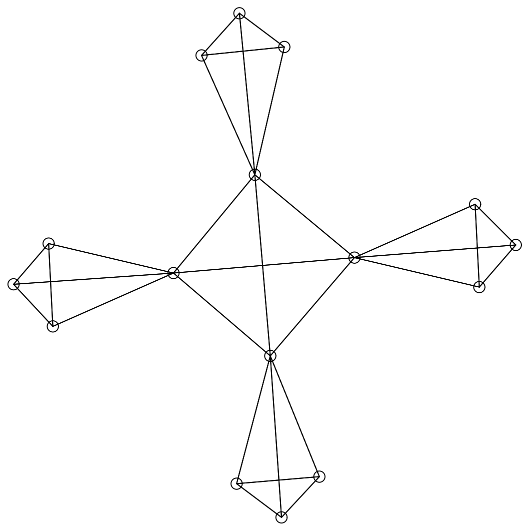
Figure 18. Graph in Figure 16 from Davidson and Harel[9]

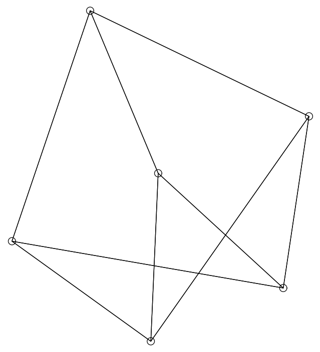
Figure 19. $K_{3,3}$

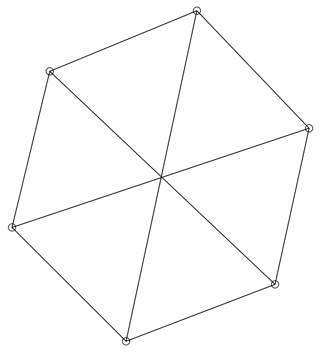
Figure 20. $K_{3,3}$

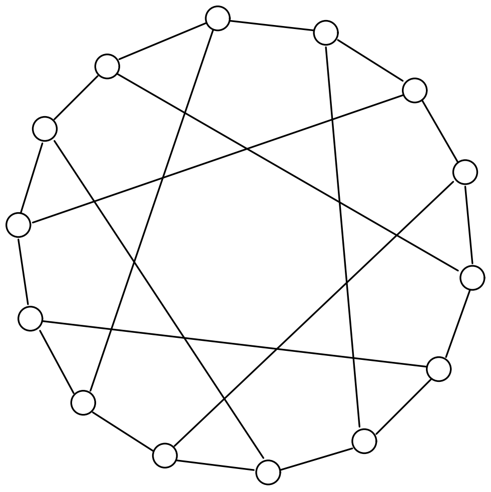

Figure 21. Figure 18(a) from Davidson and Harel[9] as proposed by Davidson and Harel

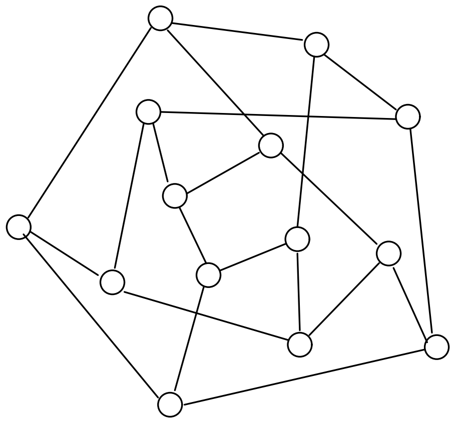

Figure 22. Figure 18(b) from Davidson and Harel[9] as drawn by Davidson and Harel

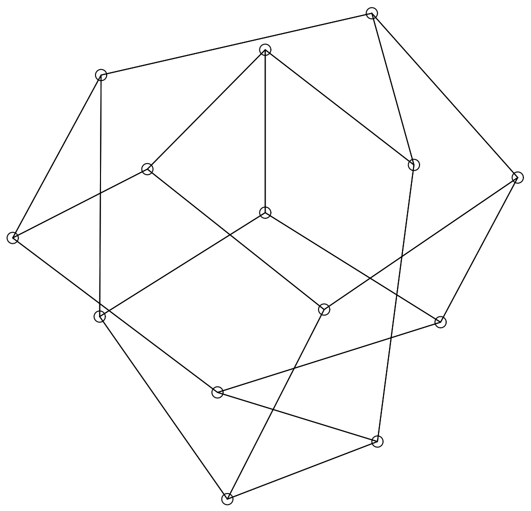

Figure 23. Heywood Graph (Graph in Figure 18 from Davidson and Harel[9])

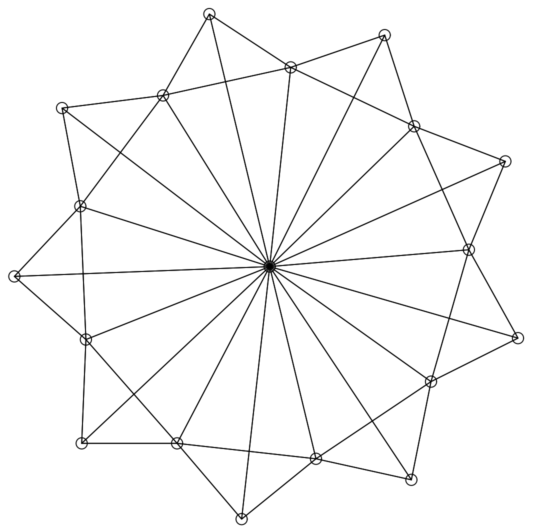

Figure 24. Graph in Figure 1 from Davidson and Harel[9]

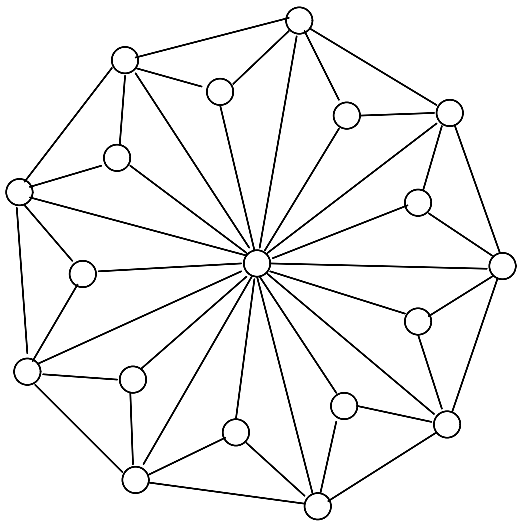

Figure 25. Figure 1 from Davidson and Harel[9] *as drawn by Davidson and Harel*

would require 10 minutes to generate the figure, but our algorithm takes 30 seconds on a comparable machine.

Figure 28 and 28 displays two different layouts of an icosahedron missing one vertex and its incident edges; the first layout looks something like the traditional solid representation of the icosahedron (but with a vertex missing) and the other layout is planar but some edges are hard to see. Of all the methods we have discussed, only the simulated annealing of Davidson and Harel explicitly attempts to avoid occluding vertices with edges. Figure 29 is the icosahedron. The layout is extremely attractive, although not planar (the icosahedron is planar).

PLANAR GRAPHS

In this section we produce, with little comment, some of the asymmetrical figures from other papers we found easy to draw. Nearly all these graphs are planar. The layout we produce for each figure is nearly identical to the layout from the paper we cite as the source of the graph.

Kamada and Kawai demonstrated that isomorphic graphs, such as those in Figures 36 and 37, will have identical layouts, (through rotation and reflection), while similar but not isomorphic graphs, such as in Figures 38 and 39 will have similar but distinct layouts. Our results agree with this.

A simple binary tree is represented in Figure 40. As do many other algorithms, our algorithm draws trees as expanding radially from the root. Some of the problems with our algorithm, which does not directly penalize edge crossings, are illustrated by Figure 41. Here, the children of the second level cross each other because that packs the vertices in the frame. A similar problem that occurred in an early implementation of our algorithm, but to a more serious degree, was the blocking affect illustrated in Figure 42. Vertices *aaab*, *aaaba* and *aaabb* are attracted to ancestor *aaa*, but cannot

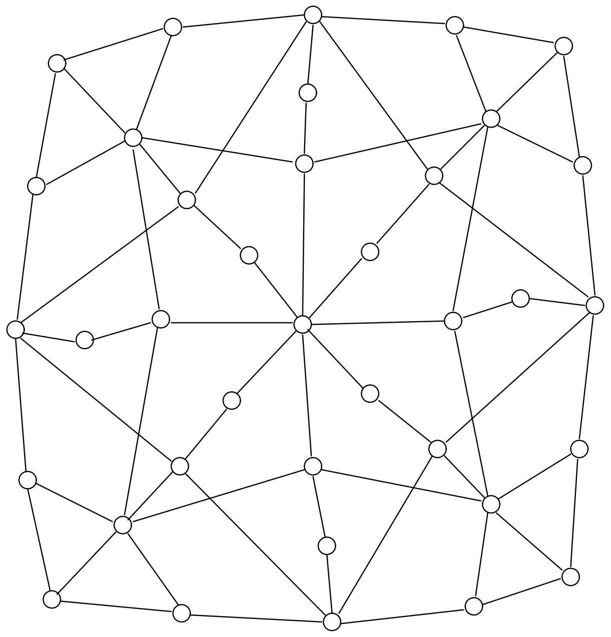
Figure 26. Figure 20 from Davidson and Harel[9] as drawn by Davidson and Harel

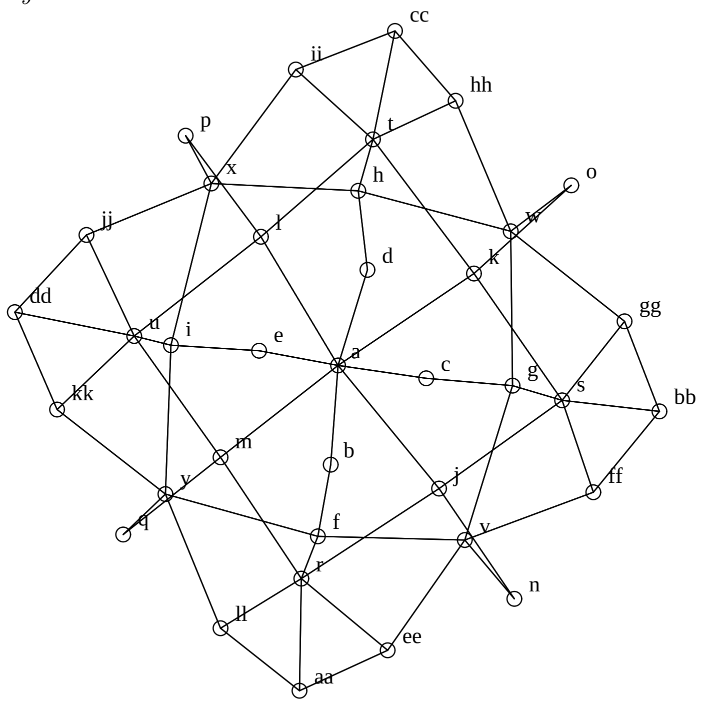

Figure 27. Graph in Figure 20 from Davidson and Harel[9]

