Drawing Graphs Nicely Using Simulated Annealing

RON DAVIDSON and DAVID HAREL
The Weizmann Institute of Science

The paradigm of simulated annealing is applied to the problem of drawing graphs "nicely." Our algorithm deals with general undirected graphs with straight-line edges, and employs several simple criteria for the aesthetic quality of the result. The algorithm is flexible, in that the relative weights of the criteria can be changed. For graphs of modest size it produces good results, competitive with those produced by other methods, notably, the "spring method" and its variants.

Categories and Subject Descriptors: D.m [**Software**]: Miscellaneous; E.m [**Data**]: Miscellaneous; F.m [**Theory of Computation**]: Miscellaneous

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Aesthetics, graph drawing, simulated annealing

1. INTRODUCTION

Graphs are central to almost every topic in computer science, to the point that one cannot imagine subjects such as databases, data structures, automata theory, or software engineering without them. Traditionally, graphs are drawn manually, trying to keep the picture as simple and as meaningful as possible. This is especially important when the drawing is intended to emphasize some internal structure or symmetry.

Drawing large graphs nicely, that is, placing the nodes and edges so as to form a clear and pleasing picture, is not an easy task. In fact, in most cases, attaining an optimal solution seems to be computationally infeasible (see e.g., [Garey and Johnson 1983]). The problem calls for automatic support, which, in addition to expediting and simplifying the drawing process itself, will also attempt to produce better results using heuristic and/or probabilistic means. Over the last few years the problem has often been addressed.

D. Harel's research was partially supported by grants from the Israeli Ministry of Science and Technology, The Gutwirth Foundation, and the Basic Research Fund of Yeda, Ltd.
Authors' address: Dept. of Applied Mathematics and Computer Science, The Weizmann Institute of Science, Rehovot, Israel.

The large variety of graphs and the difficulty of the general problem has caused researchers to limit themselves to various special cases, e.g., trees, DAGs or planar graphs, or to concentrate on special appearances of the layout, e.g., hierarchical lattice-like structures or rectilinear grid drawings. Many interesting results have indeed been obtained for these cases. However, there have been only few attempts to solve the problem for general graphs, and almost none for more complicated objects such as hypergraphs or higraphs. An excellent survey of the literature in this area is DeBattista et al. [1993].

In this article we address the general problem of drawing nice-looking undirected straight-line graphs. Any proposed solution to this problem requires setting general criteria for the "quality" of the picture. Defining such criteria so that they apply to different types of graphs, but at the same time are combined into a meaningful cost function that can then be subjected to general optimization methods, was one of the main objectives of our work. Another was to introduce flexibility, so that the user may change the relative weights of the criteria to obtain varying solutions that reflect his or her preferences.

Our approach incorporates several simple criteria, some of which have been used in other algorithms: (1) distributing nodes evenly, (2) making edge-lengths uniform, (3) minimizing edge-crossings, and (4) keeping nodes from coming too close to edges. As mentioned, one of the main issues we have had to deal with is the delicacy of combining these into a cost function that can then be subjected to a general optimization method. We have chosen to base the algorithm itself on *simulated annealing* (SA)—an optimization method originating in statistical mechanics [Kirkpatrick et al. 1983; Metropolis et al. 1953]. The SA approach, reviewed in Section 2, has been adopted in recent years to help solve many kinds of combinatorial optimization problems, including VLSI layout.

Our algorithm is described in Section 3, and Section 4 contains examples of graphs drawn by implementation thereof. The program can show the graph being "beautified," by displaying the current version at certain intermediate stages. This has the pleasant consequence of providing an animated illustration of the simulated annealing scheme in action. The "feel" this gives for the way SA operates is difficult to get from other kinds of applications that offer less visually appealing feedback.

One of the more appealing ways of demonstrating the power of the algorithm is to test its ability to match a human's idea of "nice." This is done by asking someone to draw an arbitrary graph that he or she thinks is "nice" (without discussing the criteria for niceness), and running the algorithm on a randomly layed out version of it. We have not gathered statistics, but have been repeatedly pleased with the results. In the vast majority of our experiments, the program, run with default settings for the weights of the various criteria, drew a version that was very close to that drawn initially by the person. This, however, is true only for graphs of quite modest size, up to around 30 nodes and 50 edges. On larger graphs, the quality of the output deteriorates rapidly. Also, simulated annealing, as is well-known, is a time-consuming procedure, and this carries over to our use of it. The time performance of our implementation is quite poor, a fact we shall have more to say about in Section 3.

In Section 5 we review related work that addresses the problem in its generality, notably the "spring-method" Eades [1984], Fruchterman and Reingold [1991], Kamada [1989] and Kamada and Kawai [1989], and point out similarities with our approach.

Section 6 discusses future research, and includes a brief description of our subsequent work, reported upon in Harel and Sardas [1995, 1996]. This is important, since the present paper was originally written in 1989. In our later work, simulated annealing is used only to fine-tune a rough solution found using other, more elaborate, techniques which are designed to try to lay the graph out in a manner that is as close as possible to being planar.

2. A QUICK LOOK AT SIMULATED ANNEALING

Simulated annealing (SA) is a flexible optimization method, suited for large-scale combinatorial optimization problems. See Hajek [1985], Johnson et al. [1989, 1991] and van Laarhoven and Aarts [1987] for surveys of the method and its use. Simulated annealing originated in statistical mechanics [Metropolis et al. 1953], and was formulated in general terms in Kirkpatrick et al. [1983]. It has been applied successfully to classical combinatorial optimization problems, such as the traveling salesman problem, and problems concerning the design and layout of VLSI. SA differs from standard iterative improvement methods by allowing "uphill" moves—moves that spoil, rather than improve, the temporary solution.

The problems for which SA is useful are characterized by a very large discrete configuration space, too large for an exhaustive search, over which an objective cost function is to be minimized (or maximized). After picking some initial configuration, most iterative methods continue by choosing a new configuration at each step, evaluating it and possibly replacing the previous one with it. This action is repeated until some termination condition is satisfied (e.g., no move reduces the objective function). The procedure ends in a minimum configuration, but generally it is a local minimum, rather than the desired global minimum. The SA method tries to escape from these local minima by using rules that are derived from an analogy to the process in which liquids are cooled to a crystalline form, a process called *annealing*.

It is well known that when a liquid is cooled slowly, it reaches a totally ordered form, called *crystal*, which represents the minimum energy state of the system. In contrast, rapid cooling results in amorphous structures, that have higher energy, representing local minima. The difference lies in the fact that when a liquid is cooled slowly, the atoms have time to reach a thermal equilibrium in each and every temperature. In this state, the system obeys the Boltzmann distribution:

$$p(E) \simeq e^{-E/kT}.$$

Here, $p(E)$ specifies the probability distribution of the energy values of the states E, as a function of the temperature T and the Boltzmann constant k. On the one hand, for every temperature, each energy E has nonzero probability, and thus the system can change its state to one with higher energy. On the other hand, at low temperatures, the system tends to be in states with very low energy, with the global minimum achieved at temperature zero. (In the sequel, we shall assume that $k = 1$, since our temperatures are artificial anyway.)

Metropolis et al. [1953] devised an algorithm for simulating this annealing procedure by a series of sequential moves. The basic rule is that the probability with which the system changes its state from one with energy $E1$ to one with energy $E2$ is:

$$e^{-(E2-E1)/kT}.$$

This rule implies that whenever the energy $E2$ of the new candidate state is smaller than the current energy $E1$ the system will take the move, and if it is larger the state change is probabilistic. Kirkpatrick et al. [1983] were apparently the first to realize that the above procedure could be used for general optimization problems.

Several entities must be determined whenever the SA method is applied. These include the following:

—The set of configurations, or states, of the system, including an initial configuration (which is often chosen at random).
—A generation rule for new configurations, which is usually obtained by defining the neighborhood of each configuration and choosing the next configuration randomly from the neighborhood of the current one.
—The target, or cost, function, to be minimized over the configuration space. (This is the analogue of the energy.)
—The cooling schedule of the control parameter, including initial values and rules for when and how to change it. (This is the analogue of the temperature and its decrease.)
—The termination condition, which is usually based on the time and the values of the cost function and/or the control parameter.

Having defined all of these, the schematic form of the SA method is as follows:

```
(1) choose an initial configuration $\sigma$ and an initial temperature $T$;
(2) repeat the following (usually some fixed number of times):
    (a) choose a new configuration $\sigma'$ from the neighborhood of $\sigma$;
    (b) let $E$ and $E'$ be the values of the cost function at $\sigma$ and $\sigma'$
        respectively; if $E' < E$ or *random* $< e^{(E - E')/T}$ then set $\sigma \leftarrow \sigma'$;
(3) decrease the temperature $T$;
(4) if the termination rule is satisfied, stop; otherwise go back to step 2.
```

(In clause 2(b), *random* stands for a real number between 0 and 1, selected randomly.)

Many variations on this general scheme have appeared in the literature. Some are motivated by theoretical investigations into the analogy to the physical phenomenon, and others are derived from implementation experience. Many are described in van Laarhoven and Aarts [1987], which also contains an abundance of references.

