Graph Parsing Techniques for Visualizing Directed Graphs

C. L. McCreary and Fwu-Shan Shieh
Department of Computer Science and Engineering
Auburn University

1. Background

The goal of the proposed work is to develop techniques for visualizing directed graphs. Our preliminary work uses a two step process that first analyzes the graph through parsing it and then applies attributes to the parse tree to determine the node and edge layout. Our experimentation has convinced us that the technique is superior to others currently in practice.

Many researchers have studied this problem and many graph drawing systems have been developed (see 10. for a complete list). The aesthetic criteria of the systems vary. The objectives may include requirements of uniform edge length, minimum number of edge crossings, straight edges, grid drawings (edges are either horizontal or vertical), minimal bends in the edges, minimum area covered, and display of symmetries. Some limit the input to a graphs with particular propeties, (planar graphs, trees, and graphs with maximum degree of four), or to graphs taht are intended for specific applications (Petri nets, network representations, digital system schematic diagrams, PERT diagrams or flowcharts). We are developing CG (**C**lan-based **G**raph Drawing Tool) to convert a textual description of an arbitrary directed graph to a visual representation. The objective of the system is to provide an aesthetically pleasing visual layout for arbitrary directed graphs. CG uses a unique graph parsing method to determine intrinsic substructures (clans) in the graph and to produce a parse tree. The tree is given attributes that specify the node layout. Through the use of clans as substructures, there are relatively few unnecessary edge crossings. Routing nodes are added that guarantee few bends in long edges, and long edges are shortened whenever possible.

CG is the first graph drawing tool to use a graph parsing as the fundamental structure that describes the node layout. Graph-grammar researchers have described schemes for graph layout that are similar to ours, but difficulties in graph-grammar parsing have inhibited their implementation. In particular, Brandenburg [4.] defines a layout graph grammar as a graph grammar together with a layout specification. The layout specification associates a finite set of layout constraints with each production. Our approach is to classify the productions of the graph parse and associate layout attributes with each production type in the parse tree of the graph.

The use of graph parsing distinguishes CG from all other general directed graph drawing schemes. The standard approachto automatic layout of directed graphs has three steps: 1. Assign nodes to layers; 2. reorder the nodes in each layer to reduce crossings; 3. Adjust the position of the nodes in each layer to reduce the number of bends. Usually the nodes are layered by distance from a source node. Our method also creates a *hierarchical* drawing where nodes and bends are constrained to lie on a set of equally spaces horizontal lines called *layers* [8., 14.]. CG groups subgraphs (clans) which have two-dimensional affinity in contrast to the one-dimensional constraints of the standard method. CG also reduces bends by placing routing nodes in their correct positions in the parse tree.

CG is unique in several ways:

- The node layout is balanced both vertically and horizontally.
- Nodes within a clan are placed close to each other in the drawing.
- Nodes are grouped according to two-dimensional affinity.

Graph Parsing

The node layout is determined by the combination of (1) parsing of the graph into logically cohesive subgraphs and (2) defining layout attributes to apply to the resulting parse tree. The parse is based on simple graph productions, and the attributes produce a layout that can be specified by a particular application. The foundation for the parse comes from the theory of 2-structures[6.,7.].

Clan-based graph decomposition (CGD) is a parse of a directed acyclic graph (DAG) into a hierarchy of subgraphs. The subgraphs defined by our graph grammar and parse are called *clans*. Let G be a DAG. A subset $X \subseteq G$ is a *clan* iff for all $x, y \in X$ and all $z \in G - X$, (a) z is an ancestor of x iff z is an ancestor of y, and (b) z is a descendant of x iff z is a descendant of y. An alternate description of a clan depicts it as a subset of nodes where every element not in the subset is related in the same way (i.e. ancestor, descendant or neither) to each member in the subset. Trivial clans include singleton sets and the entire graph.

A *simple clan C* with nodes $n_1 \ldots n_k$ is classified as beloning to one of three types. It is (i) *independent* if it is a union of disconnected nodes; or (ii) *linear* if for every pair of nodes n_i, n_j in *C*, n_i is an ancestor of n_j or n_j is an ancestor of n_i. *(iii) primitive* if neither (i) nor (ii) applies. Simple independent clans are sets of isolated nodes. Simple linear clans are sequences of one or more nodes $v_i, v_{i+1}, \ldots, v_{j-1}, v_j$ where for $i < k$, v_i is an ancestor of v_k. Any DAG can be constructed from these simple clans through a series of productions. Any DAG can be parsed into a tree of clans whose leaves are trivial clans (graph nodes) and whose internal tree nodes are complex clans built from their descendants. The parse tree is unique when maximal linear and maximal independent clans are identified at each step in the parse.