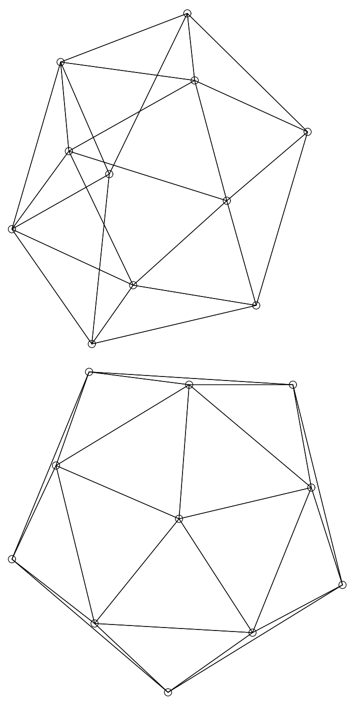

Figure 28. Two Layouts of Icosahedron Variant

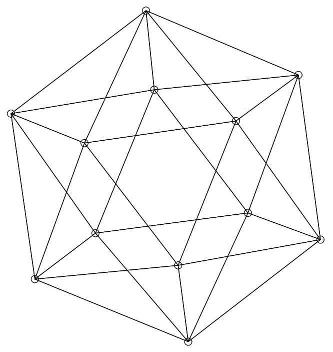

Figure 29. Icosahedron

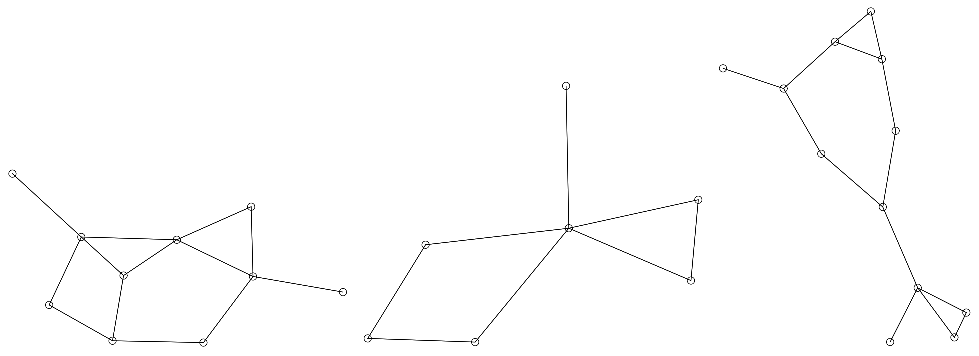

Figure 30. Graphs in Figures 7(c), 7(a), and 7(d) from Kamada and Kawai[7]

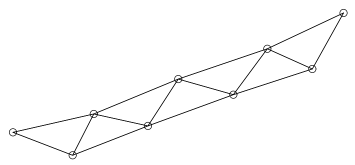
Figure 31. Graph in Figure 2(b) from Eades[1]

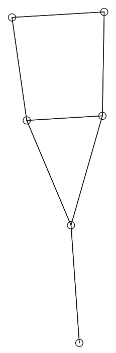
Figure 32. Graph in Figure 5(c) from Eades[1]

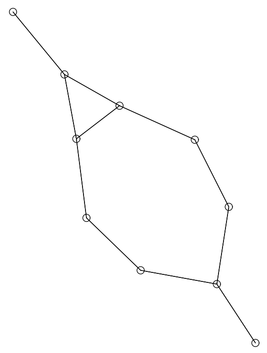
Figure 33. Graph in Figure 5(b) from Eades[1]

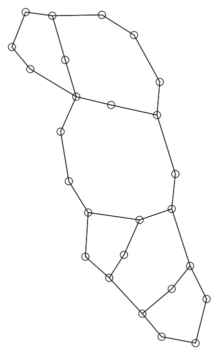
Figure 34. Graph in Figure 5(a) from Eades[1]

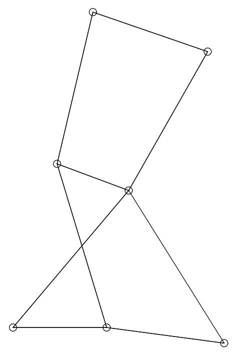
Figure 35. Graph in Figure 6(c) from Eades[1]

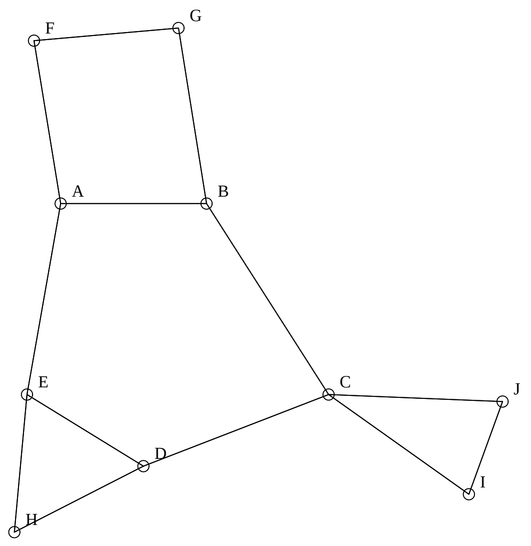

Figure 36. Graph in Figure 8(c) from Kamada and Kawai[7]

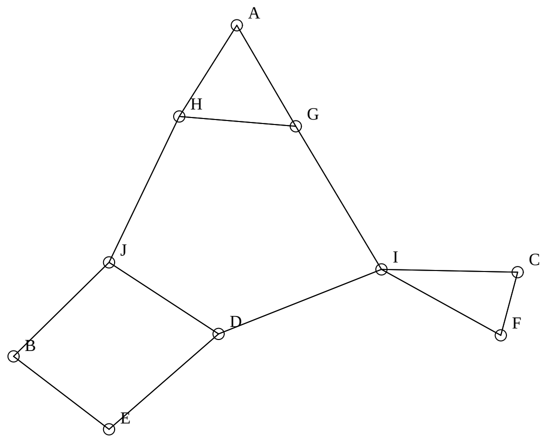

Figure 37. Graph in Figure 8(d) from Kamada and Kawai[7]

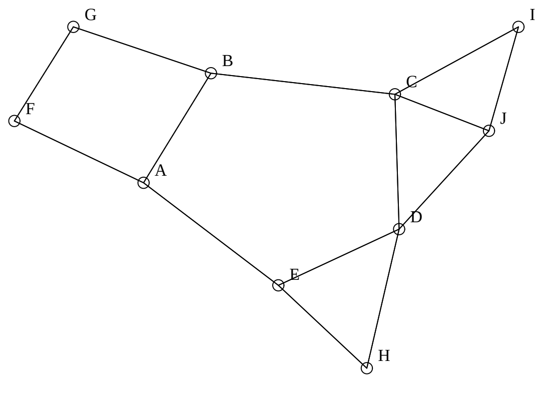

Figure 38. Graph in Figure 9(a) from Kamada and Kawai[7]

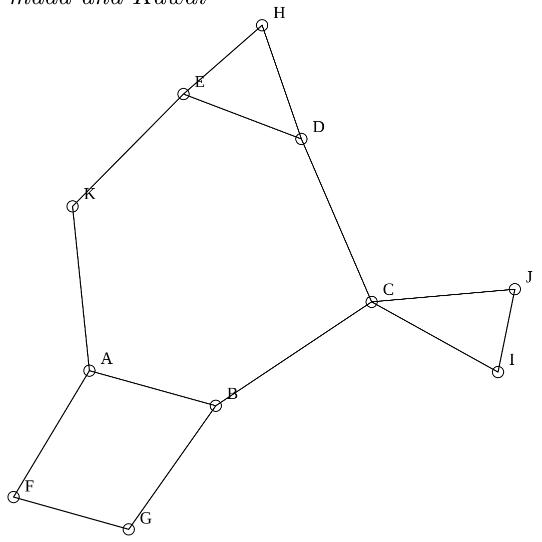

Figure 39. Graph in Figure 9(b) from Kamada and Kawai[7]

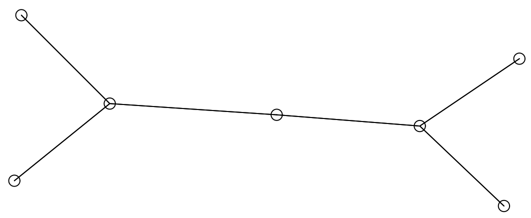
Figure 40. Binary Tree

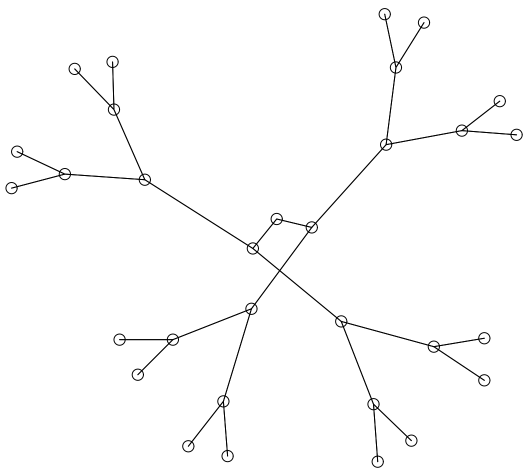
Figure 41. Tangled Binary Tree (Graph in Figure 14(a) from Davidson and Harel[9])

overcome the potential barrier posed by vertices *ab* and *aba*. We tried to find an analogy in nature that would suggest a way that such a blocking force could be overcome. Quantum jumps by electrons came to mind, but the heuristic we used to solve this problem was to turn off the repulsive force every fifth iteration. This heuristic is helpful but it can't overcome a problem that involves many vertices such as in Figure 41. We suspect that if we apply a multi–grid technique which allowed whole portions of the graph to be moved, it might be of some help, but such a technique might be more suitable for an optimization algorithm such as simulated annealing.