SA is not always suitable for a given optimization problem. A basic requirement from the cost function is that it should not have overly steep descents, that is, the width of the valleys (around the minima points) should be roughly proportional to their depth. A very narrow valley that contains the desired minimum point has a low probability of being found. To be amenable to solution by SA, the problem must also admit near optimal solutions that are acceptable. SA is very successful in finding minima values that are close to the global minimum, but seldom does it detect the global minimum itself. Another important requirement is that the cost function should be easily calculated for a new configuration (possibly using the value for the old one). This is due to the fact that these calculations are carried out repeatedly and constitute the major computational task in implementing SA.

The SA method has been applied successfully to many problems. Its first application Kirkpatrick et al. [1983] was to the traveling salesman problem. It has been used for many problems in the design and testing of VLSI, including the placement problem, channel routing, and the folding of PLAs, as well as for problems in image processing, coding theory, floor planning, graph and number partitioning, and graph coloring.

3. THE GRAPH DRAWING ALGORITHM

An undirected graph is simply a pair (V, E), where V is the set of nodes and E, the set of edges, is a symmetric subset of $V \times V$. Transforming such a nongraphical description of a graph into a visual representation leaves many issues to be resolved, such as the shapes and sizes of the elements, and, most significantly, their locations. While it is quite easy to draw very small graphs manually in an acceptable way, the manual drawing of large graphs is a painful task, and the resulting picture is often unacceptably deformed. A similarly difficult task is to modify a given drawing of a graph by adding or deleting elements. Carrying out a manual modification can become as tedious as redrawing the whole graph, because it is often necessary to reposition many of the nodes and the edges. This task becomes far more difficult when considering more intricate types of graphical objects, such as hypergraphs [Berge 1973] or higraphs [Harel 1988], which have applications in system design, database theory, and knowledge representation.

The basic obstacle that has to be overcome in preparation for an algorithmic solution is the definition of criteria for a good drawing. Several such criteria come to mind (and have indeed been considered in several of the papers written on the subject; see Battista et al. [1993]), including evenly distributed nodes, edges of close-to-uniform lengths, a minimal number of crossings, symmetry whenever possible, and conformation to the shape of the frame (say, the computer screen). Clearly, these could be mutually conflicting when applied without care. In fact, in Section 4 we shall see conflicts between the desire to achieve symmetry and minimize crossings. Moreover, aesthetics is a subjective issue; it would be very helpful to be able to provide a sufficiently flexible set of rules for controlling the quality of a drawing, parameterized and tuned to meet personal taste.

The input to our algorithm is an adjacency list or matrix, defining the edges of a general undirected graph. The output is to be projected on a rectangular grid that represents the monitor or the plotter resolution. Restricting ourselves to the convention of representing edges by straight lines, the output of the algorithm reduces to a list of locations of the nodes on this grid. As discussed in the previous section, the use of the annealing scheme requires the definition of analogues to the components of the physical annealing process. We now discuss how our algorithm deals with each of these in turn.

Configurations. A configuration is simply a candidate for a drawing, that is, an assignment of a unique grid point to each of the nodes. If we have no *a priori* information about the graph's visual representation, the initial configuration is chosen at random, by selecting arbitrary grid locations for the nodes. Our implementation allows the user to draw an initial configuration, or to have the program perform a random scattering of a given configuration.

Neighborhoods. In the spirit of the physical annealing process, neighboring configurations must be similar, in the sense that they represent only a slight perturbation in the system's state. Consequently, we have defined the neighborhood of a configuration to contain all configurations that differ from it by the location of a single node. Thus, the generation rule for new configurations amounts to moving one node to a new location. Furthermore, we have bounded the distance between two consecutive locations of a node by limiting a move to be in a circle of decreasing radius around the node's original location. The radius is taken to be very large at the beginning of the process, not constraining the choice, but it gets smaller as the algorithm proceeds. This is *range limiting* of sorts, whereby the better the configuration the smaller the allowed perturbation. Here we differ from applications of SA that use a fixed and constant rule for creating new configurations. (See Siarry et al. [1987] for a similar approach.) This decision, we believe, accelerates the convergence of the algorithm, but in general it might affect the quality of the results. We actually take the idea further, by restricting the new location to lie on the *perimeter* of the circle. This eliminates small perturbations at the beginning of the process, when the radius is still large.

The Cost Function. The most important step in applying the simulated annealing scheme is the definition of the energy, or the cost function, which, in our case, captures what we mean by a nice-looking drawing. Thus, all desired features of the final graphs must be translated into a contribution to this function, in accordance with their relative importance. From a computational point of view, great care should be taken when defining the energy function, since its repeated calculation is the heaviest computational task of the algorithm.

Here are the criteria we have used and their contributions to the energy function. We also point out possible generalizations.

—*Node distribution*: One obvious criterion for drawing a "nice" graph is to spread the nodes out evenly on the drawing space. The distances between the nodes need not be perfectly uniform, but the graph should occupy a reasonable part of the drawing space, and, if possible, the nodes should not be overcrowded. These objectives are achieved by two components of our energy function. The first prevents the nodes from getting too close to each other, while the second deals with the borderlines of the drawing space. The first component is the sum, over all pairs of nodes, of a function that is inverse-proportional to the distance between the nodes. We have tested several such functions, and have found that the best results stem from applying a function that is analogous to the electric potential energy: for each pair of nodes i, j, the term

$$a_{ij} = \frac{\lambda_1}{d_{ij}^2}$$

is added to the energy function, where d_{ij} is the Euclidean distance between i and j. Here, λ_1 is a normalizing factor that defines the relative importance of this criterion compared to the others. Increasing λ_1 relative to the other normalizing factors we describe in the sequel causes the algorithm to prefer pictures with smaller distances between nodes.

Notice that we did not use the analogue of the electric charge of the two nodes. We could have multiplied a_{ij} by $q_i q_j$, where q_i reflects the attraction of other nodes to node i. This way, the terms a_{ij} for nodes with large q's would be larger, and the algorithm would tend to shorten the corresponding distances. One could imagine applications in which particular nodes are to be spread out more spaciously than others, in which case the electric charge analogue may be useful. We have implemented this option, and an example of its effect appears in Section 4.

—*Borderlines*: As in physics, truly minimizing the potential energy might result in spreading out the elements indefinitely. To avoid this, and to reflect the physical limitations of the output device, a component is added to the energy function to deal with the borderlines of the drawing space. We have examined several models for this. One was to add dummy nodes, spread out evenly along the borderlines, and forbidden to move during the process (see, e.g., Tutte [1963]). We rejected this idea, since it turns out that the number of additional nodes required depends on the number of nodes in the input graph, and that adding less than the required number results in real nodes "escaping" between the borderline points and being placed beyond them.

We have decided to base the energy function on the distance between a node and the borderlines, taking into account the sum of distances to all four borderlines (rather than the distance to the closest one only). We have thus added the following term to the energy function:

$$m_i = \lambda_2\left(\frac{1}{r_i^2} + \frac{1}{l_i^2} + \frac{1}{t_i^2} + \frac{1}{b_i^2}\right),$$

where r_i, l_i, t_i and b_i stand for the distances between node i and the right, left, top and bottom sides, respectively. This way, the nodes are prevented from getting too close to the borderlines, but they do not stay in the center of the drawing space either because of the first term of the energy function. Clearly, increasing λ_2 relative to λ_1 pushes the nodes towards the center, while decreasing it results in using more of the drawing space near the borderlines.

As before, an appropriate charge q_i could have been associated with each node, and used as a multiplicative factor with the m_i's. This would differentiate between nodes with respect to the importance of placing them near the center of the picture. Nodes with a large charge would then be placed closer to the center. This feature has not been implemented.

—*Edge lengths*: So far, we have discussed the nodes. As far as the edges go, a naive criterion is to try to shorten them to the necessary minimum without causing the entire graph to become too tightly packed. Accordingly, the next component of our energy definition is intended to penalize long edges. We have experimented with a number of such functions. One is to use the potential function for the distance between nodes, multiplying the aforementioned a_{ij} by an additional coefficient e_{ij} that reflects the presence or absence of an edge between nodes i and j. A more direct way is to add to the cost the length of each edge, or its square. This last possibility seemed to be the most appropriate. Hence, for each edge k of length d_k, we add the term

$$c_k = \lambda_3 d_k^2$$

to the energy function, where, again, λ_3 is an appropriate normalizing factor. This is similar to the energy function used in Tutte [1963].

Here too, a special weight e_k could have been associated with edge k, to be added as a multiplicative factor. This would enable us to influence the algorithm into making the kth edge longer or shorter than an average edge, something that might be valuable in some applications.

We found that the simple criteria described so far produce fairly nice pictures for many different graphs. The short-edges criterion turned out to take care of positioning the nodes so that most unnecessary edge crossings were eliminated. However, some graphs require a more direct treatment of edge crossings.

—*Edge crossings*: In general, minimizing the number of crossings is an important goal, but is difficult to achieve. An obvious special case is to try to eliminate all crossings when the graph is planar. Algorithms for producing a crossing-free picture of a planar graph do exist, but it would be beneficial to try to minimize crossings while retaining the other niceties of the drawing. This way, we might not reach the minimal number of crossings (and some crossings might remain even in planar graphs), but we will generally be producing a more pleasing picture.

We have incorporated edge crossings into our algorithm simply by adding to the cost function a constant penalty λ_4 for every two edges that cross. The particular fixed value that we use as a default seems to perform well in practice. Of course, increasing λ_4 means attributing more importance to the elimination of edge crossings, and results in pictures with fewer crossings on average. However, this may be at the expense of other aesthetics. Clearly, one could implement a penalty that would vary, say, by multiplying λ_4 by $f_k f_l$, where f_k is some weight attributed to edge k, representing the importance we attach to edge k not intersecting other edges. Adjusting λ_4 against the other constants we have discussed leads to interesting variations on the algorithm's behavior, as the examples in Section 4 show.

