Automatic Graph Drawing Algorithms

Susan Sim
simsuz@turing.utoronto.ca
December 17, 1996.

Embeddings of graphs have been of interest to theoreticians for some time, in particular those of planar graphs and graphs that are close to being planar. One definition of a planar graph is one that can be drawn in the plane with no edge crossings. While working on the four-colour problem, Wagner(1936) was the first to show that every planar graph has a straight-line embedding. Tutte (1960, 1963) showed that every 3-connected planar graph has a convex embedding. While graph theory was originally an artifact from mathematics, it has become quite prevalent as a means of solving problems or representing data. With automatically generated data sets being represented as graphs, came the need to automatically generate embeddings of graphs in a 2-dimensional space, such as a computer terminal or a sheet of paper. A wide variety of fields each with their own requirements utilize automatic graph drawing algorithms. As a result, many different algorithms have been developed over the last decade. Within the last five years, there have been annual conferences on graph drawing (Di Battista, et al., 1993; Tamassia and Tollis, 1995; Brandenberg, 1996), special issues of journals on the topic and this year a monograph was published. (Cruz and Tamassia, 1994) Three different algorithms from the field will be presented briefly in this paper: the spring model algorithm, simulated annealing method and the Sugiyama algorithm.

Applications of Graph Drawing Algorithms

Work in this area was done mainly in response to requirements of data visualization techniques and interactive computer systems. Many fields in computer science, such as software engineering, electronic circuit design and database design, have found it useful represent data as graphs, with vertices denoting elements and edges denoting relations between them. These graphs are normally generated by software tools based on

information in the system. As the size of a graph generated from data or constraints grew, so has the sophistication of embedding algorithms.

In software engineering, the architecture of a large software system can be visualized as a directed graph with vertices representing modules and edges denoting various use relations between them. These systems are often hierarchical in nature and their drawings should reflect this. Furthermore, this information can be used to make the graph drawing task easier. (Müller, et al., 1993)

Computer hardware and microchips are now sufficiently complex that they are designed using CAD tools. It is then the responsibility of the tool to create a layout of the logical gates and the connections between them on microchips and circuit boards. This layout should be a *grid* drawing. An *orthogonal* drawing is one in which an edge is a chain of horizontal and vertical segments. A *grid* drawing is an orthogonal drawing in which all the vertices and bends of edges have integer coordinates. (Di Battista, et al., 1994)

There are many other examples of applications which use graph drawing algorithms. Entity-relationship diagrams in database design can have a visual representation or an algebraic one. One of the requirements of systems analysis and design tools is that a database description need only be entered once in either format and the alternate format will be generated. There is a project management technique that uses PERT charts (Project Evaluation and Review Technique) to track dependencies among tasks. These dependencies form a directed graph from which other information can be derived, such as a project critical path. One technique used by the Human Genome Project analyses the gene structure by representing raw data as a directed graph (Harley and Bonner, 1994).

Aesthetics

It is not difficult to design a naïve algorithm to display a graph. A random layout places vertices randomly within a finite space. Edges can be drawn as minimum length straight lines between vertices or they may be *polylines*, that is, lines with bends in them to avoid

drawing elements. A circular layout algorithms places the vertices along the perimeter of a circle and edges are drawn across the circle. A similar strategy places vertices at the intersections of an $n \times n$ grid along the main diagonal. (Noik, 1996)

The circular layout method is particularly effective for representing cliques as it emphasizes the regularity of the graph. Unfortunately, most graphs are not cliques. None of the above methods pays much attention to the readability of the resulting graph, either as a collection of lines and dots or the data that they represent. From a graph theoretic point of view, isomorphic graphs should look similar. Also, planar graphs should be drawn without edge crossings so it is easy to visually confirm the planarity of the graph. Consequently, the drawings generated by the above strategies may not be very informative.

The quality or usefulness of a particular embedding is highly dependent on its application domain. Therefore a graph drawing algorithm must take into account *aesthetics*: criteria for making salient characteristics of the graph easily readable. Readability and "salient characteristics" are highly subjective and dependent on the purpose for which the drawing is generated. Some aesthetic criteria include:

- minimize the number of edge crossings;
- draw edges as straight as possible;
- vertices should be evenly distributed;
- the majority directed edges should be drawn pointing in the same direction;
- in polyline drawings, minimize bends in the edges;
- minimize the area of the area drawing;
- maximize display of symmetries;
- maximize *angular resolution*.