Four more examples of trees, some of which are not binary, are Figures 43, 44, 45, and 46. For the latter two, one is not sure where the root is, and indeed, because our current implementation gives no indication of the direction of edges, these are not really trees but undirected planar graphs with no circuits, and in the other examples we were deceived into believing we knew which vertex was the root.

LAYOUTS THAT APPEAR TO BE THREE–DIMENSIONAL

Three-dimensional objects often must be represented by two–dimensional projections. The appearance of three-dimensionality is a product of human perception. An example of graph that looks three-dimensional when drawn by our algorithm, and by the other algorithms we have discussed here, is the cube of Figure 47. Although it can be drawn as a planar graph (the planar representation can also be viewed as the head–on perspective projection of an opaque cube), rarely would a human being want to do so. We take a further step and attempt to draw a hypercube, which is shown in Figure 48. The layout is good, but it is difficult to picture the original four–space object.

Davidson and Harel tuned their algorithm by changing the weighting of various components. In an attempt to produce a planar rendition of the cube, they experi-

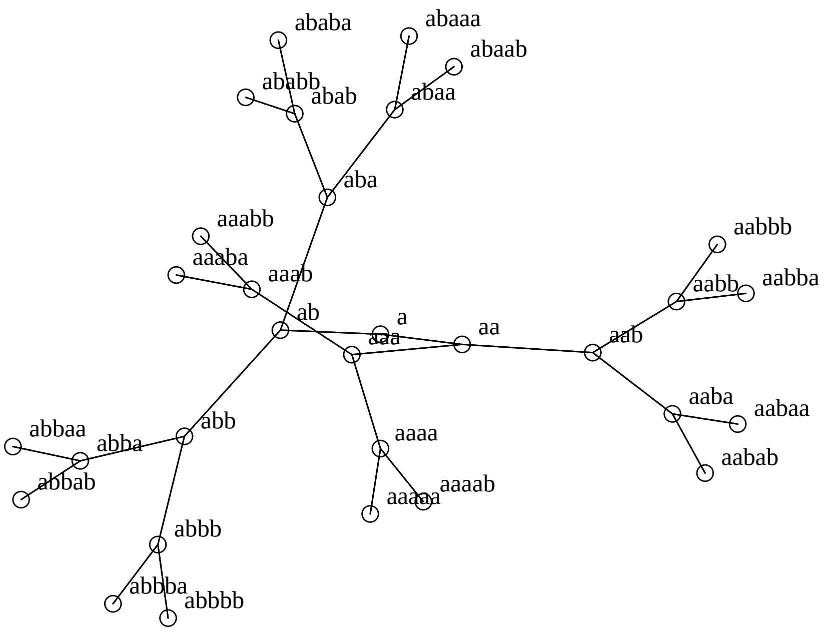

Figure 42. Example of Potential Barrier

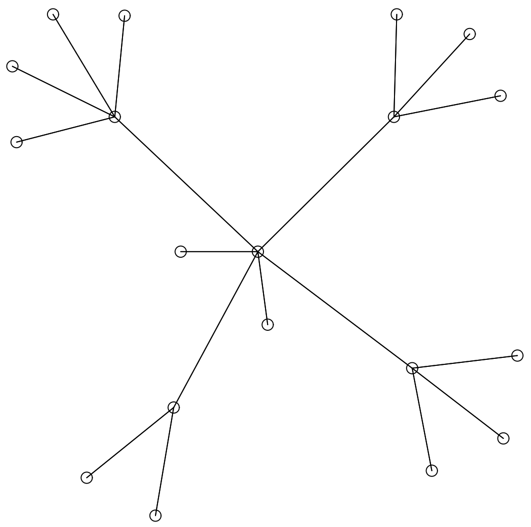

Figure 43. Tree (Graph in Figure 4(a) from Eades[1])

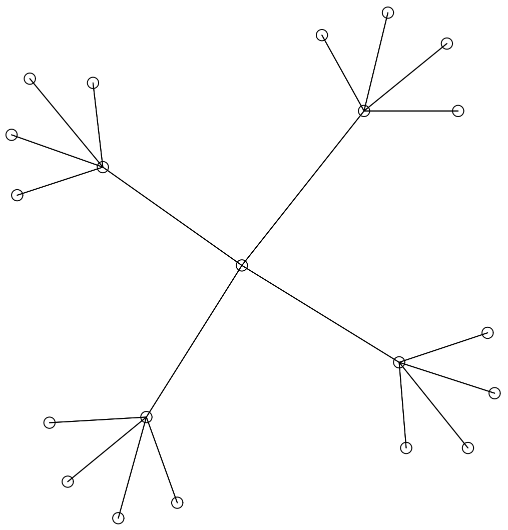

Figure 44. Tree (Graph from Figure 14(b) from Davidson and Harel[9])

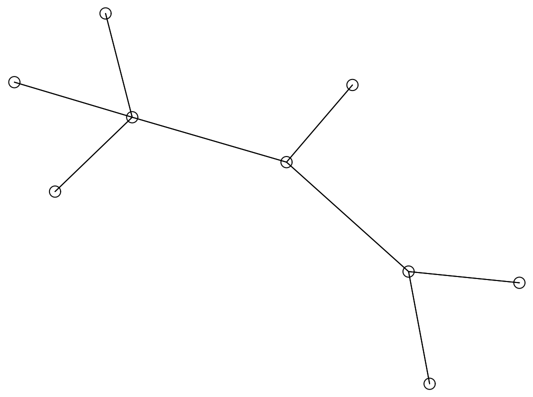
Figure 45. Graph in Figure 7(b) from Kamada and Kawai[7]

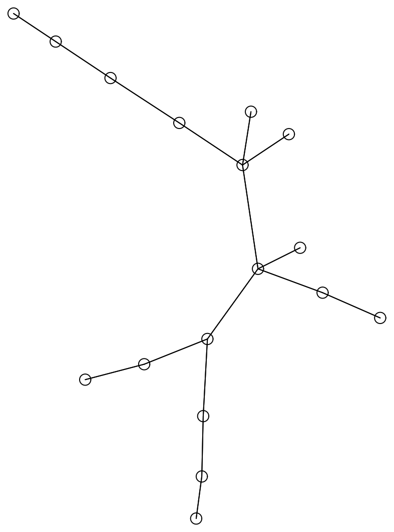
Figure 46. Unbalanced Tree (Graph in Figure 4(b) from Eades[1]*)*

mented with weighting more heavily the term penalizing edge–crossings and were able to draw the cube as two squares, one inside the other, but with the inner square rotated with respect to the outer one because that made the edge lengths more uniform. Two other graphs that are planar but are usually drawn as three–dimensional are the pentagonal prism (Figure 49) and the twin cubes (Figure 50). They also tried to make these graphs planar, but they failed in the latter case. Figure 51 is the planar layout they sought. Figure 52 was the result of weighting the edge crossing component more favorably, but it has just as many crossings as 53 the layout produced by their default settings. However, the former does give the effect of being a perspective projection.

Davidson and Harel did an extended example upon Figure 54, except that in their paper, it appeared planar and like a closing shutter (Figure 55). Once again, our representation appears to be an oblique parallel projection of an object, whereas Davidson and Harel have something like a perspective projection (once again, their algorithm adds a 'twisted' effect to keep edge lengths nearer uniform).

Kamada and Kawai, and Davidson and Harel, discussed how their algorithms draw isomorphic graphs in the same way, though perhaps rotated and reflected. Apparently, both papers were referring to rotation and reflection in the plane, but we were struck by how our algorithm appears to rotate graphs that are three-dimensional–looking graphs in three–space. An example is the dodecahedron of Figure 56 and Figure 57. The algorithm produced different results because of different initial configurations. The first figure is not nearly as acceptable in appearance as the other two.

THREE–DIMENSIONAL LAYOUT

Our layouts that appeared three-dimensional led us to experiment with doing the layout in three dimensions and projecting the result obtained onto two dimensions. What we hoped to obtain from this was more control over the final image. As we saw