The concept of a *quotient graph* is important because it allows the classification of clans as linear, primitive or independent to be applied to complex clans. Let C be a clan and $\{C_1,,\ldots C_k\}$ a partition of C, where each C_i is a maximal proper subclan of C. The *quotient graph* of C, denoted $C/C_1 \ldots C_k$, is the graph with nodes $C_1, \ldots, C_k$. There is an edge from C_i to C_j in the quotient graph precisely when there is an edge (x,y) of C with $x \in C_i$ and $y \in C_j$. Every clan can be identified as linear, primitive or independent according to the classification of its quotient graph. For every transitively reduced digraph there is a unique decomposition into quotient graphs that are identified as linear, primitive or independent [12.]. The quotient graphs form a hierarchy we call the *parse tree.*

Spatial Analysis for Node Layout

The parse tree of the graph is used to provide the graph with geometric interpretations that define the visual layout. A *bounding box* with known dimensions is associated with each clan, and the nodes in the clan are assigned locations within the bounding box. Synthetic attributes are associated with the parse tree hierarchy to show the embedding and dimensions of the bounding boxes and inherited attributes compute the locations of nodes and bounding boxes. The attributes can vary with the application to present a customized visual representation of the graph. For illustrative purposes we choose simple attributes and semantic rules which we call *natural.* The natural attributes give a balanced layout.

The bounding box associated with each node (clan) of the parse tree specifies the allotted area for the subgraph. Semantic rules compute the dimensions of the bounding boxes at a tree node from the dimensions of the bounding box for its children. The intrinsic attributes of singleton DAG nodes (or parse tree leaves) describe initial bounding boxes.

For the **natural** attributes, the bounding boxes are rectangles and the attributes are the length and width of the rectangles. The intrinsic attributes of the parse tree leaves assign the bounding boxs to be unit squares. A linear clan is bounded by a rectangle whose length is the sum of the lengths of the component clans and whose width is the maximum width of the component clans. An independent clan is placed in an area whose width is the sum of the widths of the component clans and whose length is the maximum of the lengths of the component clans.

Definition: Denote the *natural bounding box attribute* of node N in parse tree T by (N.l,N.w). We define the *natural* values of the attribute to be:

(1) (N.l, N.w) = (1,1), if N has no children;

(2) (N.l, N.w) <-- (C_1.l+...+C_k.l, Max(C_1.w,...,C_k.w)), if N is a **linear** node with children C_1...C_k;

(3) (N.l, N.w) <--- (Max(C_1.l,...,C_k.l), C_1.w+...+C_k.w), if N is an **independent** node with children C_1 ...C_k.

Bounding Box Placement

After the spatial requirements for each bounding box have been computed it must be mapped onto coordinates in the viewing window. The inherited attributes place a node's bounding box within the bounding box of the parent and insure that its space does not overlap with that of its siblings.

For the **natural** attributes, a node's bounding box is described by its horizontal (x-value) and vertical (y-value) locations. In order to obtain a balanced layout, child bounding boxes are centered within the bounding box of the parent. The x and y values denote the upper left corner of the bounding box. If the upper left corner of the entire window is denoted (0,0), the location (x,y) represents x units to the right and y units below (0,0). The natural location attribute of the root will anchor its upper left corner at (0,0). A tree node N that is a child of a linear node will have an x-coordinate that centers it within the width of the parent's bounding box and a y-coordinate will place N directly below its left sibling, when it has one and otherwise at the y-level of its parent. Similarly, a tree node N that is a child of an independent node will have a y-coordinate that centers it within the length of the parent's bounding box and an x-coordinate that will place N directly to the right of its left sibling.

Definition: Denote the *natural location attribute* of node N in the parse tree T by (N.x, N.y). We define the *natural* values of the attribute to be:

(1) (N.x, N.y) = (0, 0) if N is the root of T.

(2) If the parent P of N is a **linear** node

(a) if N has left sibling F, (N.x, N.y) = (P.x + (P.w - N.w)/2, F.y+F.l).

(b) if N has no left sibling, (N.x, N.y) = (P.x + (P.w - N.w)/2, P.y)

(3) If the parent P of N is an **independent** node

(a) if N has left sibling F, (N.x, N.y) = (F.x + F.w, P.y + (P.l - N.l)/2).

