Using Graph Parsing for Automatic Graph Drawing

C.L. McCreary and Fwu-Shan Shieh

Department of Computer Science and Engineering

Auburn University

Abstract: This paper presents a procedure for automatically drawing directed graphs. Our system, CG, uses a unique clan-based graph decomposition to determine intrinsic substructures (clans) in the graph and to produce a parse tree. The tree is given attributes that specify the node layout. CG then uses tree properties with the addition of “routing nodes” to route the edges. The objective of the system is to provide, automatically, an aesthetically pleasing visual layout for arbitrary directed graphs. The prototype has shown the strengths of this approach. The innovative strategy of clan-based graph decomposition is the first digraph drawing technique to analyze locality in the graph in two dimensions. The typical approach to drawing digraphs uses a single dimension, level, to arrange the nodes.

1. Introduction

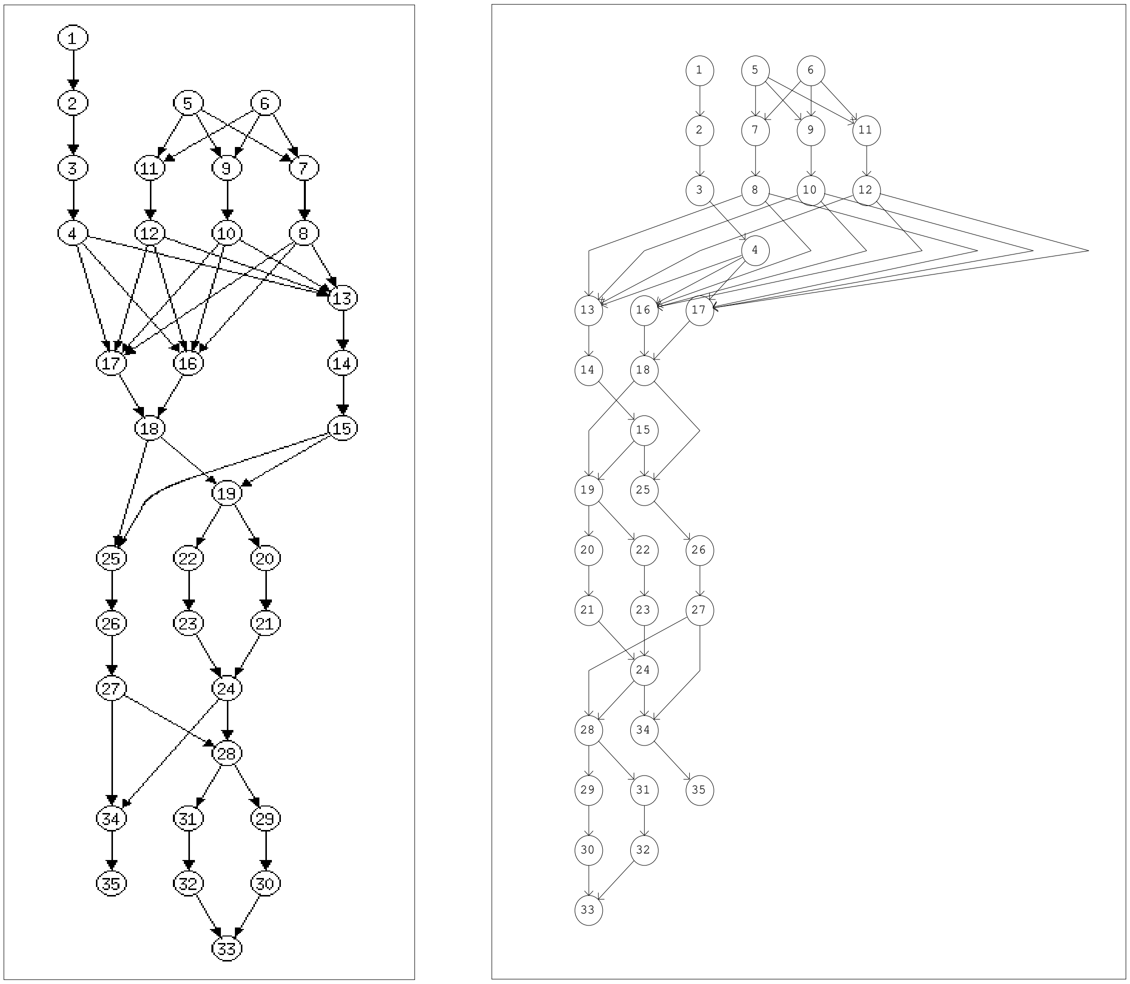
Directed graphs, or digraphs, are an excellent means of conveying the structure and operation of many types of systems. They are capable of representing not only the overall structure of such a system, but also the smallest details in a simple and effective way. However, drawing digraphs by hand can be tedious and time consuming, especially if the number of nodes and edges is large. In addition, much time can be spent just trying to plan how the graph should be organized on the page. We have developed an automated system capable of converting a textual description of a digraph into a well organized and readable drawing of the digraph.