—*Node-edge distances*: The previous item discussed penalizes edges that cross, but does not penalize ones that are close and do not quite cross each other. In some cases, this situation is taken care of implicitly by the node separation component of the energy function, but we would like to cover more cases. We have chosen to generalize the edge crossing notion, considering crossing edges to be the extreme case of close edges with distance zero. Extending the edge crossing notion in this way has another advantage. It is best that the energy function in applications of SA be continuous, changing smoothly with the change of configurations (see, e.g., van Laarhoven and Aarts [1987]). The edge crossing component of our energy function does not enjoy this property, since a slight change in a node's location might change the energy by λ_4. The generalization suggested eliminates this problem. We generalize further by measuring the distance between every node and every edge (and not just between pairs of edges). This way, the calculations are simpler and we cover more cases (such as V-like pairs of edges).

Accordingly, we define the distance between a node and an edge as the minimal distance between the node and any point on the edge. The contribution of this distance to the cost function is inverse proportional to its square; that is, for node k and edge l with distance g_{kl}, the value of

$$h_{kl} = \frac{\lambda_5}{g_{kl}^2}$$

is added to the cost function. A minimum distance $g_{\min}$ is defined, and distances less than $g_{\min}$ contribute

$$\lambda_4 = \frac{\lambda_5}{g_{\min}^2}$$

to the energy function, as if edge-crossing occurs. As with edge crossings, we could have generalized this component by introducing node-specific and edge-specific weights q_k and f_l as multiplicative factors.

As explained later, this criterion is not used in the default settings of the main algorithm, but is used in the fine-tuning parts.

The Cooling Schedule. The cooling schedule is one of the most delicate parts of the annealing algorithm. Most applications follow the basic geometric schedule described in van Laarhoven and Aarts [1987], and we have done so too. We now describe the details of our schedule.

—*Initial temperature*: Since the initial configuration of the system is chosen at random, we set the initial temperature to be high enough to accept almost any move at the beginning. The goal is to "shake" the graph well, virtually letting any configuration be a possible starting point of the algorithm. The exact value of the control parameter representing the temperature was determined empirically, so as to conform with the above rule. In cases where the initial configuration is known to have some resemblance to the desired final result (i.e., it is in the neighborhood of the minimal point), a lower temperature can be chosen at the beginning, so as to retain the partial result already given.

—*Temperature reduction*: As the general SA scheme dictates, a series of moves is attempted at each temperature, with some of the trials being successful and the configuration possibly changing. We have to decide when to change the temperature, and how to change it. Typical applications set the number of trials for each temperature to be polynomial in the length of the input. We have used a linear number—actually 30 times the number of nodes—and this seems to be sufficient. For particularly difficult examples, we found that running the algorithm with a larger number of trials per temperature yielded only marginal improvement. A correct simulation of the physical process would prescribe allowing a potentially unlimited number of moves at each temperature, and this is one of the assumptions in the mathematical analysis of SA. However, our linear approximation to such an infinite process yields acceptable results.

Nevertheless, studying the behavior of our algorithm, it seems that most of the changes at a given temperature take place during the first moves, and as equilibrium for that temperature is approached changes become rare. This suggests a method for accelerating the execution of the algorithm, which we have not implemented: rather than adhering to a number of moves that is fixed in advance (based on the size of the graph), keep a record of the success rate in the attempts to change configurations, and when this rate becomes low enough, decrease the temperature. Some upper limit on the number of attempts at each temperature should also be kept.

Our cooling rule is geometric. We follow most researchers in that we apply the following rule. If T_p is the temperature at the pth stage, then the temperature at the next stage is given by

$$T_{p+1} = \gamma T_p,$$

with γ between 0.6 and 0.95. We used $\gamma = 0.75$ in most of the examples to achieve a relatively rapid cooling. Cooling too rapidly results in the inability to escape from local minima, and the resulting drawings turn out to be far from optimal. Slow cooling can improve the results at the expense of increasing the running time.

The Termination Condition. We have set the number of stages (with different temperature values) to be a constant, fixed in advance, independent of the size of the input graph. Running the algorithm for ten stages, we found, yields better than acceptable results. Highly complex examples, however, can benefit from a few more stages, increasing the chances of escaping local minima. It is generally the case that if no change has occurred during two or three stages then none is likely to occur until the end of the run. We could have instructed our implementation to stop when such a situation arises, but have not done so.

Fine Tuning. The algorithm as described so far finds its way into a valley that contains one of the globally lowest points in terms of energy. The result usually reflects some structural aspects of the graph reasonably well. For example, nodes of connected components quite rapidly gather in the same area. However, it is unlikely that we will reach a truly minimal point in the valley in this fashion, since our node-moving rule is too coarse. Of course, we could solve the problem by running the algorithm until the radius of the circle that determines a node's move becomes a single pixel. This, however, is terribly time-consuming, and unnecessary.

What we found is needed, in addition to the annealing scheme described so far, is a *fine tuning* procedure, whereby only very small changes (just a few pixels) are allowed at a time, and uphill moves are completely forbidden. We have implemented such a procedure as a three-stage repetition, which is carried out after the ten steps of the main part of the algorithm.[1]

[1]Both of these numbers can be changed by the user.

During the fine-tuning iterations, the energy function is enriched by adding the component that deals with distances between nodes and edges. To the human eye, the improvement obtained by the fine tuning is quite striking, as the nodes seem to suddenly "jump" to their very best positions, given the result of the first ten stages.

Complexity and Performance. Using V and E to stand for $|V|$ and $|E|$, respectively, we find that the algorithm runs in time at most $O(V^2 E)$. Here is why:

The number of iterations of the external loop is bounded by a constant, with each of these performing $30V$ attempts to change the configuration. The components of the energy function are updated for each such attempt, rather than being recalculated from scratch, and we now show that they take a total of $O(VE)$ time. One should keep in mind that new configurations are generated by moving a single node only.

Updating the component concerning distances between nodes is linear in V, since only distances to the moved node require recalculation. The same goes for updating the edge lengths, since there at most V nodes adjacent to the one moved. Updating the distance of the moved node from the border-lines takes constant time. The heaviest calculations are incurred when updating the components for edge crossings and distances between nodes and edges. Both of these take $O(VE)$ in the worst case, since the number of pairs of edges, one of which is adjacent to the node and the other of which is not, is at most VE.

The algorithm also uses an $O(\max(V^2, E^2))$ amount of memory.

The algorithm was implemented in C, and runs on a SUN SPARCstation 1 or a Convex C2 machine.[2] The program is about 1500 lines long, more than half of which takes care of the user interface and the graphics. Using the Convex, most of our examples (all those with up to 25 nodes) complete the annealing iterations in less than 16 seconds. Small graphs (with around 10–12 nodes) terminate in 3–5 seconds, while our largest example (Fig. 13, with 37 nodes and 68 edges) ran for about two minutes. These times apply to a run with default settings: 10 annealing iterations and $30V$ trials per iteration. For many input graphs these parameters could have been set lower, and the running time would have been reduced accordingly.

The running time of a fine-tuning iteration is 30% to 50% higher than an annealing iteration, because of the decision to consider the distances between nodes and edges in the fine-tuning phase even when they were not considered in the annealing phases (as is the default). With our choice of three fine-tuning iterations, this phase adds about 40% to the aforementioned running-times. That is, most of our examples need an additional time of up to 7 seconds for fine-tuning on the Convex. The SPARCstation is slower by approximately 120%. An average sized graph (20–25 nodes) runs for 30–40 seconds (annealing stage only).

[2]The first implementation was completed in late 1988, and was revised a number of times since. The present version is from late 1990.

We should emphasize three points. First, no sophisticated cooling schedule was incorporated to reduce the running time of the algorithm. Simple stopping rules could have been used to save a considerable amount of time. Had we decided to terminate a stage after a series of unsuccessful attempts to move nodes, we estimate that the average time for a stage would decrease by 10% to 20%. Furthermore, if the termination condition for the entire algorithm would be reaching a stage with no significant improvement in the energy function, we believe that the running time could be reduced by an additional 20% to 25%, on the average. Our observations, which are supported by the published material on this subject, indicate that the effects of the above savings on the quality of the final results are minor.

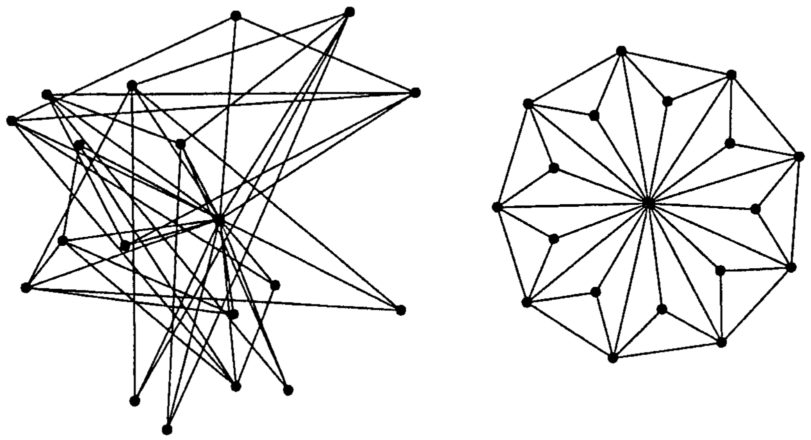
Second, the code was written to test the feasibility of applying annealing to the graph drawing problem, and not to serve as a practical, efficient implementation. Accordingly, no particular attention was paid to optimizing the code. In particular, it was not vectorized for the Convex, so that one of the strongest capabilities of the machine was not used. We are confident that rewriting some parts of the code and translating small portions into machine language could reduce the running time significantly. Some of the necessary changes involve adapting the code that deals with crossings of edges and distances between nodes and edges to integer arithmetic.