(b) if N has no left sibling, (N.x, N.y) = (P.x, P.y + (P.l - N.l)/2).

For child C of linear node N, the actual rectangle in which the node is to be centered has length C.l and width N.w. For child D of independent node I, the rectangle in which D is to be centered has length I.l and width D.w. Figure 1 illustrates the process: (a) shows a graph with the maximal clans identified; (b) is the parse tree for this graph; (c) labels the tree nodes with the bounding box attributes; (d) shows the node layout.

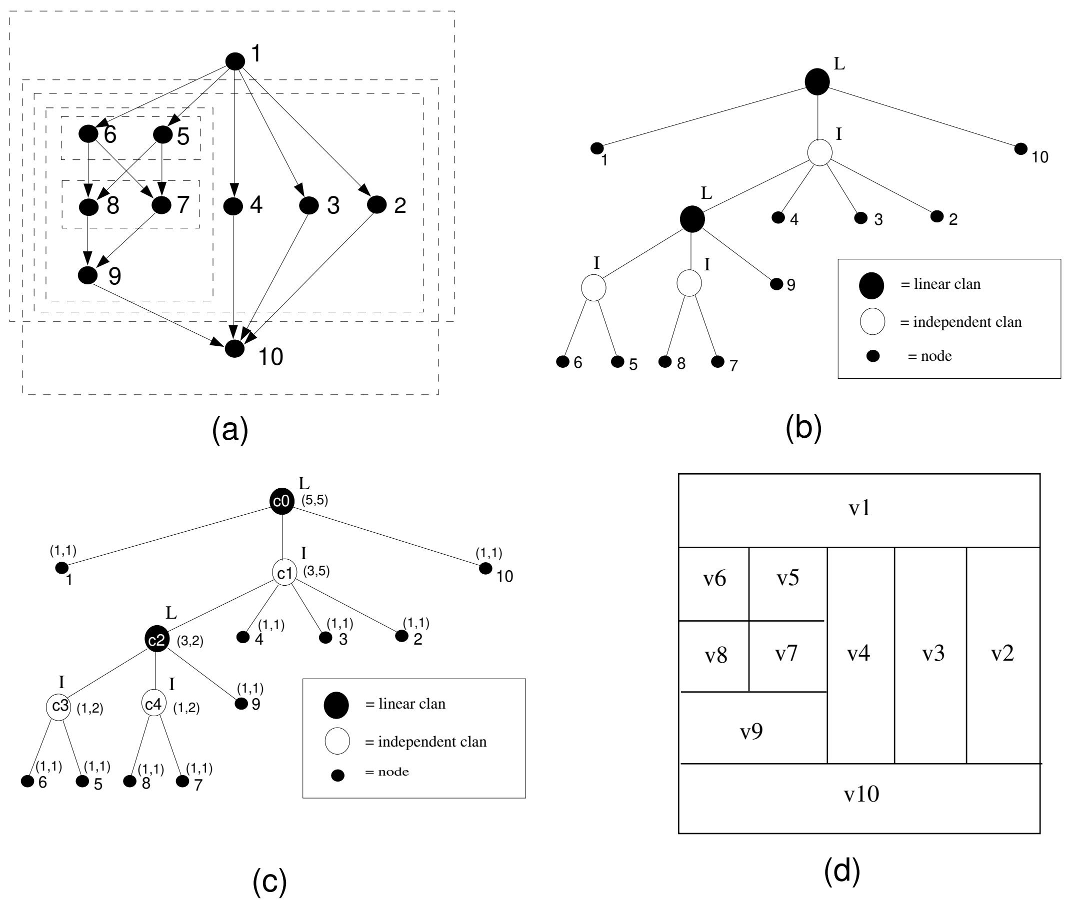


Figure 1. Node Layout

Drawing Edges

Using the parse tree to place nodes is simple and elegant, and it provides for a balanced node placement. However, the node placement must be modified if edges are to be displayed. If adjacent nodes are connected by straight edges, several unacceptable visualizations may occur when the nodes are placed according to the location attributes.