Many researchers have studied this problem and many graph drawing systems have been developed [see 3 for 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 graphs to a particular class such as planar graphs, trees, graphs with maximum degree of four, or some application-specific graphs such as Petri nets, network representation, digital system schematic diagrams, PERT diagrams, flowcharts, etc. CG (Clan-based Graph Drawing Tool) is a tool we use for our work with program dependency graphs. Like *dot* [12.] and its predecessor DAG [10.], CG takes a textual description of an arbitrary directed graph (*digraph*) and produces a visual representation of it. 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. CG then uses tree properties with the addition of "routing nodes" to route the edges. The objective of the system is to provide an aesthetically pleasing visual layout for arbitrary directed graphs. 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.

The remainder of the paper is divided into sections that relate CG to other drawing systems and describe the five major steps in our graph drawing scheme: (1) parsing the graph into subgraphs of clans. (2) determining the dimensions of the bounding boxes that contain the clans; (3) adding dummy nodes to route long edges; (4) recomputing the bounding box dimensions for the augmented graph; and (5) assigning x and y-coordinates to all bounding boxes, including those that represent individual nodes. Section 2 compares our work with that of others. Section 3 gives an overview of the algorithm. Section 4 defines clans and the parse tree and briefly describes the parsing algorithm. Section 5 explains the spatial analysis of the parse tree by identifying the bounding box and node placement rules and attributes. Section 6 describes the issues in drawing edges and illustrates CG's solutions. Section 7 extends the parsing to general directed graphs. Section 8 illustrates CG's unique ability to abstract the graph by contracting clans into single nodes. The applications section (Section 9) features actual output from CG for a social networks application.

2. Related Work

The standard method for automatic layout of directed graphs, creates a *hierarchical* drawing where nodes and bend are constrained to lie on a set of equally spaced horizontal lines called *layers*. The standard approach has 3 steps: 1. Assign nodes to layers; 2. re-order 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. Figure 1 contrasts the output currently produced by CG with GraphEd's interpretation of Sugiyama's algorithm for the same graph[13.]. Note that CG's drawing has fewer edge bends (1 versus 13); CG groups nodes 13, 14, and 15 far more clearly; and the CG produces a more balanced appearance of the drawing, both horizontally and vertically. Furthermore, despite the fact that CG lacks explicit procedures to reduce edge crossings, the drawing produced by CG has nearly the same number of edge crossings as that produced by GraphEd (23 for CG, 21 for GraphEd.).



(a) result of CG

(b) result of GraphEd

Figure 1. Two graph drawings

CG is the first graph drawing tool to use graph parsing as the fundamental structure that describes the node layout for a general directed graph. 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 [1] 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. Assigning intrinsic attributes to the parse tree nodes and defining procedures for computing inherited and synthesized attributes has been applied to trees and the diversity of specifying the layout is illustrated in Figures 2-5. Four tree layout schemes by Reingold/ Tilford, Moen, Carpano, and Ullman have been described by attributed parse trees[20.]. In the first three algorithms, the node size is dependent on the size of the label. The Reingold/Tilford algorithm [Figure 2.] places nodes of the same level on a line at a given distance from the root. Meon's layout [Figure 3.] reduces the space required by placing children as close to their parents as possible. In the Carpano layout [Figure 4.], nodes at the same level are placed on concentric circles. The Ullman H-tree layout [Figure 5.] was designed for VLSI [18.] and has edges that are only vertical or horizontal and the layout minimizes area. While this paper will show only one example attribute algebra, the layout possibilities are limited only by one's imagination.