Third, and perhaps most importantly, simulated annealing is known to be a relatively time-consuming method for solving combinatorial optimization problems, and our implementation is consistent with this.

In its current state, the program cannot be integrated into an interactive system as a utility for drawing graphs. However, we have used it as the finetuning portion of the more powerful approach of Harel and Sardas [1995, 1996] discussed in Section 6. The program may also be used off-line to produce graphs for other systems. The ongoing and fast-growing research into the implementation and the acceleration of the SA scheme could widen the applicability of such a tool to different types of computer systems. (See van Laarhoven and Aarts [1987] for a few studies of parallel implementations.)

4. EXAMPLES

This section contains a series of examples that demonstrate the various features of the algorithm. The structure of the section is as follows: we first give a typical example, and then a more detailed one that shows some of the intermediate results. We then illustrate the various criteria for aesthetics by a series of simple graphs. To demonstrate the flexibility of our approach we show several drawings for the same graph, produced using different parameters. We have run our algorithm on most of the graphs that we found in other relevant publications (especially those discussed in detail in the next section). In general, our results compare favorably with these.



(a) (b)

Figure 1

Figure 1(a) is a typical input graph and Figure 1(b) is the output produced by the algorithm. Our implementation can accept a graphical version of an input graph, drawn by the user, or a textual adjacency matrix. In the latter case, the algorithm picks random locations for the nodes. Unless otherwise specified, the output graphs appearing in the article were generated by the default settings of the parameters for aesthetics, as discussed earlier. Several of the examples demonstrate the effects caused by changing these defaults.

It is worth noting that the output in Figure 1(b) is planar, although the algorithm has no *a priori* knowledge of its planarity, and it does not look explicitly for a crossing-free version. The contribution of the edge-crossings component to the energy function is significant enough (even by the default values) to yield crossing-free versions of many planar graphs of this size. Also, bearing in mind that the algorithm does not "understand" symmetry at all, the highly symmetric form of Figure 1(b) should be taken as evidence of the relevance of our simple heuristics to the problem of recognizing symmetry. Running the algorithm of Fruchterman and Reingold [1991] on this graph results in a nonplanar layout (see Figure 24 of Fruchterman and Reingold [1991]).

Here is another example of a planar graph. Figure 2(a) shows the input graph, again, with random locations for the nodes. Figure 2(i) is the final output, and Figure 2(b) through Figure 2(h) are seven intermediate steps. Most of the lengthy edges that appear in the input graph Figure 2(a) were eliminated in the first iteration and do not appear in Figure 2(b); the edge-length component of the energy function takes care of that. In Figure 2(d) most of the edge-crossings are eliminated and the rounded shape of the graph is already discernible. In Figure 2(f) there are no edge-crossings at

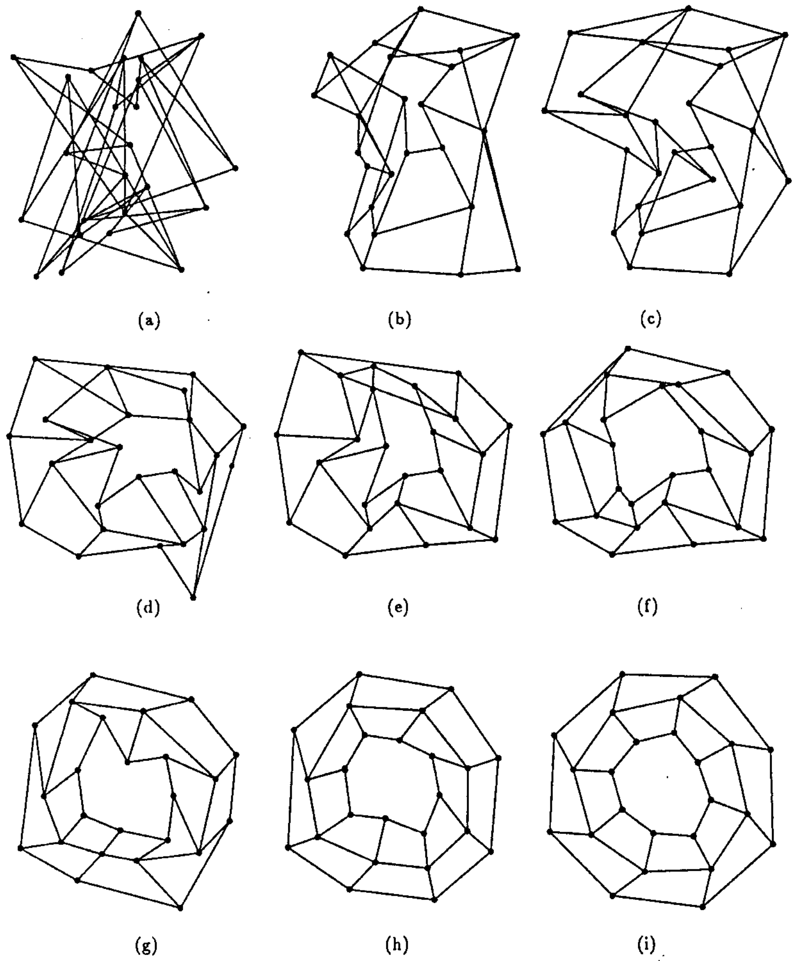


Figure 2

all. In Figure 2(g), which represents the end of the annealing part of the algorithm, there are still nodes that are unpleasantly close to edges. This is taken care of by the fine-tuning procedure, which is applied to 2(g), yielding 2(h) as an intermediate result and Figure 2(i) as the final result of the entire algorithm.

We now demonstrate the isolated effect of two of our criteria for aesthetics: the drive towards uniform node distribution and uniform edge lengths. Figure 3 shows the effect of the node distribution component on a graph of 70 disconnected nodes. Figure 3(a) shows a typical random distribution of

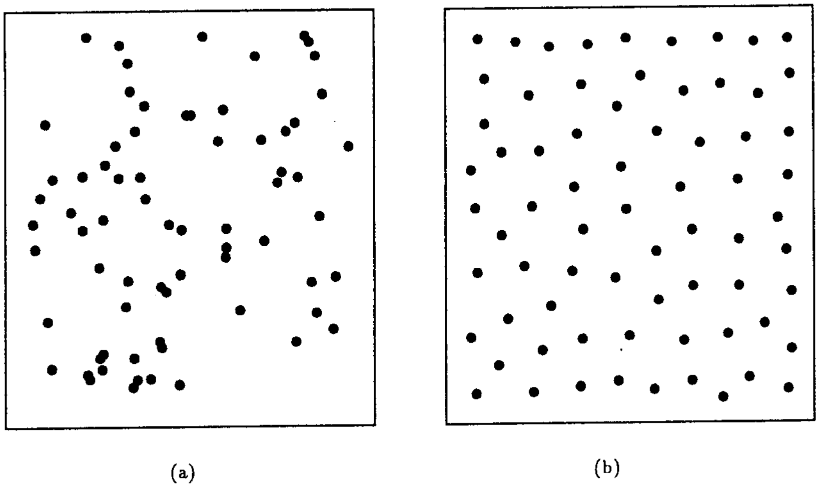


Figure 3

the nodes, while Figure 3(b) is the result of the algorithm. Although simulated annealing is driven by *random* relocation of nodes, it results in a pleasing even distribution.

Figure 4 is an example of how edges are spread over the drawing space, and it isolates the part of the cost function that prefers edges of uniform length. Figure 4(a) shows 35 disconnected edges, randomly drawn, and Figure 4(b) shows the result of applying the algorithm. The drawing space is used uniformly, and the edges are essentially of the same length.

Figure 5(a) shows the result of applying the algorithm to a graph that contains a closed cycle of 24 nodes. The drawing is crossing free, and occupies most of the drawing space (which is one of the reasons why it is not symmetric), but does not result in a circle, since the algorithm does not like to have nodes with only very few close neighbors and many nodes that are far away. It prefers a "curled up," fetus-like graph, in which nodes are generally closer to each other in a more uniform way. Figure 5(b) is a similar graph, but with 40 nodes. Here, the algorithm does not find a crossing-free solution, since our application of simulated annealing is based on trying to move single nodes; the algorithm doesn't seem to find a way to remove the crossing using only single-node relocations. Had we allowed multinode relocation (at a much higher cost), we would probably achieve better drawings in such cases.

As explained earlier, the implementation is flexible, and can be tuned to produce different results by varying the weights of the various aesthetics. We now present some examples illustrating this.