(Di Battista, et al., 1994; Eades and Sugiyama, 1990; and Cruz and Tamassia, 1994) The angular resolution of a line drawing of a graph is the smallest angle formed by two edges incident on the same vertex. If this angle is too small, it may be beyond the resolution of

graphic display device or even the human eye, and the two incident edges end up looking like a fuzzy blob.

In general, it is not possible to optimize two criteria simultaneously. For example, in the two embeddings of K_4 below, the one on the left minimizes the number of edge crossings, whereas the one on the right maximizes the display of symmetries. (Cruz and Tamassia, 1984)

Many of these optimization problems are either NP-hard or NP-complete. Minimizing the number of crossings in an embedding is NP-complete, even if the graph is hierarchical with only two layers. (Garey and Johnson, 1983, in Eades and Sugiyama, 1990; and Di Battista et al., 1994) Minimizing the area of a grid drawing is NP-hard. (Kramer and van Leeuwen, 1984, in Di Battista et al., 1994), as is minimizing the length of the maximum edge length (Miller and Orlin, 1984, in Di Battista et al., 1994).

Polytime graph drawing algorithms tend to either use heuristics to approximate an NP-hard optimization, use innovative techniques to manipulate the layout or some combination of the two. The more complex algorithms that yield more aesthetically pleasing graphs are usually able to approximate optimizations of several criteria. The specific criteria used are often determined by the specific application domain or graph type for which the algorithm was developed. Three such algorithms will be discussed in the remainder of this paper. They are the Spring Model Algorithm, Simulated Annealing Algorithm, Sugiyama Algorithm.

Spring Model Algorithm

There are many different strategies that can be used to draw a general undirected graph. One method is to use a planar embedding algorithm because a planar embedding can be constructed in linear time. The first step is to test for planarity and if the test is positive construct the embedding. If the graph is not planar, then it can be *planarized* by a variety of techniques such as deleting edges, splitting dummy vertices or adding dummy vertices at edge crossings. Planarity testing can be done in linear time. Planarization is an NP-hard problem but can be done using heuristics in $O(n^2)$ or less. So overall, this method is relatively efficient. (Di Battista, et al., 1994; Cruz and Tamassia, 1994)

Another method is to orient the edges, say using an Eulerian walk and use a directed graph drawing algorithm. Finally, there is a family of force-directed algorithms which involves transforming the vertices and edges into a system of forces and finding the minimum energy state of the system. This state is found either by solving differential equations or by running a simulation of the forces. The spring model is the most popular algorithm in this family. The spring model was originally developed by Eades, 1984. The version presented here is a descendant of that algorithm developed by Kamada (1989).

Conceptually, the spring embedder works by replacing edges with springs unit some natural length. It also adds springs with larger natural lengths between non-adjacent vertices. The vertices are initially placed randomly and a system of differential equations is solved to find the system state with minimum energy. By nature, springs attract their endpoints when stretched and repel their endpoints when compressed. Vertices that are adjacent are kept close to each other by shorter springs. While the longer springs between non-adjacent vertices keep them apart, and at the same time they limit the overall size of the embedding.

The total energy of the system is represented by the following summation:

$$E = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \tfrac{1}{2} k_{ij} \left(\left| p_i - p_j \right| - l_{ij} \right)^2$$

$p_1, p_2, \ldots, p_n$ are particles on a plane representing vertices $v_1, v_2, \ldots, v_n$

l_{ij} is the natural length of the spring between v_i and v_j and is defined as $l_{ij} = L \times d_{ij}$, where L is the desired length of the edge in the embedding and d_{ij} is the shortest length path between v_i and v_j in the graph. L is sometimes chosen as a function of the diameter of the graph and the available embedding area.

k_{ij} is the strength of the spring between v_i and v_j and is defined as $k_{ij} = K / d_{ij}^2$, where K is a constant and d_{ij} is as defined above.

Rewriting the energy summation in terms of xy-coordinates, we get:

$$E = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \tfrac{1}{2} k_{ij} \left((x_i - x_j)^2 + (y_i - y_j)^2 + l_{ij}^2 - 2l\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \right)$$