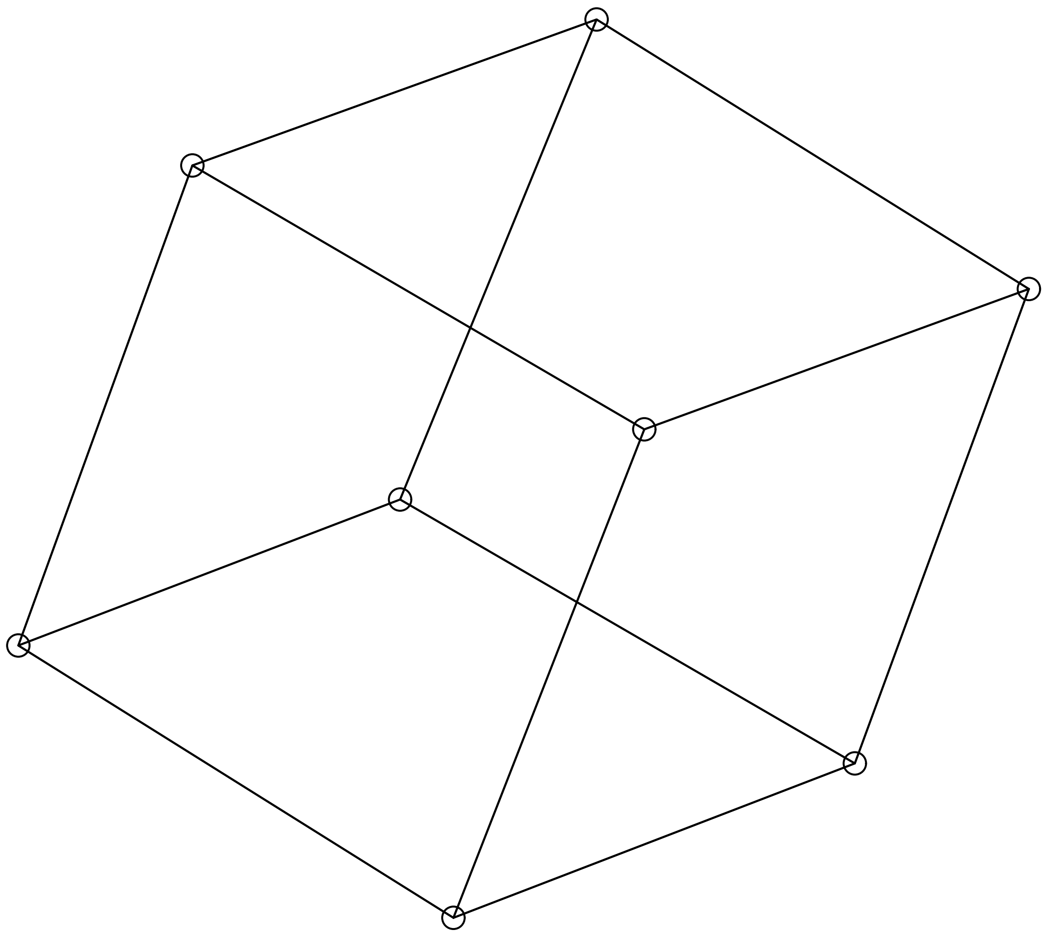

Figure 47. Cube

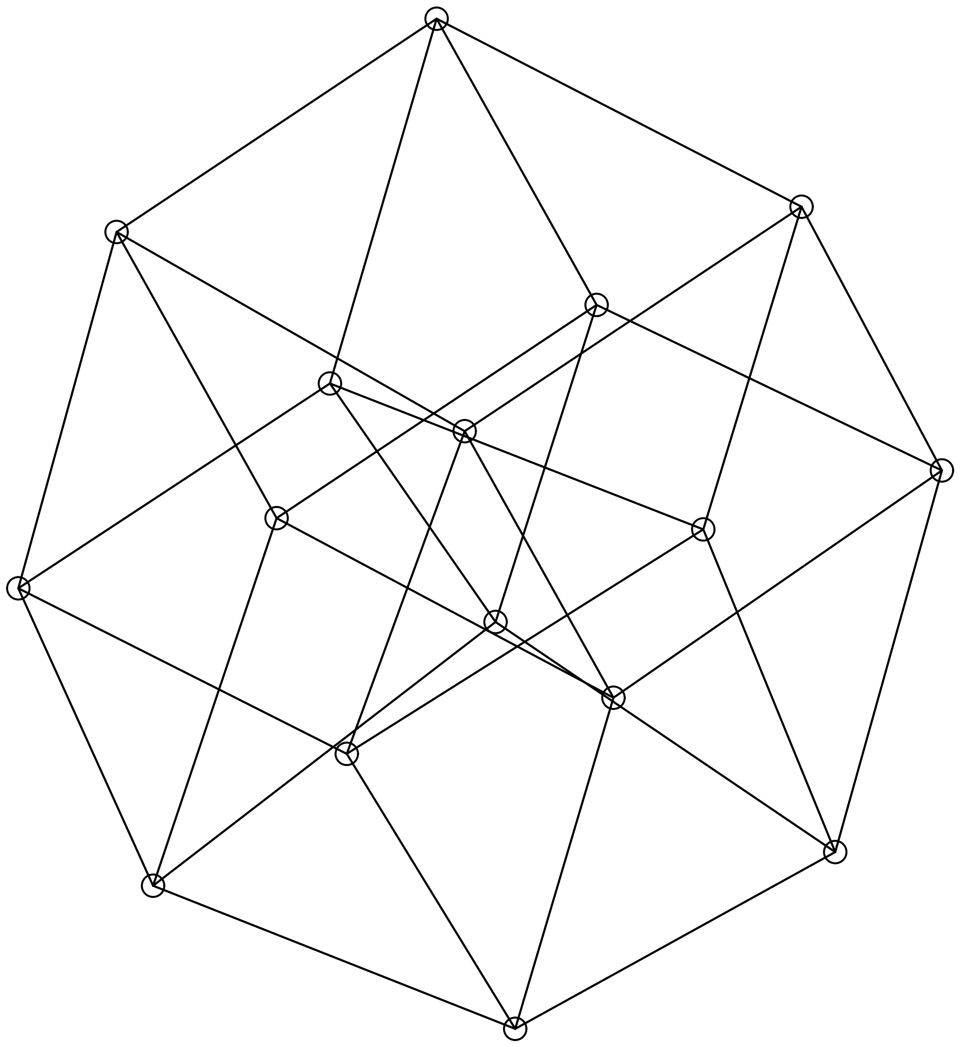

Figure 48. Hypercube

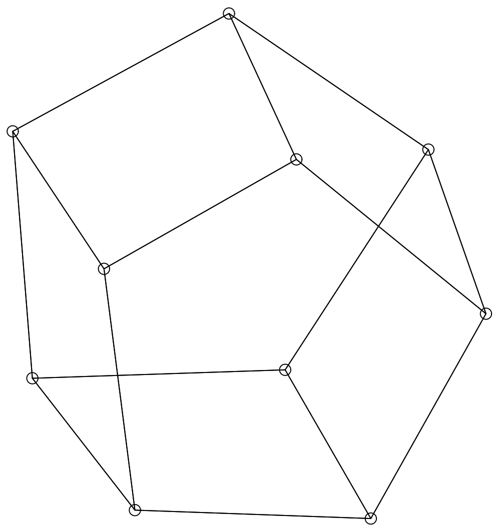

Figure 49. Pentagonal Prism

with the dodecahedron, the algorithm can generate different 'projections' of the same object, depending on initial placement and the cooling schedule, and the projection might be 'bad' because it occludes edges and vertices and hides planes that are being viewed with a view normal vector that is parallel, or nearly so, to those planes. We created another filter to do projections of vertices laid out in three dimensions onto two, with options to choose the view reference point, view normal vector, and view up vector, with choice of parallel or perspective projection (see Hearn and Baker[12] for definitions of these terms).

We now show some examples repeated from previous sections, but now laid out in three dimensions. Each example is a triplet of projections from the x, y, and z–axis.

The first set of examples are the cube (Figure 58), the mesh (Figure 59), and the pentagonal prism (Figure 60). Examining the dodecahedron (Figure 61), the icosahedron (Figure 62), and the twin cubes (Figure 63), we see in the potential for a good projections is there, but there is no guarantee that any two or three projections will necessarily be better than the one we can get from laying out in two dimensions. Calculating a projection from the layout has little cost but currently requires what we believe is too much intervention by the user to choose a projection. A better interface would allow the user to interactively manipulate the object in three dimensions. Kamada and Kawai [19] discusses the problem of selecting the 'general position' for viewing a three dimensional object so that the maximum shape information is obtained, and presents an algorithm.

When Davidson and Harel tried to find a planar lay out for the icosahedron, they also tried Figure 64, the icosahedron missing one vertex and its incident edges. The nature of this modification is clear from our three–dimensional layout.

If a graph appeared to be three-dimensional when laid out by the two–dimensional version of our algorithm, it laid out well in three dimensions, but in general, if it did not look three–dimensional originally, it did not benefit from being laid out in

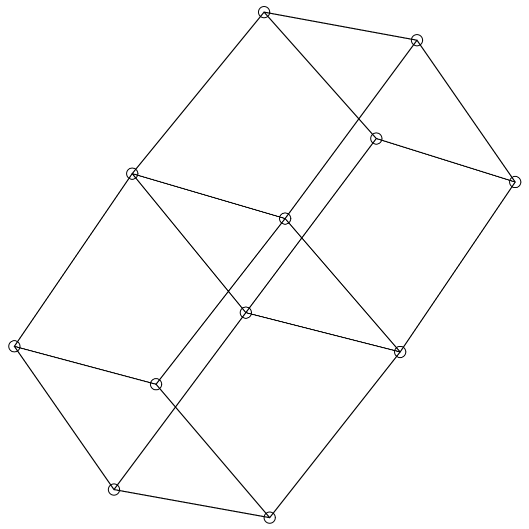

Figure 50. Twin Cubes(Graph in Figure 11 from Davidson and Harel[9])

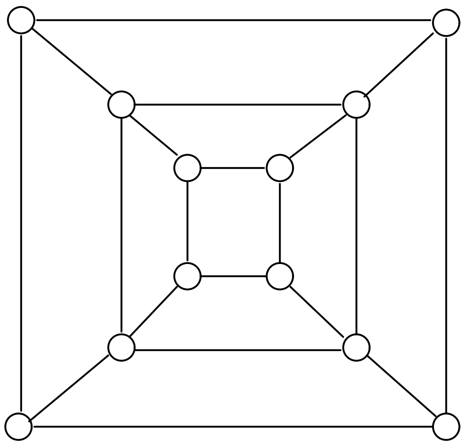

Figure 51. Figure 11(a) from Davidson and Harel[9] as proposed by Davidson and Harel

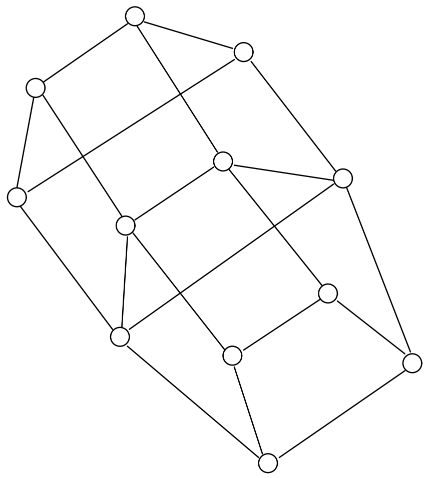

Figure 52. Figure 11(b) from Davidson and Harel[9] as drawn by Davidson and Harel

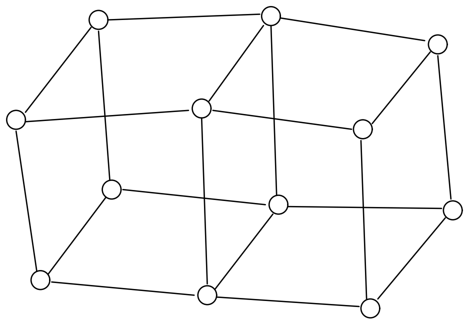

Figure 53. Figure 11(c) from Davidson and Harel[9] as drawn by Davidson and Harel