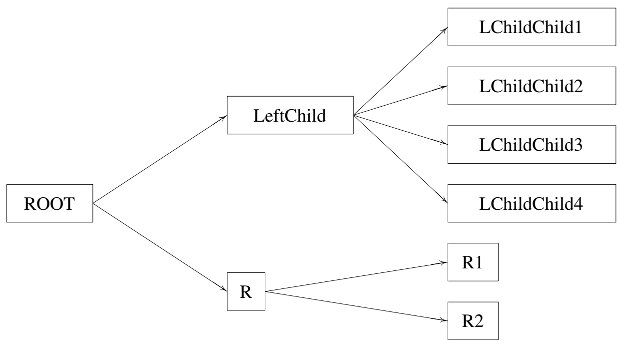


Figure 2. Reingold/Tilford

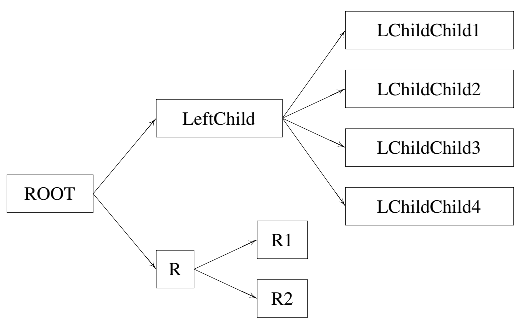


Figure 3. Moen

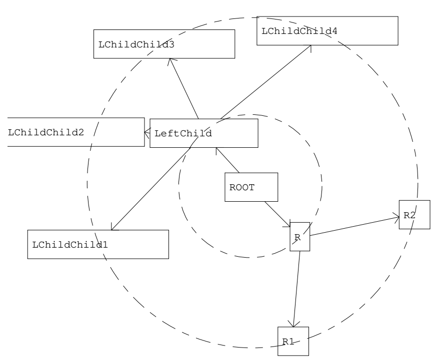


Figure 4. Carpano

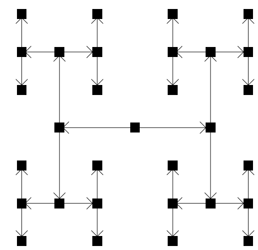


Figure 5. Ullman

CG extends the drawing of directed acyclic graphs to general directed graphs. Its parser is the first we know about that decomposes directed graphs into a tree of subgraphs (clans). CG defines an attribute grammar for the parse tree, and computes node layout through the attributes. Graph parsing in an improvement over the hierarchical approach because it discovers clans, structures which have two-dimensional affinity rather than layers which have only one-dimensional similarity. The use of graph parsing distinguishes CG from all other general directed graph drawing schemes.

CG's drawings are unique in several ways:

- The node layout is balanced both vertically and horizontally.
- Nodes within a clan, a subgraph of nodes that have a common relationship with the rest of the nodes in the graph, are placed close to each other in the drawing.
- Nodes are grouped according to a two-dimensional affinity rather than a single dimension such as level.
- The users can contract a clan into a single node and later expand the node to show the subgraph in its original clan.

3. Algorithm Overview

The algorithm consists of several non-trivial steps, each of which will be explained in greater detail in later sections of this paper. The input to the algorithm is a list of nodes and edges of a directed graph and the output is a drawing of the graph.

Step 1. Parse the graph, creating a parse tree whose leaves represent graph nodes and whose internal nodes represent subgraphs.

Step 2. For each node in the parse tree, compute the dimensions of a bounding box required to contain the subgraph represented by the parse tree node.

Step 3. Determine long edges in the graph and add dummy nodes through which the long edges should be routed. Place the dummy nodes in the parse tree.

Step 4. Recompute bounding box dimensions on the augmented tree.

Step 5. Using the bounding boxes, determine actual node locations.

Step 6. Draw the graph by placing the node labels in a circle and connecting the circles with straight lines when possible and splines at curves. Use dummy nodes to route the long edges.

4. 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. To simplify the presentation, sections 2 and 3 will consider the node layout of only series-parallel graphs. Section 4 will extend the ideas to general directed acyclic graphs (DAGs) and then to digraphs.

4.1 Basic concepts

Graph-theoretic definitions. This section reviews the standard graph concepts we shall use. A *directed graph (digraph)* $G = \langle V, E \rangle$ consists of a finite set of *nodes* V and a finite set of *edges* E. Each edge is an ordered pair (x, y) of distinct nodes; x is a predecessor of y, and y is a *successor* of x. Node z is a *source* node if z has no predecessors, and node z is a *sink* if it has no successors. $G' = \langle V', E' \rangle$ is a subgraph of G if $V' \subseteq V$ and $E' \subseteq E$. A directed acyclic graph (*DAG*) is *transitive* if for any two nodes x and y such that there is a path from x to y, either $x = y$ or (x,y) is an edge. An edge (x,y) is *transitive* if there is a path from x to y that avoids (x,y). The *transitive reduction (or minimal graph)* of graph $G = \langle V, E \rangle$ is the largest subgraph $G' = \langle V, E' \rangle$ of G that contains no transitive edges.