Figure 6 shows four different drawings of a regular *hexahedron* (cube). Figures 6(a) and 6(b) resemble projections of a 3-dimensional box from two different angles. Both drawings were obtained using the default settings

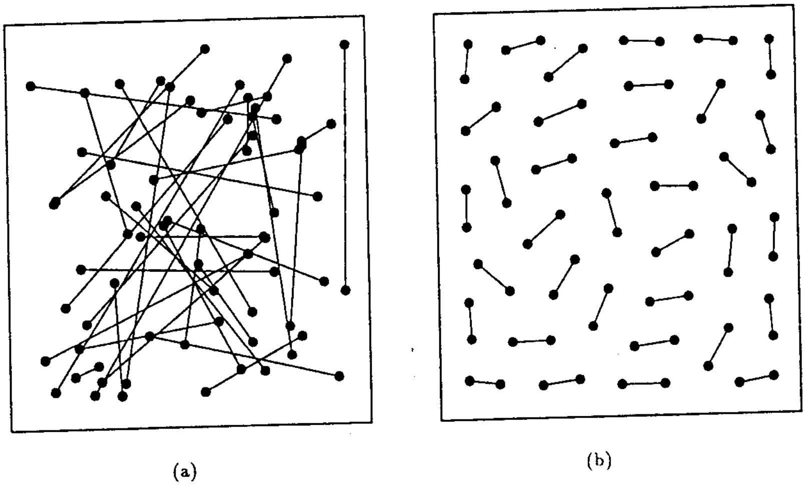

(a) (b)

Figure 4

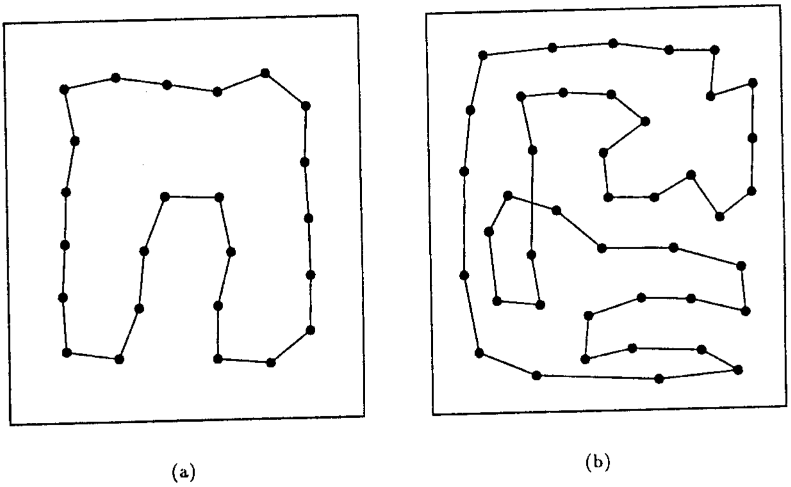

(a) (b)

Figure 5

(which also shows that different runs often produce somewhat different results, a consequence of the random nature of the annealing approach). Note that both Figures 6(a) and 6(b) have two crossings, and, while Figure 6(b) has regular edge lengths, in Figure 6(a) the average distance between a node and an edge is larger. In contrast, the crossing-free versions in Figures 6(c) and 6(d) were obtained by almost doubling the weight of the edge-crossing component. In Figure 6(d) the weight of the component that measures the distances between nodes and edges was lowered. This results

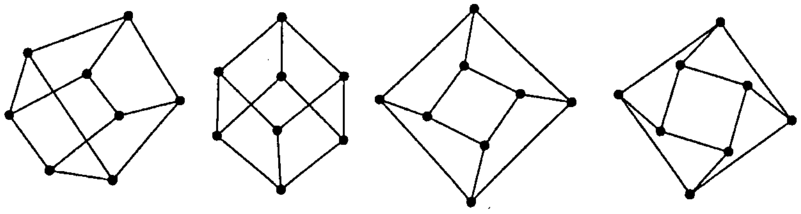


Figure 6

in lengthening the edges of the inner square and those that connect the inner and the outer squares, while shortening some of the distances between nodes and edges.

It is interesting to note that all the other drawing algorithms for which we have seen results on this graph [Eades 1984; Fruchterman and Reingold 1991; Kamada and Kawai 1989] draw it like our Figure 6(b), which is the conventional three-dimensional depiction. The flexibility in choosing the parameters in our algorithm allows us to produce different drawings, and it is reasonable to assume that such flexibility could be introduced into the algorithms of Eades [1984], Fruchterman and Reingold [1991] and Kamada and Kawai [1989] too.

The examples we have seen so far were all of planar graphs, for which the crossing-free drawing is usually the most pleasing one (although others can be acceptable too). Figure 7 contains the well known nonplanar graph K_{33}. Figure 7(a) depicts the standard way of drawing it, which we were not able to reproduce with the algorithm. Figure 7(b) was produced by the algorithm running with its default values, and Figure 7(c) was generated by disregarding the crossings component. Although more symmetric, Figure 7(a) turns out to be worse than both of these in terms of the criteria we use. Both Eades [1984] and Fruchterman and Reingold [1991] produce two different drawings for this graph (with different initial configurations), which are very similar to Figure 7(a) and Figure 7(c).

Our next example concerns the regular dodechahedron (with 12 pentagonal faces), which can be drawn in many different and interesting ways. Figure 8(a) is a symmetric and crossing-free picture of this graph, which, as far as we know, has not been generated by any of the algorithms we are aware of. Our algorithm produces this drawing, with only a very minor distortion, as shown in Figure 8(b). We also obtained two additional interesting drawings of this graph, given in Figures 8(c) and 8(d). The former was obtained with default parameters, and the latter by increasing the weight of uniform edge lengths. Both are reminiscent of a 3-dimensional soccer ball. Fruchterman and Reingold [1991] produces layouts similar to those of Figures 8(d) and 8(e) (which is also the one drawn by Kamada and Kawai [1989]). We can produce a very similar picture, with

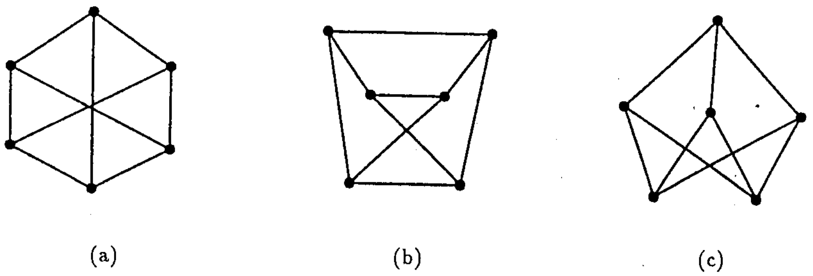


Figure 7

only five crossings, by drawing the inner 5-node circles one inside the other. Finally, Figure 8(f) shows the result of applying the algorithm in Lipton et al. [1985] to the dodecahedron. It may be viewed as an example of a case in which seeking mainly symmetry can yield less appealing results. (Compare with Figures 8(b) and 8(e).)

A similar phenomenon is demonstrated in Figure 9. Again, our algorithm fails to reproduce the symmetric crossing-free drawing shown in Figure 9(a). The reason is probably rooted in the variance of edge lengths required for this. Our crossing-free version, which is far less elegant, appears in Figure 9(b). For this graph, two additional runs of our algorithm yielded Figures 9(c) and 9(d), which are nonplanar, but are rather appealing projections of three-dimensional forms.

The basic version of our algorithm does not distinguish between different nodes; in our examples so far all nodes and edges were treated equally. However, we have implemented a (somewhat crude) way of preferring certain parts of the graph. We can mark some of the nodes in the graph, causing the edge-length component for all edges that are incident with marked nodes to be doubled. This makes such edges shorter than the typical edges of the graph. An edge incident with two marked nodes would become even shorter. Figure 10 shows an example of using this feature. Figure 10(a) is the regular grid, as generated by the algorithm without markings, and, in contrast, Figure 10(b) is the result of marking the four innermost nodes (the corners of the middle square).

The next example demonstrates one of the main differences between our algorithm, which does not assume any knowledge about the topology of the input graph, and other algorithms, which are tailored to deal with specific classes of graphs. Virtually all tree-drawing algorithms (some of which were discussed earlier) would draw a complete binary tree with the root at the top and the internal nodes arranged uniformly along increasingly lower horizontal lines. However, our algorithm treats trees as any other graphs, with no preference for aligning edges and no knowledge of symmetry. It thus produces Figure 11(a) for the complete four-level binary tree. Some simpler trees come out looking a little more symmetric, as in Figure 11(b). Trees also come out looking similar to these in the algorithms of Bernard [1981] and Manning and Atallah [1988]. In general, a good type-specific