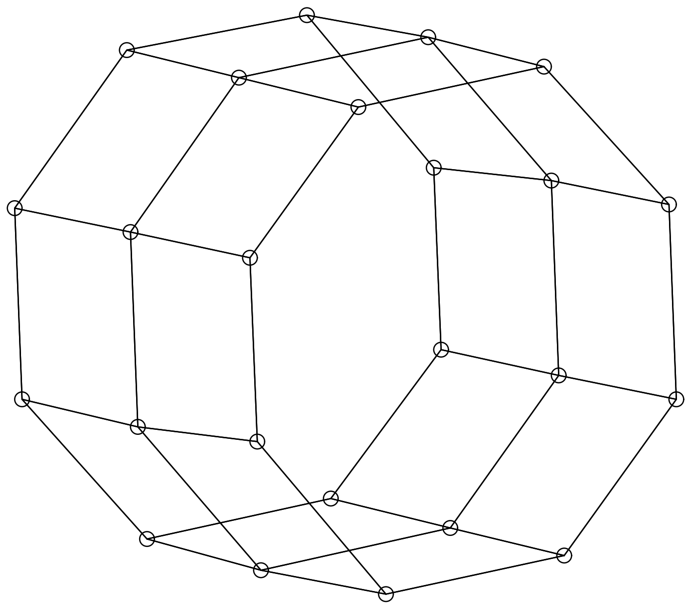

Figure 54. Mesh (Graph in Figure 2 from Davidson and Harel[9])