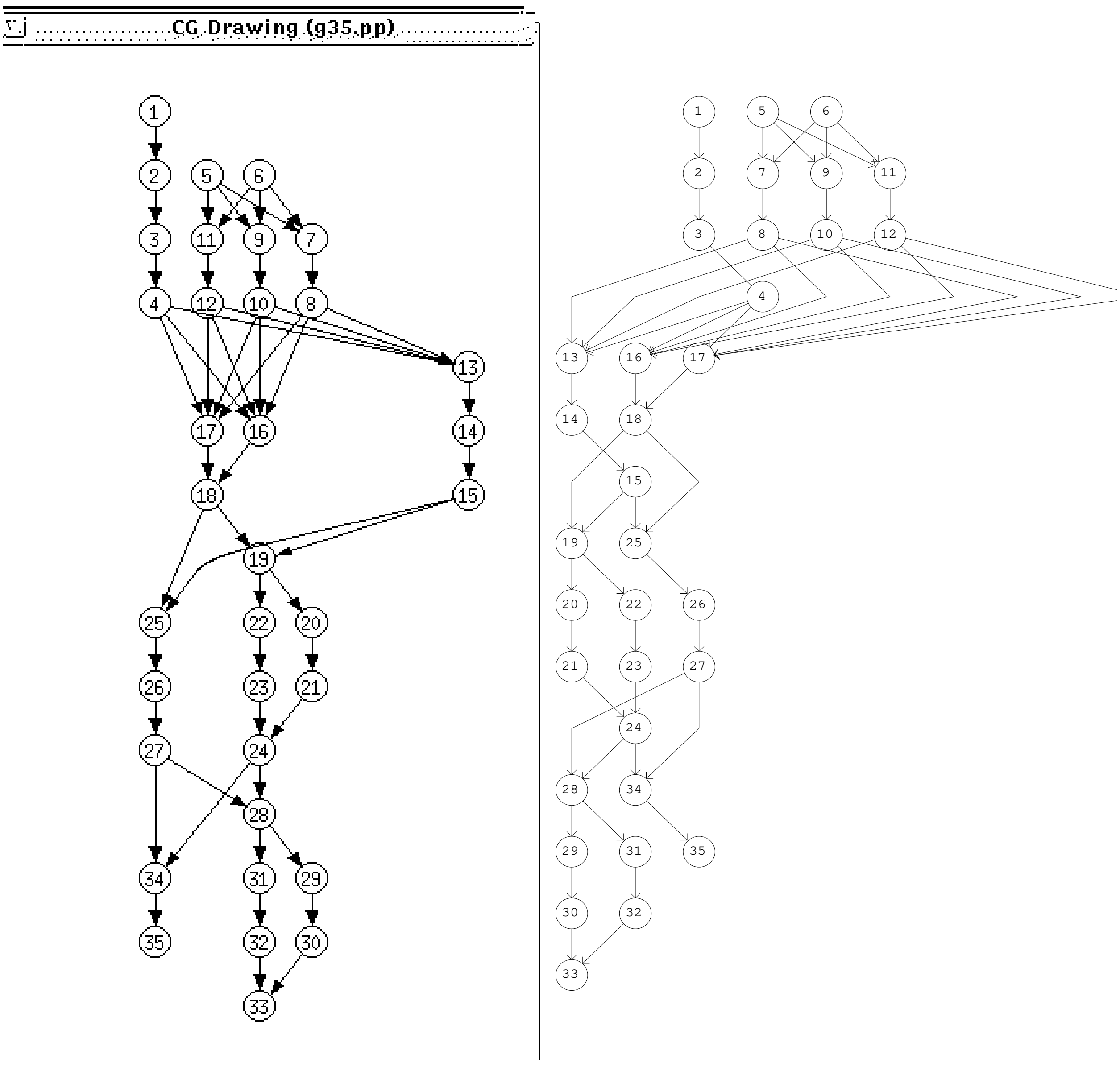
- edges could pass through nodes on their path
- in routing edges around nodes, there may be unnecessary bends
- edges might be superimposed upon other edges
- unnecessarily long edges may be drawn
- there may be an unnecessary number of edge crossings.

The first 4 problems listed above are caused by "long" edges, i.e. edges connecting nodes whose levels (y-values) differ by more than one. The traditional solution is to place dummy nodes at each intermediate level and route the long edge through the intermediate nodes [26.]. One of the problems with this approach is that the long edges, by passing through nodes placed at arbitrary horizontal displacements, may contain unnecessary bends. In a way similar to adding dummy nodes at each level, CG solves the problem by adding dummy nodes to the parse tree and reroutes the edges through the dummy nodes. However, CG's solution is superior because, since dummy nodes are added to clans rather than levels, fewer dummy nodes are added to the graph. Furthermore, they are added to clans in a way that reduces the horizontal displacement from the true graph nodes, thus reducing edge bends.