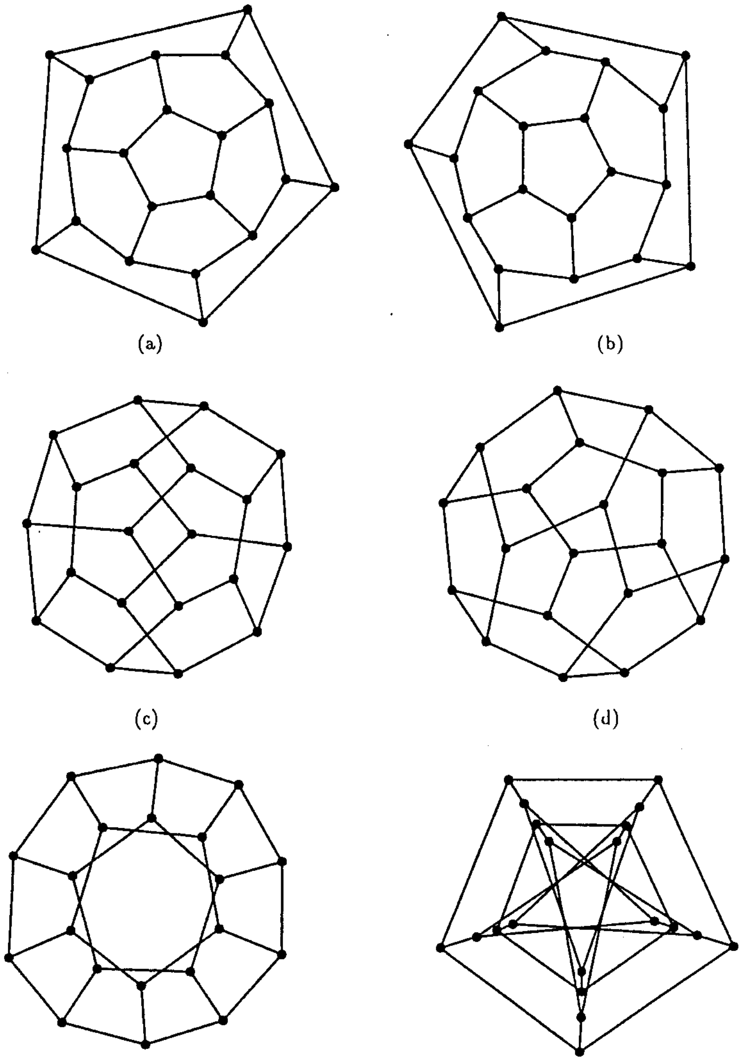


Figure 8

algorithm will perform better than a general-purpose algorithm on many of its target-type graphs.

To end our series of examples, we consider two larger graphs. Figure 12 shows the drawing of a graph containing 37 nodes and 68 edges. The graph

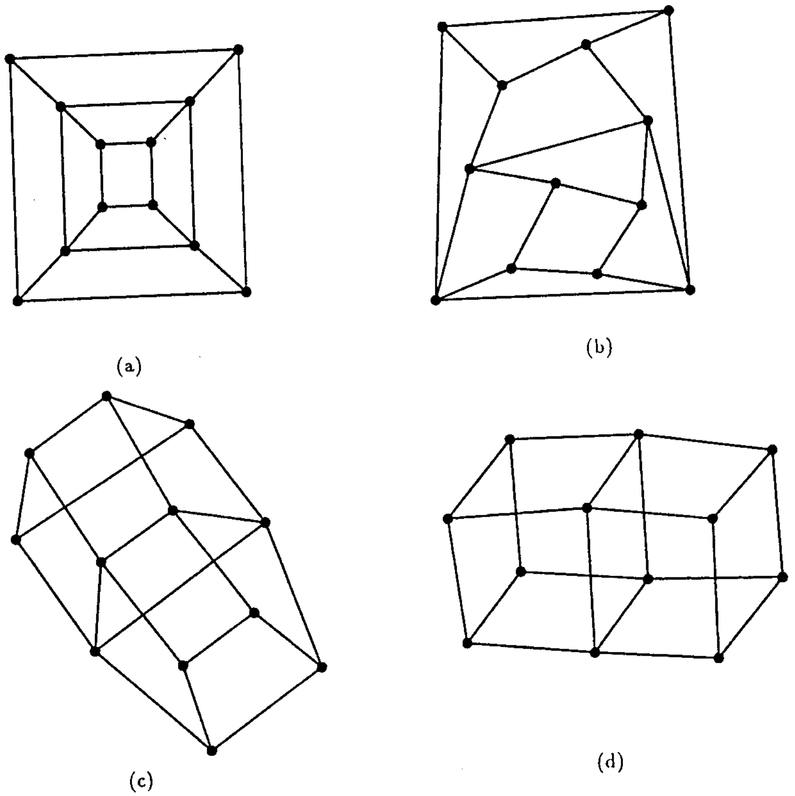
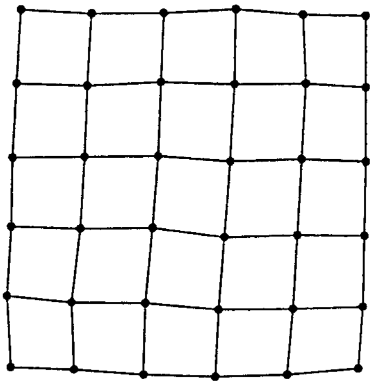


Figure 9

is nonplanar, but is nevertheless dealt with well by our algorithm (which, incidently, requires almost 60 seconds of Convex time on it). However, we do not always do as well on large graphs that are of a "looser" nature. The graph in Figure 13, for example, is a complete binary tree of depth six. It contains 63 nodes and 62 edges, and the running time to produce it was nearly two minutes. However, the drawing contains two crossings, and the topology of the tree is rather difficult to see.

5. COMPARISON WITH OTHER WORK

In this section we comment briefly on the connections between our approach and previous work on the problem of drawing general, undirected, straight-line graphs. The reader is referred to DiBattista et al. [1993] for an excellent annotated bibliography of all aspects of graph drawing.


(a)

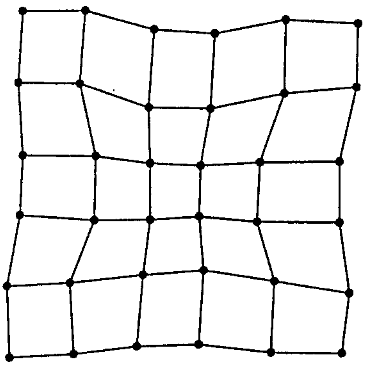
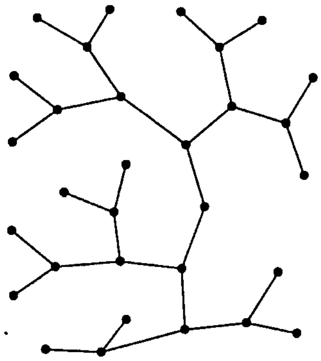

(b)

Figure 10


(a)

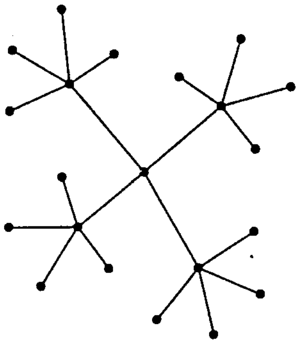

(b)

Figure 11

Eades [*1984*]. The most popular method for laying out undirected graphs is the so-called *spring method*, first proposed by Eades [1984].[3] This method likens a graph to a mechanical collection of rings (the nodes) and connecting springs (the edges). Two connected rings are attracted to each other or repelled by each other according to their distance and the proper-

[3]A different approach appears in Lipton et al. [1985], and involves finding symmetries in the input graph. In terms of the underlying ideas, it is incomparable to our method, and hence, except for a reference to it in the discussion of Figure 8, we shall not discuss it further here.

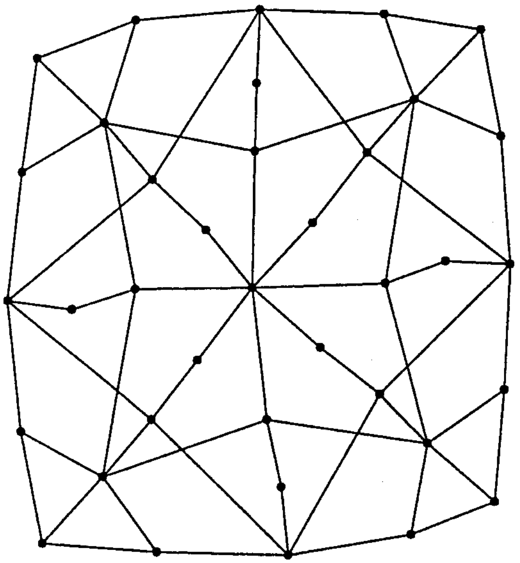


Figure 12

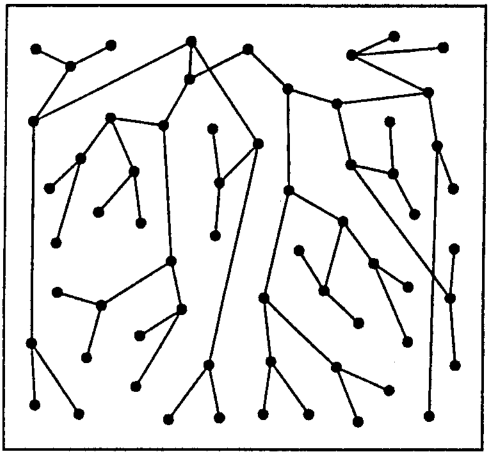


Figure 13

ties of the connecting spring. A state with minimal energy in the springs corresponds to a nice drawing of the underlying graph.

Two criteria for aesthetics were considered in Eades [1984]: the desire for uniform edge lengths, and trying to display as much symmetry as possible.

The calculations in Eades [1984] are carried out with an attraction force of

$$f_a(d) = k_1 \log(d/k_2)$$

applied to nodes connected by a spring, and a repelling force of

$$f_r(d) = k_3/d^2$$

applied to disconnected nodes. Here, the k_i are constants and d is the current distance between nodes (which, for connected nodes, is the length of the spring).

The initial configuration of the graph is chosen in Eades [1984] at random. In each iteration the forces are calculated for each node and nodes are moved accordingly. Almost all graphs achieve minimal energy within 100 iterations. The algorithm of Eades [1984] is reported to run quite fast on a VAX 11/780 on graphs with up to 30 nodes.