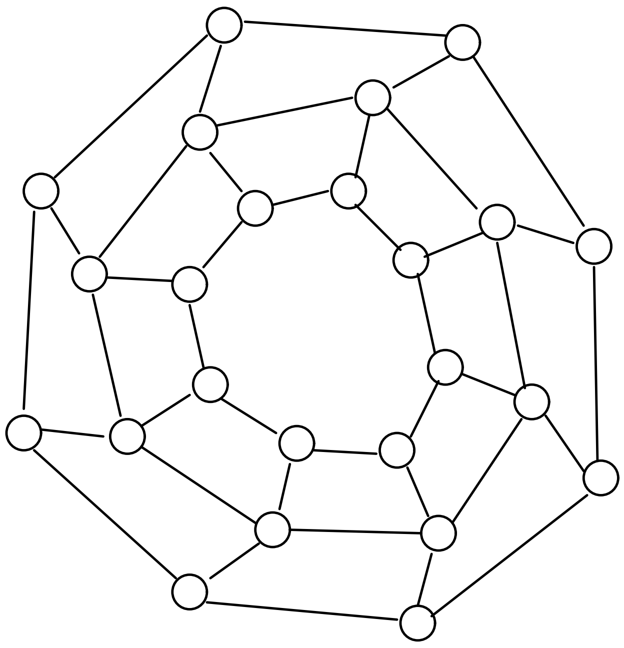

Figure 55. Figure 2 from Davidson and Harel[9] as drawn by Davidson and Harel

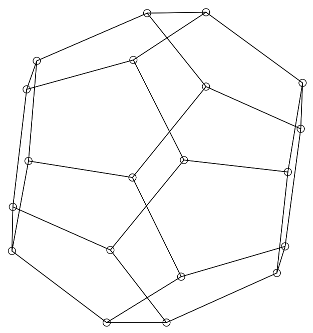

Figure 56. Poor Dodecahedron

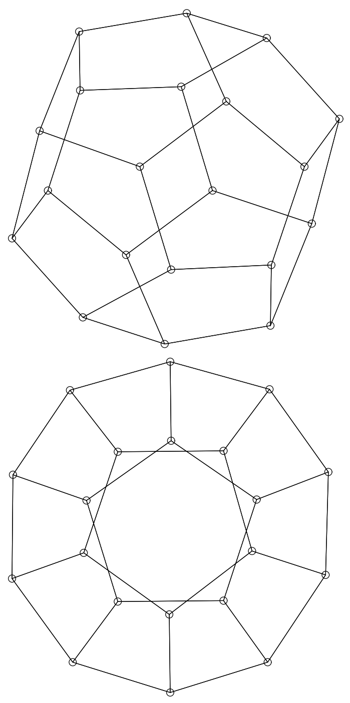

Figure 57. Good Dodecahedra

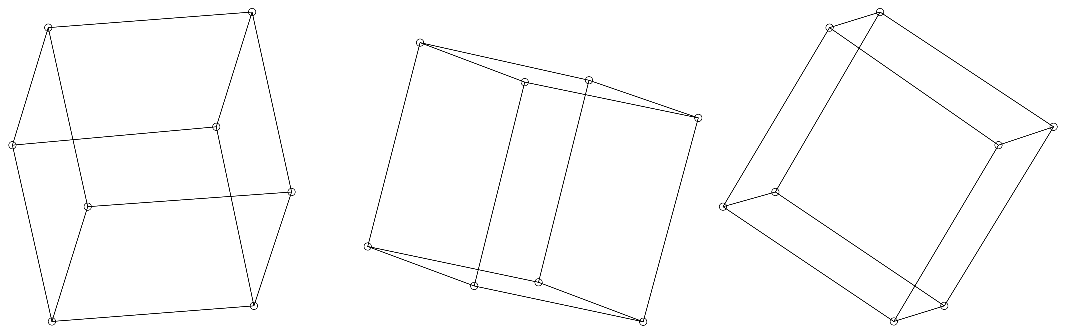

Figure 58. 3–D cube

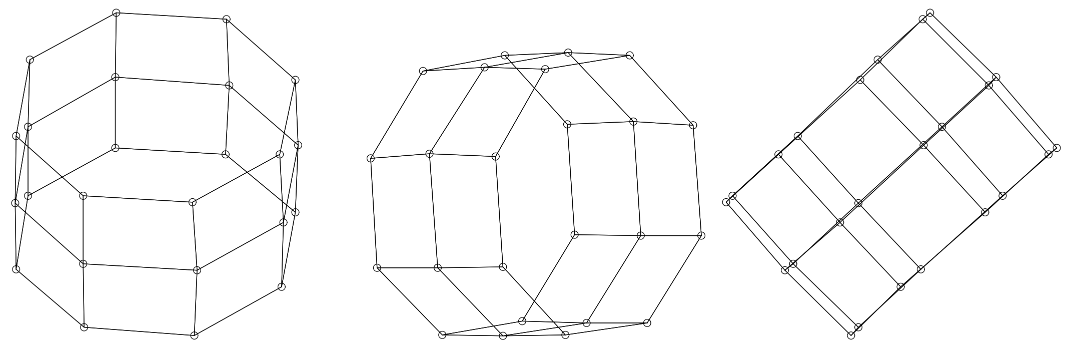

Figure 59. 3–D Mesh

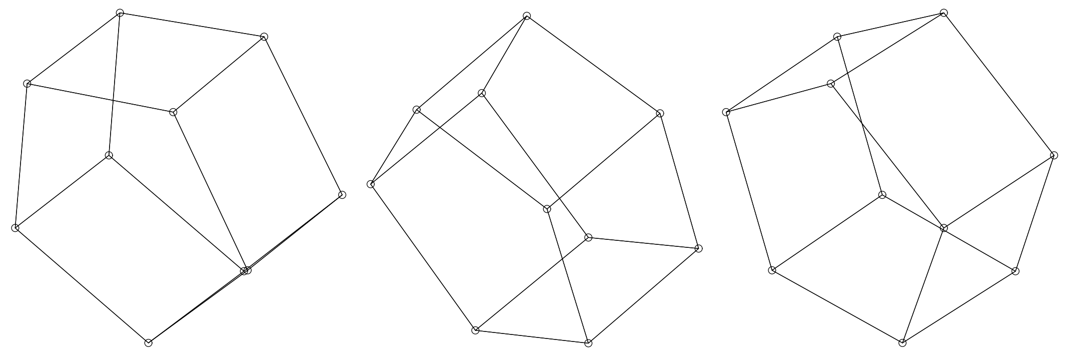

Figure 60. 3–D Pentagonal Prism

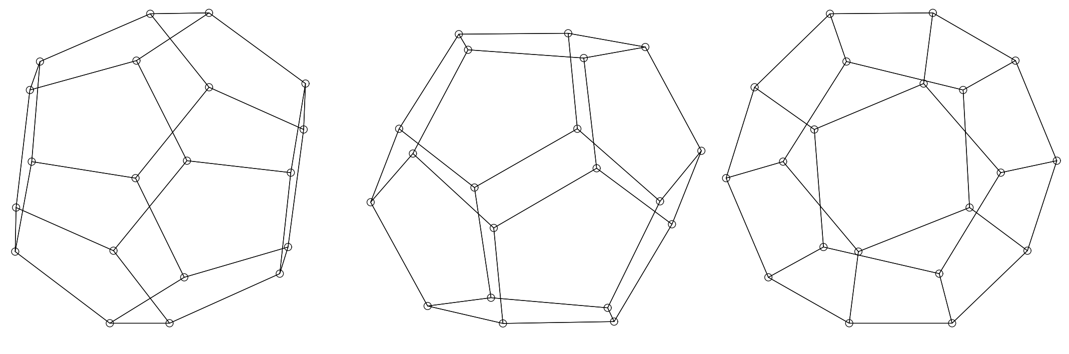

Figure 61. 3–D Dodecahedron

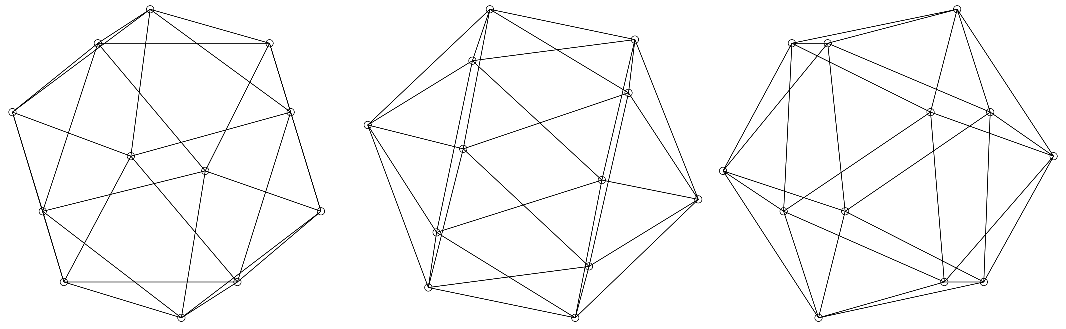
Figure 62. 3–D Icosahedron

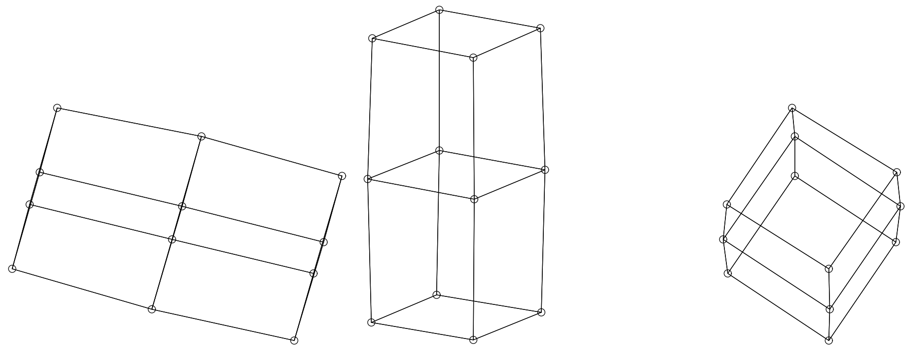
Figure 63. 3–D Twin Cubes

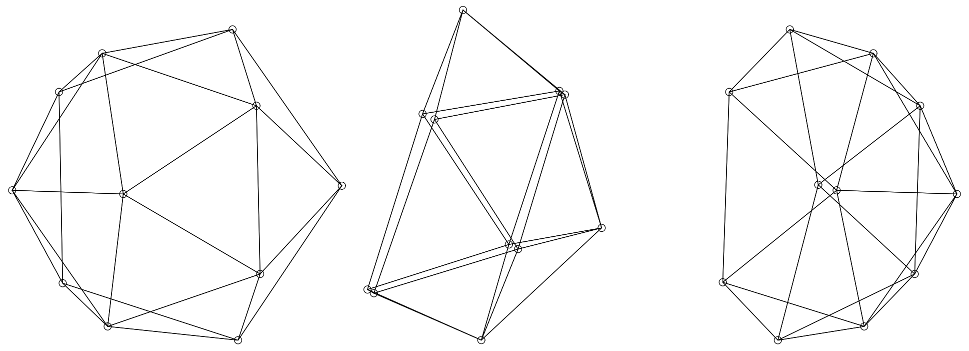
Figure 64. 3–D Icosahedron Variant

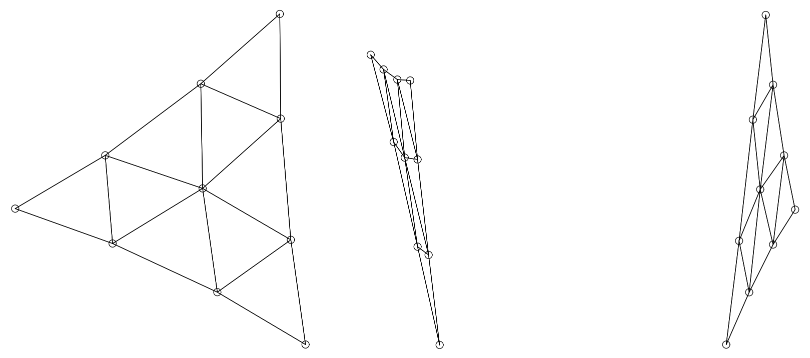

Figure 65. 3–D Triangulated Triangle

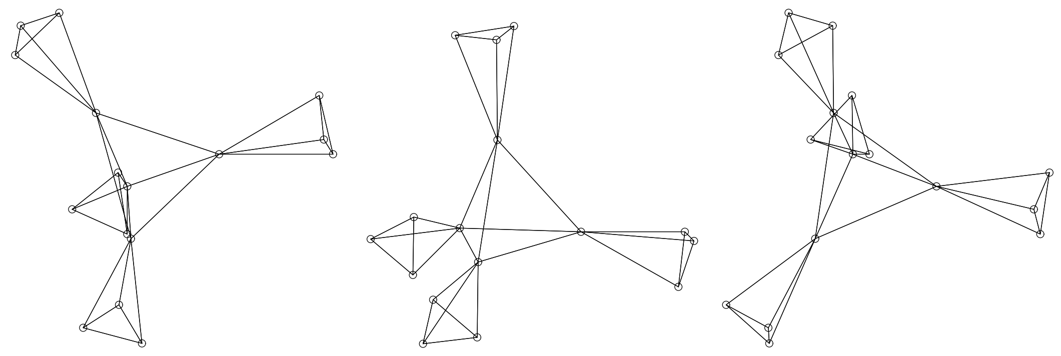

Figure 66. Monstrosity

three dimensions, and in fact, the z-orientation of vertices often appeared random and unpleasant. Some interesting exceptions are discussed in this section: first, the triangulated triangle of Figure 65 which appears to be nearly flat. The graph in Figure 66, which we saw earlier in Figure 18, might be described as K_4 with little K_4's attached to each vertex, looks here like a monstrosity. K_6 (Figure 67) is rendered as an octahedron with internal edges; it compares unfavorably with the icosahedron and the other graphs that had no lines "under" the surface. The third projection, along the z–axis, does not appear three-dimensional. K_8 (Figure 68) is a complete mess.

We have already seen the mesh and the twin cubes rendered in three dimensions. Davidson and Harel attempted to make the twin cubes planar by increasing the cost of edge–crossings in their energy function. They failed but they did produce a drawing that resembled a perspective projection. Given the three–dimensional layout of both the mesh and the twin boxes, we can choose a view normal vector that gives us a planar rendering of both. These are shown in Figure 69 and Figure 70. It required human intervention to choose the view normal vector.

In summary, three–dimensional lay out stands or falls according to the preference and expectations of the user. A graph that is 'really' three-dimensional ought to be laid out in three dimensions, and the user ought to be able manipulate this

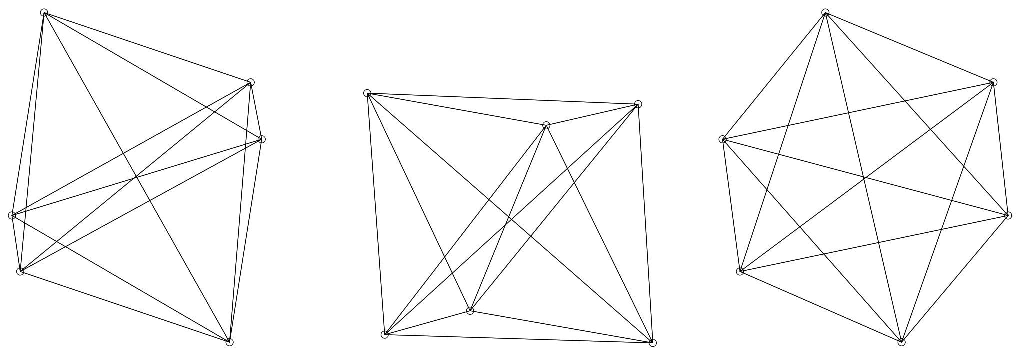

Figure 67. 3–D K_6

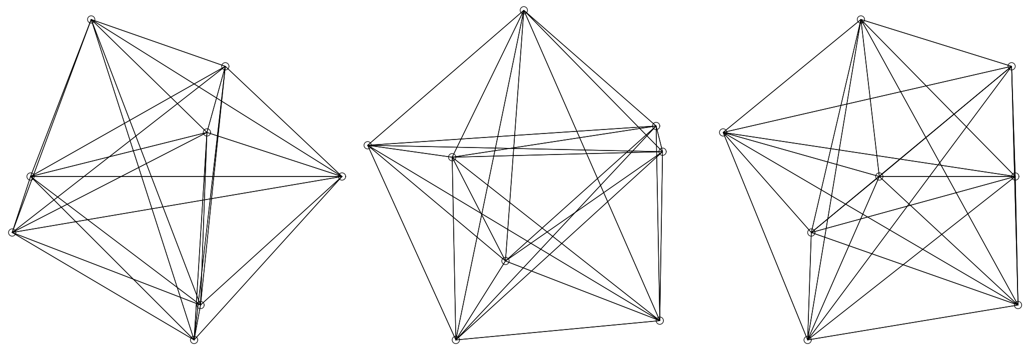

Figure 68. 3–D K_8

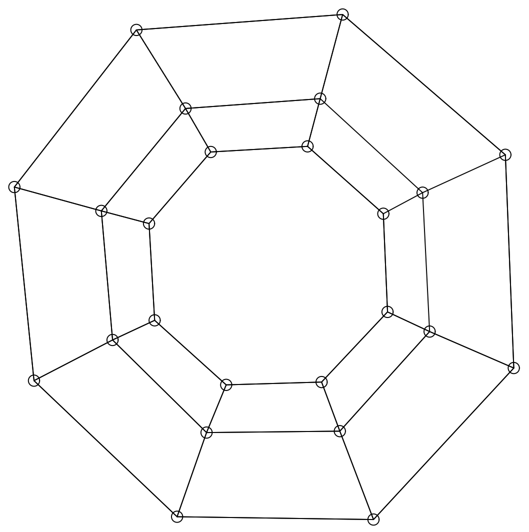

Figure 69. Planar Appearing Perspective Projection of Mesh

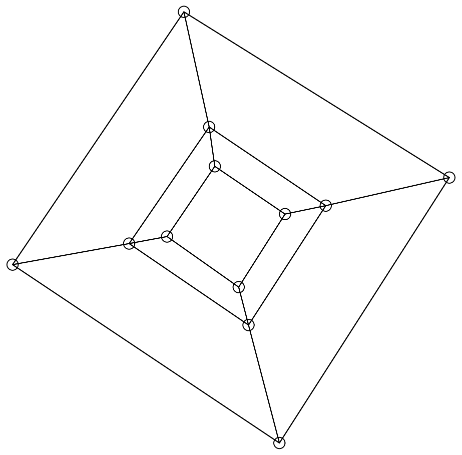

Figure 70. Planar Appearing Perspective Projection of Twin Cubes

three–dimensional layout with natural operations, such as choosing the viewpoint and direction of the view, and parallel or perspective projection.

GRID VARIANT

We must deal with a problem faced by Kamada and Kawai (and presumably Eades): unconnected graphs have nothing to hold them together; the connected components fly apart and flatten themselves against the walls. An example is Figure 72. The obvious solution is mentioned in Kamada and Kawai:[7] partition the graph into its connected components (this can be done in linear time) and give each component a 'tile' in the overall graph of area proportionate to its size, with each components layout calculated independently; Kamada and Kawai did not implement this. Without explicitly testing for connectedness or finding the connected components, we achieve this tiling effect as an added benefit of using the grid–square variant of our algorithm. We will also benefit when drawing 'nearly' unconnected graphs such as the twin K_5's connected by a single strand of Figure (71). Each clique acts as repulsive center that pushes all the vertices in the other clique far away and stretches the solitary connection. This rendering is acceptable, but the separation is perhaps a bit exaggerated.