We have developed several heuristics to resolve problems arising from long edges [12.]. Places where edge length can be reduced are identified and remedied; initial node placements guarantee a limited number of bends in the edges, and if possible, bends are eliminated completely. The grouping of nodes into clans automatically reduces the number of edge crossings. Figure 2 contrasts the output currently produced by our program with GraphEd's interpretation of Sugyama's algorithm for the same graph. Note that CG's drawing has fewer edge bends (1 versus 13); CG groups nodes 13, 14, and 15 far more clrealy; CG lacks the bunching seen at the top in the GraphEd version; and the more balanced appearance of the drawing produced by CG. Furthermore, despite the fact that CG lacks explicit procedures to reduce edge crossings, the drawing produced by CG has fewer edge crossings than that produced by GraphEd.

2.Objectives and Expected Significance

The strength of the graph parsing technique for drawing digraphs as compared to other approaches comes from two sources: the graph analysis which gives a two-dimensional description of the graph and the expressive power of attributes. In drawing digraphs, the accepted standard constraint requires nodes to be placed relative to each other in a single dimension. All edges "flow" in the same direction, e. g. from left to right, or from top to bottom, or from bottom to top. For the purpose of discussion, we'll assume the layout specification is from top to bottom. Because of this constraint, automatic layout designers have implemented the hierarchical approach where nodes are classified and laid out by level. Without adding heuristics to move nodes down, nodes tend to bunch toward the top in one-dimensional systems. Also, since there are no guidelines for horizontal placement, unnecessary edge crossings are frequent and heuristics are needed to prevent nodes from congregating on the left. The graph parsing technique provides a way for both vertical and horizontal placements to be considered. Centering nodes in bounding boxes prevents bunching toward the top and clustering on the left.



Drawing by CG GraphEd

Figure 2. Random Graph of 35 Nodes

Assigning intrinsic attributes to the parse tree nodes and defining procedures for computing inherited and synthesized attributes has proved to be a powerful way to specify user-defined layout constraints. The technique has been applied to trees and the diversity of specifying the layout is illustrated in Figure 3. Carpano's specification [3.] requires nodes at the same level to be on a circle centered at the tree root. Ullman's h-tree [15.] is used for VLSI layout and requires edges to be placed horizontally or vertically. One of the goals of our work is to study layouts in various domains and develop attributes and procedures for providing automatic layout that conforms with the standards in the domain. By expanding the graph class to general digraphs and discovering various attribution schemes, we will have a unified system that can be useful to many diverse disciplines.

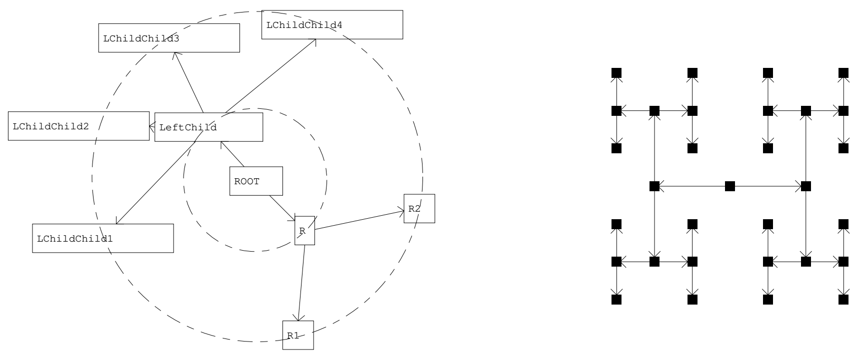


Figure 3. Tree Layouts Described by Attribute Graph Grammars

<u>BIBLIOGRAPHY</u>

1. Borgatti, S. P., M. G. Everett, and L. C. Freeman, UCINET IV. Analytic Technologies, Columbia, SC., 1992.

2. A. Borning, “The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353-387, October 1981.

3. M. Brain, <u>MOTIF Programming</u>, Digital Press, 1992.

4. F. J. Brandenburg, “Layout Graph Grammars: the Placement Approach,” <u>Lecture Notes in Computer Science</u>, <u>532, Graph Grammars and Their Application to Computer Science</u>, Springer-Verlag, Berlin, 1991.

5. M. J. Carpano, “Automatic Display of Hierarchies Graphs for computer Aided Decision Analysis”, *IEEE Trans. on Systems Man and Cybernetics,* vol. SMC-10, no. 11, pp 707-715, Nov. 1980.