To find local minima of this summation, we need to take partial derivatives of this and solve to find local minima for each vertex. This results in 2*n* non-linear non-independent equations. This system of equations is rather difficult to solve, so the problem is considered one particle at a time. The particle in the system with the highest energy is identified and moved to a stable location with low energy. Let this particle be p_m, with energy function:

$$\Delta_m = \sqrt{\left(\frac{\partial E}{\partial x_m} \right)^2 + \left(\frac{\partial E}{\partial y_m} \right)^2}$$

p_m is moved in a stepwise manner to minimize Δ_m. This location corresponds to where $\frac{\partial E}{\partial x_m} = \frac{\partial E}{\partial y_m} = 0$ is satisfied. This is done by iteratively solving the following two linear equations for δx and δy, where t is the current iteration and adding them to x_m and y_m respectively. These steps are repeated until the new energy function stops decreasing.

$$\frac{\partial^2 E}{\partial x_m^2}\left(x_m^t, y_m^t\right)\delta x + \frac{\partial^2 E}{\partial x_m \partial y_m}\left(x_m^t, y_m^t\right)\delta y = \frac{-\partial E}{\partial x_m}\left(x_m^t, y_m^t\right)$$

$$\frac{\partial^2 E}{\partial x_m \partial y_m}\left(x_m^t, y_m^t\right)\delta x + \frac{\partial^2 E}{\partial y_m^2}\left(x_m^t, y_m^t\right)\partial y = \frac{-\partial E}{\partial y_m}\left(x_m^t, y_m^t\right)$$

The algorithm is summarized by the following pseudocode.

1. compute d_{ij} for $1 \le i \ne j \le n$;
2. compute l_{ij} for $1 \le i \ne j \le n$;
3. compute k_{ij} for $1 \le i \ne j \le n$;
4. initialize $p_1, p_2, \ldots, p_n$
5. while (max $\Delta_i > \varepsilon$)
6. let p_m be the particle satisfying $\Delta_m = \max \Delta_i$;
7. while ($\Delta_m > \varepsilon$)
8. compute δx and δy;
9. $x_m \leftarrow x_m + \delta x$
10. $y_m \leftarrow y_m + \delta y$
11. end while
12. end while

When a local minimum has been found, each pair of particles are exchanged and the energy of the system is tested. If a swap results in a lower energy state, the energy minimization process restarted with the new configuration as a starting state. This exchange and compare process provides a means of escaping large local minima. Eventually the system converges to a global minimum.

It takes $O(n^3)$ time to find all pairs shortest paths. More efficient algorithms may be used to lower this bound. O(n) time is needed to compute each of Δ_m, δx and δy during each iteration. With some bookkeeping, $\max \Delta_i$ can be found in O(1) time. The time required by the energy minimization process is O(Tn), where T is the total number of inner loops. T is difficult to characterize beyond this because it depends on the value of ε, the initial position of the vertices, and the graph itself.

This algorithm reduces the number of edge crossings as their presence increases the energy in the system. Also, drawings produced by this method display symmetries present in the graph. The same graph with different initial conditions will converge to the same drawing. Graphs with similar structures will also be drawn similarly. This method can be extended to layered hierarchical graphs by assigning vertices in the same level a fixed *y* value and allowing only the *x* position to vary.

Simulated Annealing Method

In the most general terms, annealing is the process by which some substance is heated until it is a liquid and then slowly cooled until it is a solid. The slow cooling allows the molecules to organize themselves into a crystal, a totally ordered form. This process is often used in the manufacturing of products such as steel. Simulations of annealing were developed to analyze the efficiency of these manufacturing processes. Simulated annealing is used as a problem solving technique where the potential solution space is large and a combinatorial search is infeasible. It has been applied with some success to problems such as VLSI circuit design, graph partitioning and the traveling salesman problem. Davidson and Harel (1996) have applied this technique to drawing undirected graphs with straight-line edges. One big advantage of this algorithm is that the relative importance of different aesthetic criteria can be varied.

The basic shape of a simulated annealing algorithm is given in the pseudocode below.

1. Initialize temperature T;
2. Initialize configuration σ;
3. E ← cost of σ;
4. while(min T not reached)
5. while(termination rule is not satisfied)
6. choose a new configuration σ′ from the neighbourhood of σ;
7. E′ ← cost of σ′;
8. if ((E′ < E) OR (*random* < $e^{(E-E')/T}$)) then
9. σ ← σ′;
10. E ← E′;
11. end if
12. decrease temperature T
13. end while

14. end while

A *configuration* is a proposed embedding of the input graph. The graph can be entered either as a set of adjacencies or as a hand drawing. If the input is a set of adjacencies then the initial configuration is randomly generated.

A *neighbourhood* of a configuration σ, is the set of configurations that differ from σ by the location of a single vertex. σ′ is generated by taking a vertex and placing it on the perimeter of a circle drawn around its original location. To simulate the behaviour of molecules in the physical annealing process, the size of this circle is initially large and decreases with T.