Kamada and Kawai [Kamada 1989; Kamada and Kawai 1989]. The algorithm presented in Kamada [1989] and Kamada and Kawai [1989] represents a slightly different approach to the spring method. For each pair of nodes i and j of distance d_{ij}, the algorithm calculates the energy of

$$k_{ij}(d_{ij} - l_{ij})^2,$$

where l_{ij} is the length of the shortest path in the graph between i and j, multiplied by a constant that represents the size of the drawing area. The spring constant k_{ij} is defined to be

$$k_{ij} = k/l_{ij}^2,$$

so as to help normalize the deviations from the desired length in the energy function. The energy of the entire graph is taken to be the sum of the above energy over all pairs i, j, and this is the quantity to be minimized.

Since finding a global minimum is difficult, a local minimum is first sought, by repeatedly moving the node that provides the maximal decrease in the total energy. These iterations end when the maximal improvement is less than some fixed ϵ. After this, the algorithm tries to exchange each pair of nodes, restarting the search for a local minimum if this exchange yields another decrease in the energy.

The time complexity of the algorithm in Kamada [1989] and Kamada and Kawai [1989] cannot be expressed as a function of the size of the graph, since it also depends on the initial configuration, on ϵ, and on the graph itself. The examples given in Kamada [1989] and Kamada and Kawai [1989] contain up to 20 nodes, and ran within 10 seconds on a VAX 8600.

Kamada [1989] and Kamada and Kawai [1989] also suggest a way of dealing with weighted graphs (i.e., ones with numeric weights on the edges), by redefining l_{ij} to be the sum of the weights on the shortest path between i and j. They note that their algorithm usually draws isomorphic graphs in roughly the same way (up to obvious rotations and transpositions), which means that it essentially yields a heuristic for solving the graph isomorphism problem.

Fruchterman and Reingold [*Fruchterman and Reingold* 1991]. A recent paper on the spring method is Fruchterman and Reingold [1991], which constitutes an enhancement and adaptation of Eades [1984].[4] As in Eades [1991], attraction forces are calculated only for neighboring nodes, and repelling forces are calculated for all pairs of nodes. The attraction force for nodes at distance d is

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

and the repelling force is

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

Here, k is the optimal distance between nodes in the graph, calculated from the number of nodes and the size of the drawing area.

The process in Fruchterman and Reingold [1991] is carried out iteratively. At each iteration, the forces are calculated for each node, and at its end all nodes are moved simultaneously. Following decisions we had taken in the SA approach, the range of the allowed movement of nodes in Fruchterman and Reingold [1991] was made to decrease as the algorithm proceeds. Moreover, the rate of this decrease was made to be high in the early stages, and towards the end it is constant and small, as in our fine-tuning stages.

The algorithm in Fruchterman and Reingold [1991] uses 50 iterations, and all the examples presented therein were drawn in 10 seconds or less on a SparcStation.

Discussion. Our approach in this article is similar to that of the spring method. Both are based essentially on physical systems, and use simple attraction and repelling criteria iteratively to try to find a layout that is "content." The way the cost, or energy, is minimized in our algorithm is more akin to that of Kamada [1989] and Kamada and Kawai [1989] than to those of Eades [1984] and Fruchterman and Reingold [1991]. In Kamada [1989] and Kamada and Kawai [1989], energy is minimized in stages, node after node, as in SA, but in a deterministic way. The attempt to escape from a local minimum is carried out there by exchanging the locations of two nodes. A nice property of the algorithm in Kamada [1989] and Kamada and Kawai [1989] is the need for only one expression in the energy function (the difference between the actual distance between nodes and the shortest distance in the graph).

[4]Our work and that of Fruchterman and Reingold [1991] were carried out independently. The final version of each was written with each set of authors reviewing early versions of the other's work.

In terms of the iterative nature of the solution, our algorithm is closest to that of Fruchterman and Reingold [1991], as discussed above. The algorithm of Fruchterman and Reingold [1991] has a variable controlling the iterations (the temperature); it continuously limits allowed node movement as a function of the temperature; it has a set of final iterations with constant temperature for fine tuning; and it has special provisions for taking into account the drawing area. However, Fruchterman and Reingold [1991] differ from Kamada [1989] and Kamada and Kawai [1989] and from our approach in that all nodes are moved together, making it possible to reach configurations that are not necessarily in the vicinity of the current local minimum.

It is interesting to note the contrast between the generality of the cost function in our approach and the rigidity of the optimization method we have adopted. Simulated annealing is an integral part of the method, for better or for worse. On the other hand, there is flexibility in the construction of the cost function: other criteria for aesthetics could probably be added without too much trouble, and the relative weights of the criteria can be varied, thus making it possible to have some control over the final appearance of the graph. We have not carried out a detailed evaluation of the importance of the various criteria, and it seems that in many cases only one or two of them suffice, as is evident from some of the papers discussed above.

Since our exposition is based mainly on test-case examples, there is no point in making absolute judgements in comparing our approach to others. The results show that the SA method is competitive in terms of the quality of the resulting layouts, although, as mentioned earlier, it inherits its unattractive running time from the general framework of simulated annealing. We are confident that a more serious attempt at optimization would result in much better time performance. For large graphs (say, over 60 nodes or so) our method does not perform well. However, we suspect that this is common to all the methods that address the problem of laying out general graphs aesthetically, and that a significant improvement will be possible only with a radically different approach, perhaps incorporating parallel processing.

6. CONTINUATION AND FUTURE WORK

Using SA for fine-tuning only. As a continuation of the work reported upon here, we have completed a more ambitious project, reported upon in Harel and Sardas [1995; 1996], in which the graph is subjected to simulated annealing only after a rather complex series of preprocessing stages. The first is to test for planarity, using a method based on PQ-trees. If the graph is planar, the system finds a planar embedding, which it then proceeds to draw using a specially tailored drawing algorithm. It then fine-tunes the solution using the SA algorithm of the present paper.

If the graph is not planar, another version of the PQ-tree algorithm is employed to try to find a maximal planar subgraph, by eliminating as few edges as possible. An embedding is then found for this planar subgraph as in the planar case, after which we reinsert the eliminated edges while trying to minimize the number of crossings that arise. At each crossing point a dummy node is inserted, yielding again a planar graph. This graph is then drawn according to the algorithm of Harel and Sardas [1996], and is fine-tuned by the SA algorithm. Finally, the dummy nodes are removed and edges are straightened out.[5]

This approach achieves significant improvements over the bare SA algorithm, in the quality of the resulting layouts, in the size of graphs that are handled well, and in the time performance. The improvements are most readily observed in planar or close-to-planar graphs. We feel that this continuation work beneficially exploits the conclusions of the research described in the present (earlier) article: Simulated annealing can indeed be used in graph drawing, but is probably better employed in a tandem system whose front-end contains more specific heavy-duty tools for finding a reasonable first-cut solution.

Nonrandom initialization and partial changes. It is often the case that we have some *a priori* knowledge about the way a given graph should be drawn. The graph may have been sketched before, subgraphs may have been required to be drawn in separate areas, and so on. In general, any such prior information can be used to start the algorithm in a given configuration, rather than in a random one. When the initial configuration of the algorithm is nonrandom and closer to the final desired result, the initial temperature should be set low enough, to prevent total randomization at the beginning. Starting with an overly high temperature might cause the initial good configuration to be lost.

Another slight modification is to allow only part of the drawing to be changed. This option is useful when adding or deleting nodes from a given graph. If the graph has already been drawn to our satisfaction, there is no need to redraw it in its entirety. In this case, we can restrict the changes by marking the nodes that are allowed to change place (an option we have implemented and experimented with), or by limiting the area in which changes are allowed. A modified version of the algorithm that chooses new possible configurations under these restrictions is then run. This option is also useful for correcting imperfect pictures: the area to be redrawn is marked, and the algorithm is rerun with high initial temperature, but restricted to the area of interest. We have implemented a rather crude version of this idea, and our experience shows that this often improves unsatisfactory drawings produced by a standard run of the algorithm.

Modifying the neighborhoods. Our definition of neighborhoods allows only one node to be repositioned at a time, and we find this definition to be satisfactory on relatively small graphs. However, there are examples in which the algorithm is stuck in a bad configuration, which no repositioning

[5]The general framework of this approach has been proposed in Batini et al. [1984] and Tamassia et al. [1988], though the algorithms used and many of the details are different.

of a single node appears to improve. (The loop in Figure 5(b) is an example of this.) In these cases, moving an edge, rather than a node, can often deliver the algorithm from the local minimum. The uphill moves of the annealing algorithm are a powerful tool for escaping these minima, but to assure quicker, and sometimes better, convergence, whole edges (together with those incident to them) should be allowed to be repositioned. We believe that this extension will reduce the number of steps, but will also complicate the calculations that update the energy function.

The idea may be further extended by allowing arbitrary pairs of nodes to be repositioned, or even allowing whole connected components to change positions. In order to retain its current shape, the layout of the moved component should be maintained (up to a rotation or a mirror image). We have not implemented these suggestions; they might result in an unacceptable growth in the size of the neighborhoods, and the cost in running time might not be worth the potential improvement of the results.