The advantage of the $\Theta(|V| + |E|)$ (grid) variant of our algorithm is that distant vertices do not repel. We get almost the same effect as we would by separating the components. See Figures 73 and 74. At first, the connected components drift apart from one another, but eventually, they get out of each other's range and settle down. The connected graph improves in similar ways: the two cliques are not as distant as before and each is less distorted from the influence of the other.

There is considerable overhead involved in placing each vertex in a grid square each iteration and so we expect that the algorithm is more useful for larger graphs.

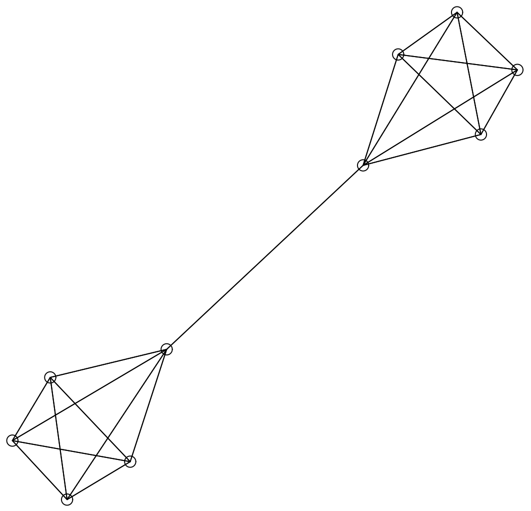

Figure 71. Twin K_5

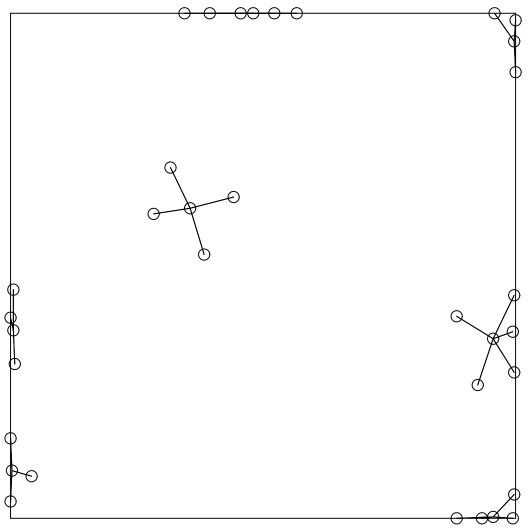

Figure 72. Unconnected Graph (Graph in Figure 3(b) from Davidson and Harel[9])

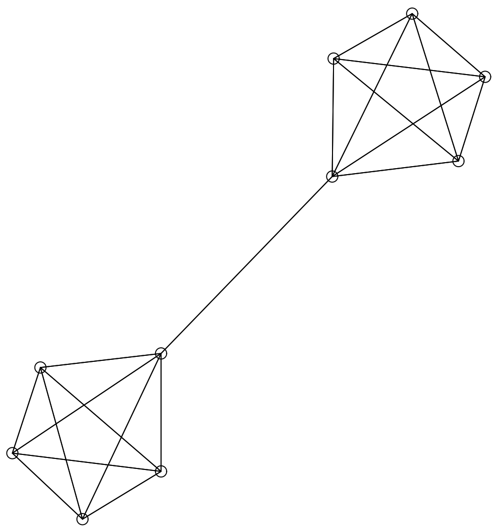

Figure 73. Variant Twin K_5

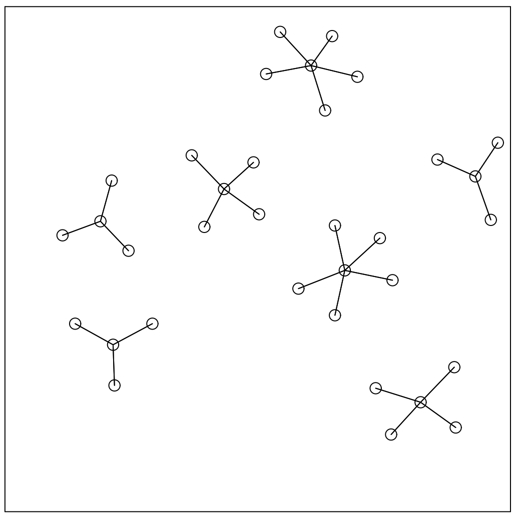

Figure 74. Variant Unconnected Graph (Graph in Figure 3(b) from Davidson and Harel[9])

In general we found to our annoyance that the variant was neither distinctly better nor distinctly worse than the basic algorithm, which is why we have presented both. On the large scale, it generated graphs that were smaller but on the smaller scale, the features of the graph appeared larger. Or, more precisely, it optimized the layout locally.

The grid variant Figure 76 appears 'fluffier' than either Figure 75, produced by the basic algorithm, or than the original from Eades, sketched in Figure 77, although it has unnecessary edge–crossings unlike the other two. A similar triplet is Figures 78, 79, and 80. We are not sure why Eades has a kink in his drawing; this could be depend on initial placement.

The remaining figures in this section show the layout produced by the basic algorithm on the left and that of the grid variant on the right. The symmetry of a large graph can be marred, as in Figure 81. The symmetry on the large scale requires that each vertex be placed with respect to every other vertex, or, every vertex must be calculated into the forces acting on every other. The tree in Figure 82 has more uniform edge lengths and better use of space when laid out by the grid algorithm. The grid algorithm is in general faster for the larger graphs, but requires more iterations. 50 iterations and 8 seconds produced the layout on the left in Figure 83, and 70 iterations and 6 seconds produced the one on the right.

CONCLUSIONS

The primary advantage of our algorithm is speed. Many of the graphs drawn in less than a second, and all in under 10 seconds on a SPARC station 1. We consider this to be 'interactive' speed.

Davidson and Harel's algorithm appears to be able to attain a slightly higher level of aesthetics and flexibility for the more complex graphs. Probably they could also lay–out graphs using a three–dimensional variant and use a grid, or multi-level approach, but then again, they face more complexities of implementation and pay a higher penalty in running time.

A major goal was that the algorithm should work reasonably well almost always, without the user having to fiddle with options, change the force laws, increase the number of iterations, or switch to either the $\Theta(|V|)$ or three–dimensional variants. Because we had no explicit concept of energy, and couldn't detect a stopping condition, we simply used 50 iterations every time; this was excessive on the simpler graphs, but that did not matter since the smaller graphs take less time per iteration in any case and the algorithm is so fast.

REFERENCES

1. P. Eades, "A heuristic for graph drawing," *Congressus Numerantium*, **42**, 149–160 (1984).
2. F. Harary, *Graph Theory* Addison–Wesley, Reading, MA, 1969.
3. P. Eades and R. Tamassia, "Algorithms for drawing graphs: an annotated bibliography," Technical Report CS-89-09, University of Queensland, October 1989.
4. N. Quinn and M. Breur. "A force directed component placement procedure for printed circuit boards," *IEEE Transactions on Circuits and Systems*, **CAS-26**, (6), 377–388 (1979).
5. P. A. Tipler, *Physics*, 2nd Edition, Worth Publishers, New York, NY, 1982.
6. J. Bentley, *Programming Pearls*, Addison–Wesley, Reading, MA, 1986.

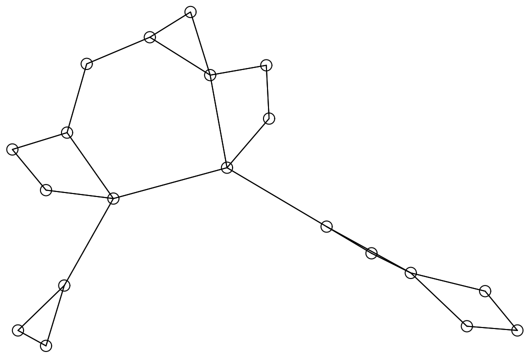
Figure 75. Graph in Figure 6(a) from Eades[1]

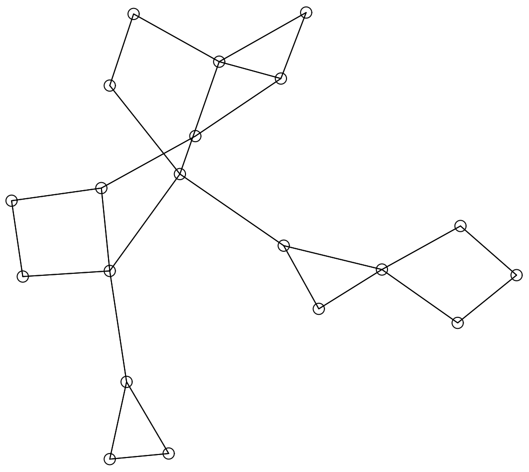
Figure 76. Variant of Graph in Figure 6(a) from Eades[1]

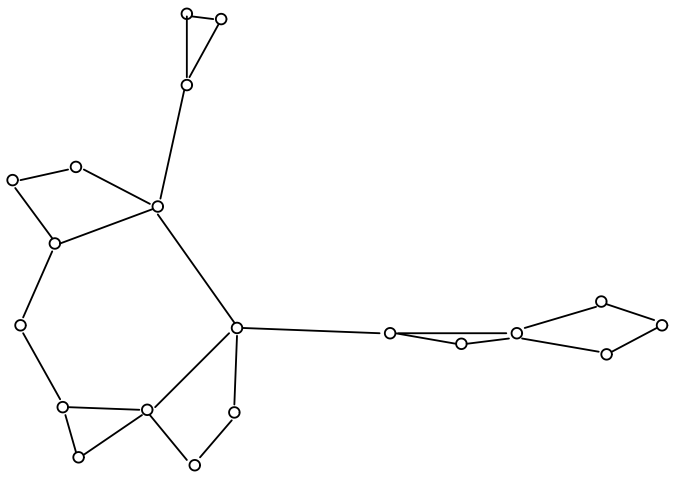
Figure 77. Figure 6(a) from Eades[1] *as drawn by Eades*