The *cost* of a configuration must be carefully chosen so that it reflects the desired aesthetics of the final graph and is not overly computationally intensive to compute.

1. To ensure that vertices are evenly distributed to avoid overcrowding, the term $a_{ij} = \frac{\lambda_1}{d_{ij}^2}$ is added to the cost function pair of vertices, *i, j*. d_{ij}^2 is the Euclidean straight line distance between *i* and *j*. λ_1 is a weighting factor whose value depend in the importance of this criteria relative to the others in the cost function.
2. A completely minimized cost function will spread out the vertices indefinitely. To ensure that vertices are drawn on the display area, the following term is added to the cost function for each node *i*: $m_i = \lambda_2\left(\frac{1}{r_i^2} + \frac{1}{l_i^2} + \frac{1}{t_i^2} + \frac{1}{b_i^2}\right)$, where r_i, l_i, t_i and b_i are the straight-line distances between the vertex and the right, left, top and bottom borders of the display area. Again, λ_2 is a weighting factor.
3. To avoid overly long edges, the following term is added for each edge *k* of length d_k^2, with weighting factor λ_3: $c_k = \lambda_3 d_k^2$.
4. To reduce the number of edge crossings λ_4 is added to the cost function for pair of edges that cross.

5. To prevent edges from being drawn too close together, particularly those that intersect at a vertex, the following term is added for every vertex *i*, edge *k* with distance g_{ik} and weighting factor λ_5: $h_{ik} = \frac{\lambda_5}{g_{ik}^2}$

The relative importance of each of the criteria can be adjusted by varying the weighting factors. The fifth factor is normally used only during an optional *fine tuning* stage of the algorithm. The main part of the algorithm is run with only the first four factors until a termination condition is reached. Then the drawing may be adjusted further by using the larger cost function.

The actual value of the *initial temperature* depends on how many iterations are desired in the annealing process. At each temperature level, approximately 30*n* perturbations should be performed. The cooling function should be geometric, i.e. $T_{p+1} = \gamma T_p$, with $0.6 \le \gamma \le 0.95$. A value of 0.75 gives a relatively rapid cooling with the resulting graphs giving good aesthetic properties.

The *termination condition* can be a fixed number of iterations, say 10, or when the proposed embedding stops changing significantly over two to three iterations.

The simulated annealing algorithm runs in $O(n^2m)$ time. The number of temperature changes required to terminate the outer loop is constant in the sense that it does not depend on the input graph. O(n) perturbations are performed in the inner loop, say 30*n*. It takes O(nm) time to update the cost function, which is more efficient that calculating it from scratch. Since the location of only one vertex is changed in a new configuration, n vertices need to be updated and for each of these at most m edges will also need to be updated.

This technique produces drawings that are comparable to those generated by the spring method. This algorithm does not produce conventional looking graphs for a several categories of graphs. One such group is a large cycle with no chords. While this is

normally drawn as a large circle, this algorithm tends to draw the cycle curled around itself so there is not a large empty space in the middle. Simulated annealing is also not very good at bringing out the regularity of a graph such as a clique because it tends to draw all edges with comparable length. Although this has not yet been rigorously shown, this algorithm appears to be more successful with more complex graphs than spring algorithms. One useful property of simulated annealing algorithms is that they are easily parallelized.

Sugiyama Algorithm

Layered graphs are of practical interest in software engineering, PERT chart generating and other fields that require data modeling. The structure of a software program can be represented as a procedure calling hierarchy. Sometimes this structure can be drawn as a tree but more often it needs to be drawn as a layered graph. Müller, et al. (1993), call this an idealized layered software architecture a (k, 2)-partite graph. This notation means that there are k layers and each layer has edges only to the layers adjacent to it, i.e. above and below. The most popular algorithm for laying out this type of graph was introduced by Sugiyama, Tagawa, and Toda (1981) and is nicknamed after the first author. Some of its popularity may be attributed to its early development.

Although the overall structure of this algorithm is less complex than the first two presented, there are many small manipulations that need to be performed. Each of these sub-parts can be as complex as either the spring method or the simulated annealing method. Consequently, only a high level description of the Sugiyama algorithm will be presented here.

The algorithm takes an hierarchical directed graph as input and draws it in four stages. In stage I, the graph is transformed into a “proper” hierarchy, if necessary. In stage II, the vertices at each level are ordered to reduce the number of edge crossings. In stage III, the horizontal position of each vertex is manipulated to reduce the length of the edges.