Parallelism and multilevel methods. Another interesting direction for future work is to try to use parallel processing to expedite the SA process, perhaps coupled with a multilevel approach to the entire problem. Results of running an annealing algorithm on parts of the graph, that are carefully chosen according to multilevel criteria, would be combined, and the process would then be carried out again on a higher level of abstraction.

Additional criteria for aesthetics. If the computational price of the algorithm becomes significantly lower, more imaginative aesthetics than the ones we have adopted could be considered. Symmetry emerges as an important attribute of a picture, emphasizing the inner structure of the graph. A natural requirement would be to draw similar subgraphs in a similar way. However, measuring the symmetry of a given picture requires a quantitative definition, and efficient methods must be developed if this is to be done within the annealing iterations. We should also remember that the obviously relevant problem of telling whether two graphs (or subgraphs) are isomorphic is still not known to be solvable in polynomial time.

Another criterion might deal with the relative positions of edges. We have limited ourselves to measuring the distances between nodes and edges. If we could say something deeper about the relationship between neighboring edges, no doubt the result would be nicer. We could measure the angles between adjacent edges; we might be able to decide, for example, if we prefer a chain of four edges to be drawn on a straight line, or as a half-octagon. If angles were considered, we could also influence the orientation of the edges relative to the borderlines.

Other output conventions. Our output graphs are drawn with straight-line edges. Other conventions have been uses in the literature, and some might be easier to implement within our framework. We might consider the grid standard, that requires nodes to be positioned on (coarsely distributed) grid points and edges to be composed of grid segments. The locations for nodes are restricted, but drawing the edges is less immediate. The choice of the path that an edge follows has some degrees of freedom, and we can introduce such choices into our definition of neighborhoods. This problem becomes similar to VLSI layout problems that have been successfully treated using SA.

Another relaxation of the straight-line standard is to allow an edge to be drawn as a continuous curve. We can use the SA method to deal with the following approximation to this convention: break each edge into a bounded number of pseudo-edges by introducing dummy intermediate nodes. We may now run the basic algorithm, and modify the resulting drawing by recombining the pseudoedges into smooth curves using a spline method [de Boor 1978; Ganser et al. 1988].

Hypergraphs and higraphs. It would be nice to address more complicated graphical objects. Here are some preliminary ideas for dealing with hypergraphs [Berge 1973] and higraphs [Harel 1988]. (Some work on hypergraphs appears in Johnson and Pollack [1987] and Mäkinen [1990].)

An edge in a hypergraph (sometimes called a *hyperedge*) is simply a subset of the nodes, and is not restricted to size two as in a graph. To deal with these, we can replace a hyperedge connecting N nodes by a dummy node and N binary edges from it to its adjacent nodes. The algorithm is then applied to this standard graph, following which the dummy nodes are redrawn in the particular form adopted for hyperedge centers. Although this is a very simple idea, it might produce reasonable results, because our algorithm tends to bring connected nodes close together. Thus, the nodes that are adjacent to an edge in the hypergraph will tend to group in the same area of the picture.

Higraphs are more complicated. They involve multilevel "blobs" that can intersect or enclose each other, as in Euler circles and Venn diagrams; they employ a partitioning convention for representing Cartesian products, as well as interlevel edges or hyperedges. Higraphs appear to be useful in a number of applications (see, e.g., Harel [1988], Harel et al. [1990] and Heydon et al. [1990]), which further motivates the search for automatic drawing procedures. However, higraphs pose many additional difficulties for nice drawings. The set-theoretic relationships represented by the blobs must be retained in the final result, whereas keeping these from being violated during the intermediate stages of the annealing procedure might severely restrict its usefulness. New criteria for aesthetics must be defined, taking into account acceptable distribution and the relative sizes and shapes of subblobs, avoiding the complications that arise when there are many intersections but at the same time trying to keep the edges short and simple-looking. The edges would probably have to be curved, and with many splined turns. We have some ideas as to how to proceed and hope to address the problem in the near future.

ACKNOWLEDGMENTS

We thank Mike Atkinson, Sandeep Bhatt, Jim Drinkwater, John Hershberger, David Johnson, Joseph Manning, Eli B. Messinger, James E.

Narem Jr., Steven C. North, Roberto Tamassia and Howard Trickey for bringing some of the references to our attention. We are grateful to the referees for their helpful comments.

REFERENCES

BERGE, C. 1973. *Graphs and Hypergraphs*. North-Holland, Amsterdam.

BERNARD, M. A. 1981. On the automated drawing of graphs. In *Proceedings of the 3rd Caribbean Conference on Combinatorics and Computing*, 43–55.

DE BOOR, C. 1978. *A Practical Guide to Splines*. Springer-Verlag, New York.

BATINI, C., TALAMO, M., AND TAMASSIA, R. 1984. Computer-aided layout of entity-relationship diagrams. *J. Syst. Softw. 4* 163–173.

DI BATTISTA, G., EADES, P., TAMASSIA, R., AND TOLLIS, I. G. 1993. Algorithms for drawing graphs: An annotated bibliography. Preprint, Dept. of Computer Science, Brown Univ., Providence, R.I., Nov. To appear in *Comput. Geom. Theory Appl.*

EADES, P. 1984. A heuristic for graph drawing. *Cong. Numer. 42*, 149–160.

FRUCHTERMAN, T. M. G. AND REINGOLD, E. 1991. Graph drawing by force-directed placement. *Softw. Pract. Exper. 21*, 1129–1164.

GANSNER, E. R., NORTH, S. C., AND VO, K. P. 1988. DAG—A program that draws directed graphs. *Softw.—Pract. Exper. 18*, 1047-1062.

GAREY, M. R. AND JOHNSON, D. S. 1983. Crossing number is NP-complete. *SIAM J. Alg. Discrete Meth. 4*, 312–316.

HAJEK, B. 1985. A tutorial survey of theory and applications of simulated annealing. In *Proceedings of the 24th Conference on Decision and Control*, 755–760.

HAREL, D. 1988. On visual formalisms. *Commun. ACM 31*, 514–530.

HAREL, D., LACHOVER, H., NAAMAD, A., PNUELI, A., POLITI, M., SHERMAN, R., SHTUL-TRAURING, A., AND TRAKHTENBROT, M. 1990. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Trans. Softw. Eng. 16*, 403–414.

HAREL, D. AND SARDAS, M. 1995. Randomized graph drawing with heavy-duty preprocessing. *J. Visual Lang. Comput. 6*, 233–253.

HAREL, D. AND SARDAS, M. 1996. An incremental drawing algorithm for planar graphs. *Algorithmica*, to appear.

HEYDON, A., MAIMONE, M. W., TYGAR, J. D., WING, J. M., AND ZAREMSKI, A. M. 1990. Miró: Visual specification of security. *IEEE Trans. Softw. Eng. 16*, 1185–1197.

JOHNSON, D. S., ARAGON, C. R., MCGEOCH, L. A., AND SCHEVON, C. 1989. Optimization by simulated annealing: An experimental evaluation, Part I (Graph partitioning). *Oper. Res. 37*, 865–892.

JOHNSON, D. S., ARAGON, C. R., MCGEOCH, L. A., AND SCHEVON, C. 1991. Optimization by simulated annealing: An experimental evaluation, Part II (Graph coloring and number partitioning). *Oper. Res. 39*, 378–406.

JOHNSON, D. S. AND POLLACK, H. O. 1987. Hypergraph planarity and the complexity of drawing Venn diagrams. *J. Graph Theory 11*, 309–325.

KAMADA, T. 1989. *Visualizing Abstract Objects and Relations*. World Scientific, Teaneck, N.J., (See also On visualization of abstract objects and relations. D. Sc. Thesis, The University of Tokyo, Dec. 1988.)

KAMADA, T. AND KAWAI, S. 1989. An algorithm for drawing general undirected graphs. *Inf. Proc. Lett. 31*, 7–15.

KIRKPATRICK, S., GELATT JR., C. D., AND VECCHI, M. P. 1983. Optimization by simulated annealing. *Science 220*, 671–680.

VAN LAARHOVEN, P. J. M. AND AARTS, E. H. L. 1987. *Simulated Annealing: Theory and Applications*, D. Reidel, Dordrecht.

LIPTON, R., NORTH, S., AND SANDBERG, J. 1985. A method for drawing graphs. In *Proceedings of the ACM Symposium on Computational Geometry*, 153–160.

MAKINEN, E. 1990. How to draw a hypergraph. *Int. J. Comput. Math. 34*, 177–185.

MANNING, J. AND ATALLAH, M. J. 1988. Fast detection and display of symmetry in trees. *Cong. Numer. 64*, 159–169.

METROPOLIS, N., ROSENBLUTH, A., ROSENBLUTH, M., TELLER, A., AND TELLER, E. 1953. Equation of state calculations by fast computing machines. *J. Chem. Phys. 21*, 1087–1091.
SIARRY, P., BERGONZI, L., AND DREYFUS, G. 1987. Thermodynamic optimization of block placement. *IEEE Trans. Comput. Aided Des. 6*, 211–221.
TAMASSIA, R., DI BATTISTA, G., AND BATINI, C. 1988. Automatic graph drawing and readability of diagrams. *IEEE Trans. Syst. Man Cybern. SMC-18*, 61–79.
TUTTE, W. T. 1963. How to draw a graph. In *Proceedings of the London Mathematical Society 13*, 743–768.

Received (by *Commun. ACM*) July 1989; final revision Nov. 1994 (*Commun. ACM*); accepted (*TOG*) February 1996