The class of *vertex series-parallel (VSP)* DAGs is defined in terms of the subclass of its transitively reduced or minimal graphs. The DAGs in this class are called *minimal vertex series-parallel (MVSP)*.

Definition: *Minimal vertex series-parallel DAGs* is defined recursively as follows:

(i). The DAG having a single vertex and no edges is MVSP.

(ii). If $G_1 = \langle V_1, E_1 \rangle$, $G_2 = \langle V_2, E_2 \rangle$ and $G_k = \langle V_k, E_k \rangle$ are k MVSP DAGs then so are the DAGs constructed by the following operations:

Parallel composition: $G_p = \langle V_1 \cup V_2 ... \cup V_k,\ E_1 \cup E_2 ... \cup E_k \rangle$.

Series Composition:

$G_s = \langle V_1 \cup V_2 ... \cup V_k, E_1 \cup E_2 ... \cup E_k \cup (T_1 \times S_2) \cup (T_2 \times S_3) ... \cup (T_{k-1} \times S_k) \rangle$, *where T_i is the set of sinks of G_i and S_i is the set of sources of G_i.*

Definition: A DAG is *Vertex series-parallel DAGs (VSP)* if and only if its transitive reduction is MVSP.

Definition: A *tree* is a connected DAG where there is exactly one vertex, the *root, r,* for which there is no edge (x,r) and for every other vertex v there is exactly one edge (x,v). Let $T = \langle V, E \rangle$ be a tree. If (v,w) is in E, then v is the *parent* of w and w is a *child* of v. If there is a path from v to w, then v is an *ancestor* of w and w is a *descendant* of v. A vertex with no descendants is called a *leaf.*

4.2 Graph Decomposition

Clan-based graph decomposition (CGD) is a parse of a DAG into a hierarchy of subgraphs we call *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. For MVSP DAGs, clans are the subgraphs in the parallel and series constructions. In the MVSP DAG of Figure 6., sets {5, 6}, {7, 8}, {5, 6, 7, 8, 9}, {2,3,4,5,6,7, 8, 9}, {1, 2,3,4,5,6,7, 8, 9} and {2,3,4,5,6,7, 8, 9, 10}are some of the nontrivial clans.

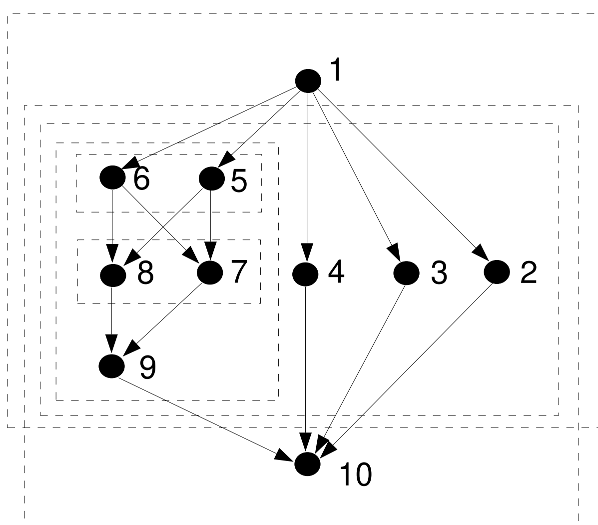


Figure 6. Clans

A *simple clan C* with nodes $n_1 \ldots n_k$ is classified as belonging to one of three types. It is (i) *parallel* if it is a union of disconnected nodes; or (ii) *series* 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 parallel clans are sets of isolated nodes. Simple series 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. For MVSP DAGs, parallel clans are the components of a parallel construction and series clans are the components of a series construction. MVSP DAGs have no primitive clans.