Finally, in stage IV the graph is drawn. Stage I is a preparatory step and stages II and III form the main part of the algorithm.

A “proper” hierarchical graph has no cycles, and only has edges between adjacent levels. In stage I Sugiyama, Tagawa and Toda suggest condensing vertices as a way of eliminating cycles and refers the reader to other sources for more detailed information. Edges that span more than one level are replaced with placeholder vertices and edges that connect two levels. This hand-waving is interesting, as finding the minimum feedback edge set is an NP-complete problem.

In stage II, an iterated barycentric method is used to reduce the number of edge crossings. The *barycentre* of a graph is a vertex with the minimum distance between itself and every other vertex in the graph. A graph may have more than one barycentre. Conceptually, these vertices are those closest to the middle of the graph.(West, 1996) The barycentric method minimizes the distance between a vertex and its neighbours and as a result reduces the number of edge crossings in the drawing. In this stage, and the next, only one level and an adjacent level are considered at a time, thus simplifying a *k*-level problem to a 2-level problem, or a bipartite graph. The edges in between the two levels are represented as an adjacency matrix, with the upper level as rows and the lower level as columns. An adjacency matrix is constructed for each pair of adjacent levels. The barycentre for a the upper level is given as B_{ik}^{R}, where *R* denotes row and *i* denotes the current adjacency matrix, i.e. pair of levels, and *k* the current row in the matrix. Similarly, B_{il}^{C} is the barycentre for lower level, where *C* denote column and *I* denotes the current adjacency matrix and *l* the current column in the matrix.

$$B_{ik}^{R} = \sum_{l=1}^{q} l \cdot m_{kl}^{i} \Big/ \sum_{l=1}^{q} m_{kl}^{i}, \quad k = 1,\ldots,\left|V_i\right|, \text{ the number of vertices in the upper level}$$

$$B_{il}^{C} = \sum_{k=1}^{p} k \cdot m_{kl}^{i} \Big/ \sum_{k=1}^{p} m_{kl}^{i}, \quad l = 1,\ldots,\left|V_{i+1}\right|, \text{ the number of vertices in the lower level}$$

The algorithm first calculates the barycentres for each column in the matrix for the top two levels. It then re-orders the columns from smallest to largest barycentre. This change

is then reflected in the next adjacency matrix. This process is repeated for all adjacency matrices. After the re-ordering the rows of the adjacency matrix between the bottom two levels, then the barycentres for the rows are calculated and re-ordered. This change is reflected in the columns of the next higher matrix. This process is repeated until the first matrix has been re-ordered.

In stage III, edge lengths are reduced by adjusting the horizontal positions of vertices within a level using a *priority layout* method. Each vertex in a level is assigned a monotonically increasing priority number, say without loss of generality from left to right. In this way, the vertex order established in stage II. When reducing the spacing of a given vertex, only vertices with a lower priority number on the same level may be modified. Finally, no two vertices may lie on top of one another. Levels are adjusted in multiple passes from top to bottom and back to the top. Specifically, the order of levels adjusted is: 2, ..., k, k-1, ..., 1, t, ..., n, where t is an integer, $2 \leq t \leq n-1$. Often, three passes are sufficient to make a nice looking drawing. In the last stage, the graph is displayed on a graphic terminal.

Stage II was tested on one hundred randomly generated graphs. The minimum number of edge crossings was found through combinatorial search. The number of edge crossing in the final embedding was compared with the minimum possible and was found to be within 5%. It takes O(n) time to generate all the adjacency matrices. For each vertex in the graph, its barycentre must be calculated twice. Each calculation takes O(n) time, so the total amount of time spent calculating barycentres is $O(n^2)$. In the worst case, each row or column re-ordering requires the entire adjacency matrix to modified. This can be done in O(n) time, with a constant coefficient depending in the size of the adjacent levels. So stage II runs in $O(n^2)$ time. One iteration of stage III, in the worst case (i.e. every vertex requires all other vertices with smaller priority on the same level to move), will take $O(n^2)$ time. In total, this stage will take $O(Tn^2)$ time where T is the number of iterations of the up-down process. So while, the algorithm is rather complex it runs relatively efficiently. It takes $O(n^2)$ with a large coefficient and large lower order terms.