6. J. R. Cordy, T. C. N. Graham, “GVL: a Graphical Functional Language for the Specification of Output in Programming Languages,” *1990 International Conference on Computer Languages,* New Orleans, March 1990, pp.11-22.

7. J. H. Cross, “Grasp/Ada: Reverse Engineering Control Structure Diagrams, “ *Proceedings of Working Conference on Reverse Engineering,* May 21-23, 1993, Baltimore, MD. 107-116.

8. I.F. Cruz, “Using a Visual Constraint Language for Data Display Specification” ,*Workshop om Prinicples and Practice of Constraint Programming,* Newport, RI, April, 1993.

9. I. F. Cruz and R. Tamassia, “How to Visualize a Graph: Specification and Algorithms,” Tutorial on Graph Drawing, available at http://www.cs.brown.edu/calendar/gd94, 1994.

10. G. Di Battista, P. Eades, R. Tamassia, I. Tollis, “Algorithms for Drawing Graphs: an Annotated Bibliography”, available via ftp from ftp.cs.brown.edu (128.148.33.66) in file /pub/papers/compgeo or WWW site:.http://www.cs.brown.edu/people/rt/

11. A. Ehrenfeucht and G. Rozenberg, “Theory of 2-Structures, Part I: Clans, Basic Subclasses, and Morphisms,” *Theoretical Computer Science*, vol. 70, pp. 277-303, 1990.

12. A. Ehrenfeucht and G. Rozenberg, “Theory of 2-Structures, Part II: Representation Through Labeled Tree Families,” *Theoretical Computer Science*, vol. 70, pp. 305-342, 1990.

13. E. R. Gansner, S. C. North, and K. P. Vo, “DAG - A Program that Draws Directed Graphs,” *Software - Practice and Experience,* vol 18, no. 11, pp. 1047-1062, Nov. 1988.

14. E. R. Gansner, E. Koutsofios, S. C. North, and K. P. Vo, “A Technique for Drawing Directed Graphs,” *IEEE Transactions on Software Engineering*, vol. 19, no. 3, pp. 214-

230, 1993.

15. M. Himsolt, "GraphEd: An Interactive Graph Editor," *Proc. STACS 89*, Lecture Notes in Computer Science, vol. 349, pp. 532-533, Springer-Verlag, 1989.

16. D. Hix & H.R. Hartson, Developing User Interfaces, Wiley, New York, 1993.

17. T. Kamada, "Visualizing Abstract Objects and Relations - A Constraint-Based Approach". *World Scientific*, Singapore, 1989.

18. Kaugars, K., Brazma, A; "CATGraph: Visualizing Large Labeled Graphs"; *Technical Report NMSU-TR-92-CS-08*; Department of Computer Science, New Mexico State University, 1992

19. Kaugars, K., Reinfelds, J., Brazma, A; "A Simple Algorithm for Drawing Large Graphs on Small Screens"; *Proceedings of GD '94*, p278-281

20. E. Koutsofios and S. North, "Drawing Graphs with *dot*", *Dot User's Manual,* AT&T Bell Labs, Murray Hill, N. J., Oct. 18, 1993.

21. J. Marks, " A Formal Specification for Network Diagrams that Facilitates Automated Design", *Journal of Visual Langues and Computing,* 2:395-414, 1991.

22. McCreary, C.L. and Roth, S.S., "Incremental Graph Parsing", *Proceedings of the 32nd Annual Southeast ACM Conference*, March, 1994, pp. 81-86.

23. C. McCreary, F.-S. Shieh and H. Gill, *CG: a Graph Drawing System Using Graph-Grammar Parsing,* Lecture Notes in Computer Science, Vol. 894, 270-273, Springer-Verlag, 1995

24. M. Sarkar and M.H. Brown, "Graphical Fish-eye Views of Graphs"; *Proceedings of CHI '91,* p83-91.

25. Seidman, S. B."Relational models for social systems", Environment and Planning B: Planning and Design 14: 135-148., 1987

26. K. Sugiyama, S. Tagawa, and M. Toda, "Methods for Visual Understanding of Hierarchical System Structures," *IEEE Trans. on Systems., Man, and Cybernetics*, vol. 11, no. 2, pp. 109-125, Feb. 1981.

27. J. D. Ullman. *Computational Aspects of VLSI,* pp 80-130, Computer Science Press, 1984.

28. C. J. Van Wyk. "A High-Level Language for Specifying Pictures", *ACM Transactions on Graphics,* 1(2):163-182, April 1982.