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 series and maximal parallel clans are identified at each step in the parse[7.]. For MVSP DAGs, the parse tree is constructed by associating a tree of one node with the MVSP DAG having one vertex and no edges, and building larger trees from smaller ones as the process of building the MVSP DAG by series and parallel composition progresses. A unique parse tree results when a series (parallel) composition is never followed by another series (parallel) composition. This composition will be called *canonical.*

For minimal DAGs, the identification of a *quotient graph* for composite clans allows their classification as series, primitive or parallel. 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 series, primitive or parallel 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 series, primitive or parallel [7.]. For MVSP DAGs, the series quotient graph G_s has nodes $\varsigma_1, \varsigma_2, \ldots \varsigma_k$ where ς_i corresponds to node set V_i and edge set $(\varsigma_i, \varsigma_{i+1})$ $1 \leq i \leq k-1$ and the parallel quotient graph G_p has nodes $\varsigma_1, \varsigma_2, \ldots \varsigma_k$ and no edges.The quotient graphs form a hierarchy we call the *parse tree.* A linear time algorithm to recognize VSP graphs is found in [19.].

In Figure 6. the clans that partition the entire graph are: $C_1 = \{1\}$, $C_2 = \{2,3,4,5,6,7,8,9\}$, $C_3 = \{10\}$, and the quotient graph is series. The clans that partition C_2 are {2}, {3}, {4}, and {5,6,7, 8,9}} and the quotient graph is parallel. Every clan can be identified as series, primitive or parallel according to the classification of its quotient graph. Figure 7. shows the parse tree of the graph of Figure 6.

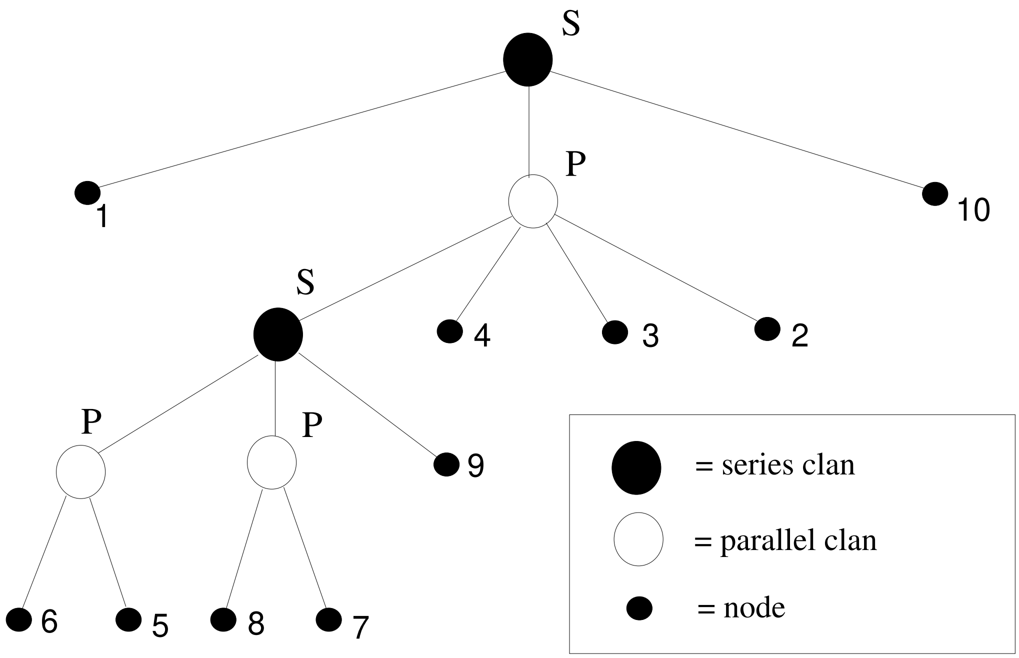


Figure 7. Parse Tree

Theorem 1: When the original graph is a minimal graph, the graph generated by CGS is also a minimal graph [11].

Theorem 2: Any minimal graph can be generated by a unique canonical sequence of productions from CGS [6].

Theorem 3: For every minimal graph there is a unique decomposition into quotient graphs that are identified as series, primitive or parallel [6.,7.].