Conclusion

Since many of the sub-problems in graph drawing are NP-hard or NP-complete, it is too computationally expensive to attack the problem directly. As we have seen, innovative models and heuristics have been developed to the solve the problem. The spring method represent a graph embedding as a physical systems with energy states. By minimizing energy state, an aesthetically pleasing drawing can be generated. Similarly, the simulated annealing algorithm is patterned after a physical process that improves the embedding by applying many small perturbations to intermediate drawings. The Sugiyama algorithm uses a statistical method to calculate metrics to represent desirable aesthetic properties. These metrics are then used to manipulate the original graph to approximate the quality of those generated by exhaustive combinatorial search. The particular algorithm used by a computer scientist depends much on the problem domain for which it is going to be used. The study of automatic graph drawing algorithms is a growing field. As more fields in computer science start to use graphs as a means of problem solving or data representation, they will bring to the study their own particular requirements and insights. So far automatic graph drawing algorithms were of interest only in relation to some application. But as the field grows, so too will interest in the development of these algorithms for their own sake.

References

F.J. Brandenberg (editor). *Graph Drawing '95, Proceedings of Symposium on Graph Drawing*, Springer Verlag, Berlin, 1996.

I.F. Cruz and R. Tamassia. How to visualize a graph: Specification and Algorithms. Available on the WWW at URL `http://www.cs.brown.edu/people/rt/gd-tutorial.html`.

R. Davidson and D. Harel. Drawing Graphs Nicely Using Simulated Annealing. *ACM Transactions on Graphics*, 15(4):301-331, October, 1996.

G. Di Battista, P. Eades, R. Tamassia, And I. G. Tollis. Algorithms for Drawing Graphs: an Annotated Bibliography. June, 1994. Available by anonymous ftp from `ftp.cs.brown.edu:pub/papers/compgeo/`.

G. Di Battista, H. D Fraysseix, P. Eades, P Rosenstiehl, and R. Tamassia (Eds). *Graph Drawing '93, Proceedings of the ALCOM International Workshop on Graph Drawing and Topological Graph Algorithms*. Available by anonymous ftp from `wilma.cs.brown.edu:pub/papers/compgeo/`.

P. Eades. A Heuristic for Graph Drawing. *Congressus Numerantium*, 42:149-160, 1984.

P. Eades and K. Sugiyama. How to Draw a Directed Graph. *Journal of Information Processing*, 13(4):424-437, 1990.

M.R. Garey and D.S. Johnson. Crossing Number is NP-Complete. *SIAM Journal of Algebraic and Discrete Methods*, 4(3):312-316, 1983.

E. Harley and A.J Bonner. A Flexible Approach to Genome Map Assembly. *Proceedings of the International Symposium on Intelligent Systems for Molecular Biology*, pp. 161-169, 1994.

T. Kamada. Visualizing Abstract Objects and Relations: A Constraint-Based Approach, World Scientific, New Jersey, 1989.

M.R. Kramer and J. van Leeuwen. The Complexity of Wire-Routing and Finding Minimum Area Layouts for Arbitrary VLSI Circuits in "Advances in Computing Research", vol. 2, edited by F.P. Preparata, pp. 129-146, JAI Press, Greenwich, Connecticut, 1984.

Z. Miller and J.B. Orlin. NP-Completeness for Minimizing Maximum Edge Length in Grid Embeddings. *Journal of Algorithms*, 6:10-16, 1985.

H.A. Müller, M.A. Orgun, S.R. Tilley and J.S. Uhl. A Reverse-engineering approach to subsystem structure identification. *Software Maintenance: Research and Practice*, 5:181-204, 1993.

E.G. Noik. Dynamic Fisheye Views: Combining Dynamic Queries and Mapping with Database Views. Doctoral thesis, Department of Computer Science, University of Toronto, 1996.

K. Sugiyama, S. Tagawa, and M. Toda. Methods for Visual Understanding of Hierarchical System Structures. *IEEE Transactions on Systems, Man and Cybernetics*, 11(2):109-125, 1981.

R. Tamassia and I.G. Tollis (editors). *Graph Drawing '94, DIMACS International Workshop*, Springer Verlag, Berlin, 1995.

W.T. Tutte. Convex Representations of Graphs. *Proceedings of the London Mathematics Society*, 10:304-320, 1960.

W.T. Tutte. How to Draw an Graph. *Proceedings of the London Mathematics Society*, 13:743-768, 1963.

K. Wagner. Bermerkungen zum Vierfarbenproblem. *Jber. Deutsches Mathematiks Verein*, 46:26-32, 1936.

D.B. West. Introduction to Graph Theory. Prentice Hall, Upper Saddle River, New Jersey, 1996.