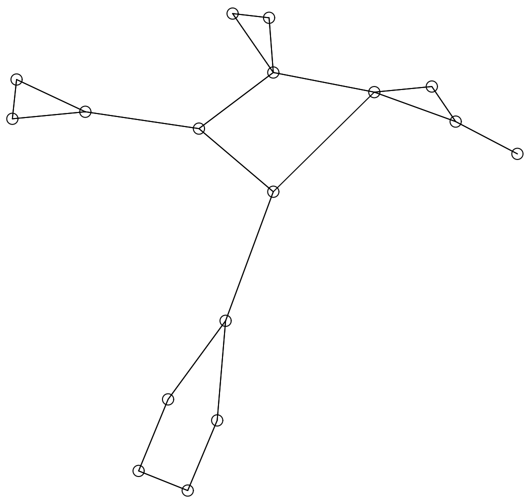
Figure 78. Graph in Figure 6(b) from Eades[1]

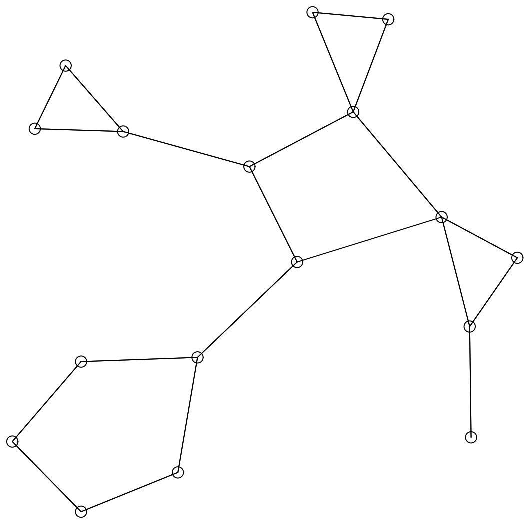
Figure 79. Variant of Graph in Figure 6(b) from Eades[1]

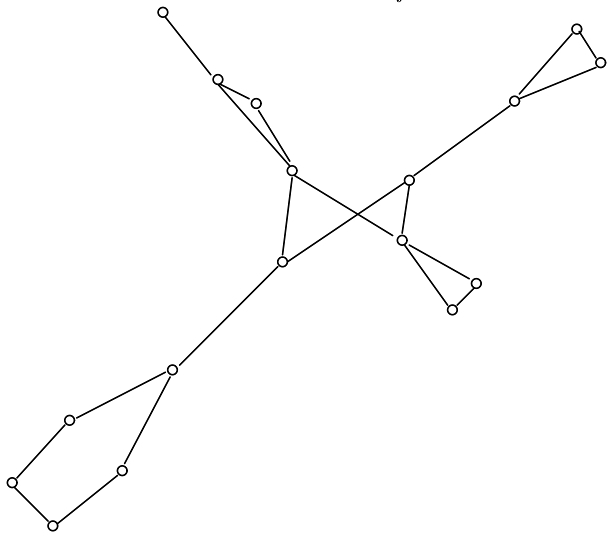
Figure 80. Figure 6(b) from Eades[1] *as drawn by Eades*

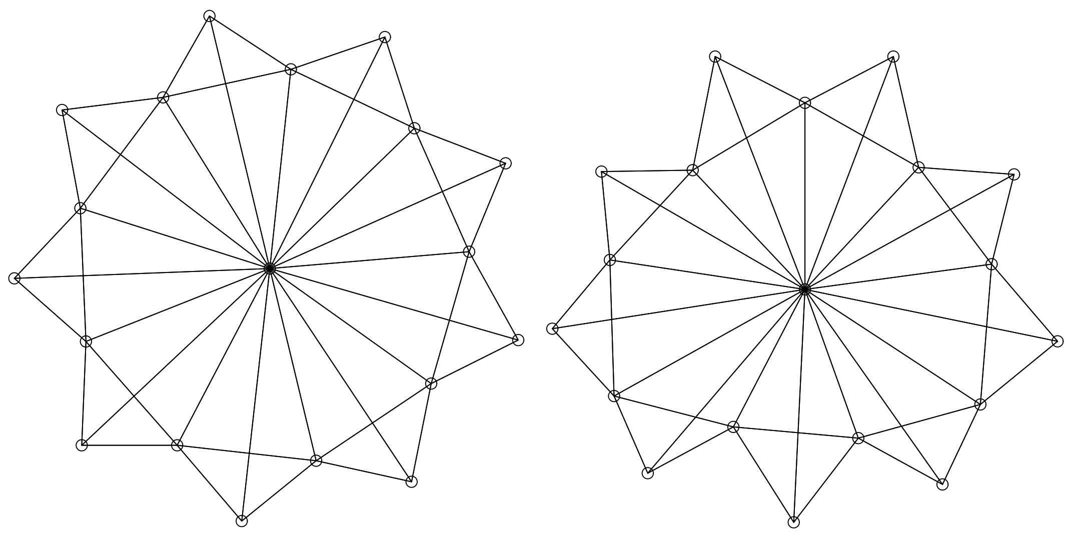

Figure 81. Graph in Figure 1 from Davidson and Harel[9]

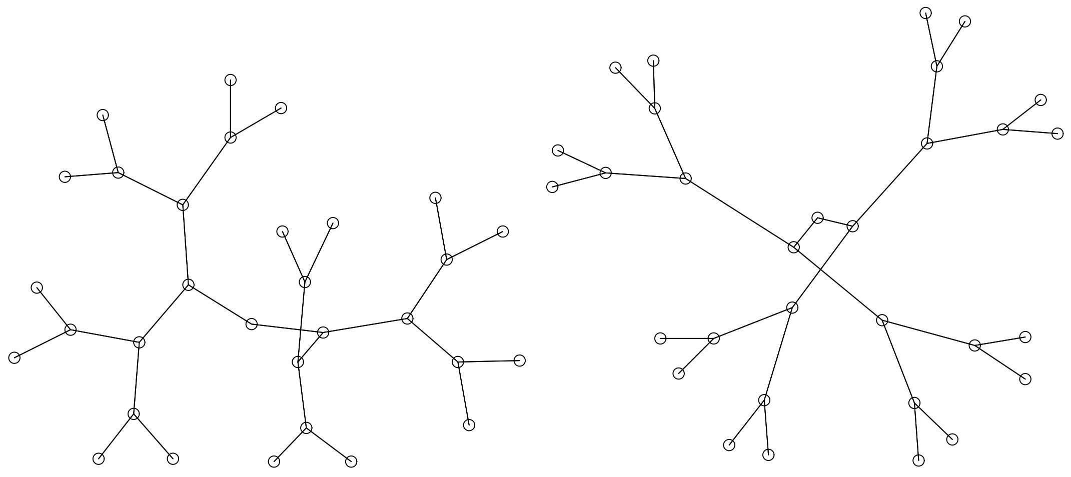

Figure 82. Graph in Figure 14(a) from Davidson and Harel[9]

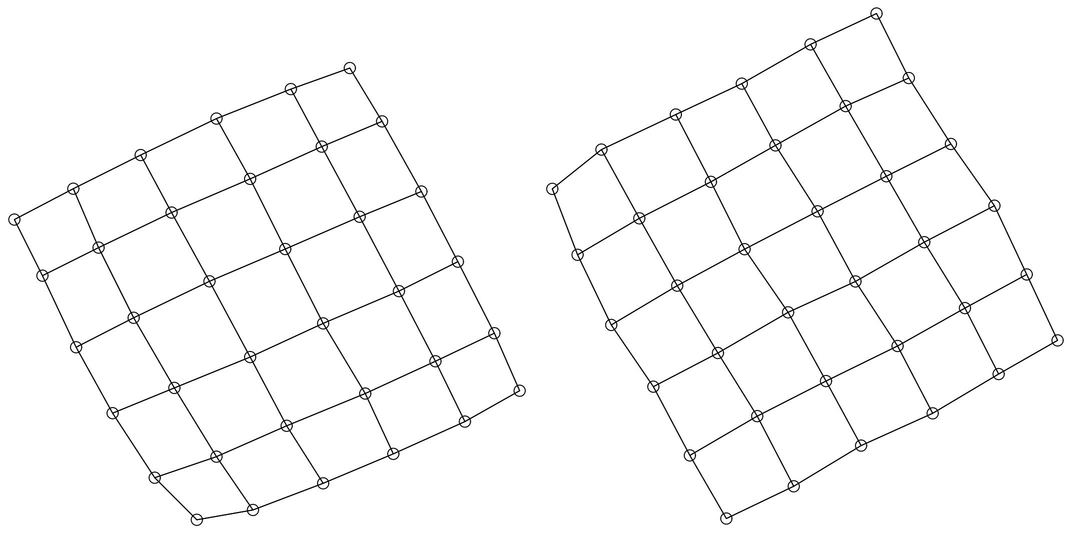

Figure 83. 25–Square Grid (Graph in Figure 13 from Davidson and Harel[9]*)*

7. T. Kamada and S. Kawai, "Automatic display of network structures for human understanding," Technical Report 88-007, Department of Information Science, Tokyo University, February 1988.
8. T. Kamada and S. Kawai, "An algorithm for drawing general undirected graphs," *Information Processing Letters*, **31**, (1), 7–15 (1989).
9. R. Davidson and D. Harel, "Drawing graphs nicely using simulated annealing," submitted for publication, July 1989.
10. S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, **220**, (4598), 671–680 (1983).
11. R. Otten and L. van Ginneken, *The Annealing Algorithm*, Kluwer Academic Publishers, Boston, MA, 1989.
12. D. Hearn and M. P. Baker, *Computer Graphics*, Prentice–Hall, Englewood Cliffs, NJ, 1986.
13. A. Appel, "An efficient algorithm for many–body simulations," *SIAM Journal on Scientific and Statistical Computing*, **6**, (1), 85–103 (1985).
14. L. K. Grover, "Standard cell placement using simulated sintering," *Proceedings 24th Design Automation Conference*, 56–59 (1987).
15. M. A. Breuer, "Min-cut placement," *Journal of Design Automation and Fault Tolerant Computing*, **1**, (4). 343–382 (1977).
16. B. W. Kernighan, A. V. Aho, and P. J. Weinberger, *The AWK Programming Language*, Addison–Wesley, Reading, MA, 1989.
17. B. Stoustrup, *The C++ Programming Language*, Addison–Wesley, Reading, MA, 1986.
18. S. C. Dewhurst and K. T. Stark, *Programming in C++*, Prentice–Hall, Englewood Cliffs, NJ, 1989.
19. T. Kamada and S. Kawai, "A Simple Method for Computing General Position in Displaying Three–Dimensional Objects," *Computer Vision, Graphics, and Image Processing*, **41**, 43–56 (1988).