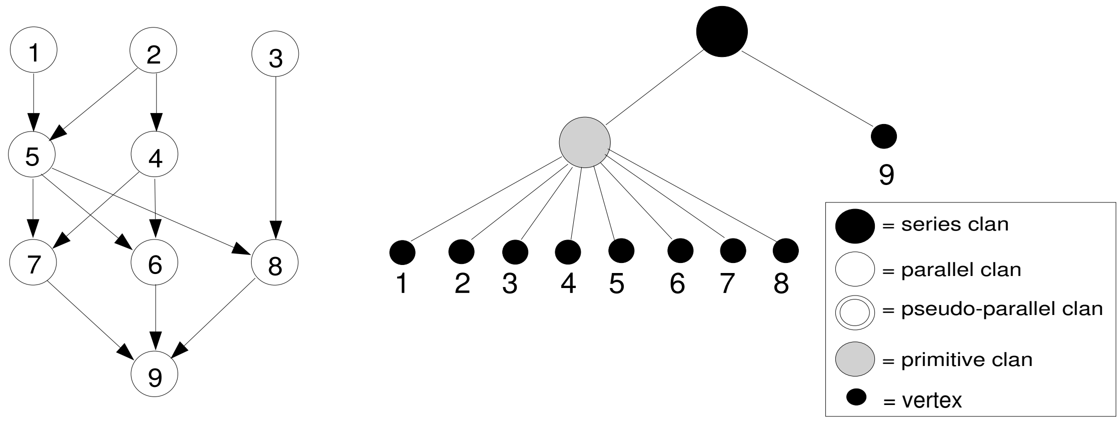
Theorem 3 guarantees the existence of a unique parse tree from the clans of a graph. An algorithm for parsing a minimal graph in this system is found in [15.]. The basic idea of the algorithm is to compute two partitions of the nodes: *S* (siblings) and *M* (mates). *S* contains sets S_1 ...S_k where all nodes in S_i have the same set of parents. *M* contains sets M_1 ...M_k where all nodes in M_i have the same set of children. A clan C will have as sources nodes a subset of some S_i (S_{iC}) and as sinks nodes some subset of M_j (M_{jC}). C contains nodes $S_{iC} \cup M_{jC} \cup \{ v_h \mid v_h \notin S_{iC} \cup M_{jC}$ and v_h is both a descendant of some node in S_{iC} and an ancestor of some node in M_{jC}. Furthermore v_h does not have any parent nor any child outside of C$\}$. Once clans have been found, they are placed in a parse tree according to their size placing the largest clan (the entire graph) at the root with the next largest clans as their children. The tree root corresponds to the entire graph and for any tree node, its children are its maximal subclans.

Valdes, Tarjan and Lawler describe a linear-time algorithm for recognizing and parsing VSP DAGs[19.]. Their algorithm produces a binary parse tree, and the same DAG may be represented by several non-isomorphic binary trees. In contrast, our parse tree of VSP DAGs illustrates the canonical composition. It is not binary, and the tree levels alternate between representing series and parallel clans.

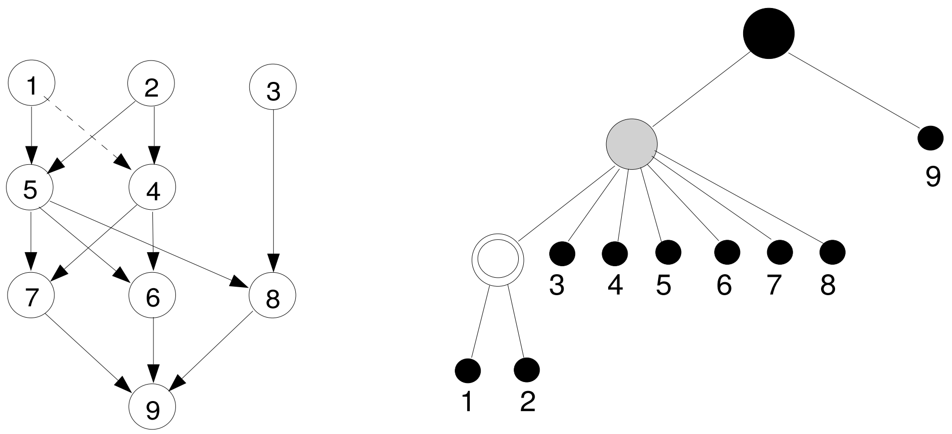
4.3 Decomposing Primitives

Primitive clans represent a special challenge. They do not fall into the clear-cut categories of nodes that should be laid out horizontally or laid out vertically. In addition, primitive clans can be arbitrarily large, and there can be indefinitely many of them. Parsing DAGs with primitives is time consuming ($O(n^2)$), and arbitrary graph generation is a problem since a list of all primitive graphs must be included in the production possibilities to create any specific graph. To facilitate layout, CG employs a heuristic to eliminate primitive clans. Primitive clans are augmented with edges until series or parallel subgraphs can be identified.

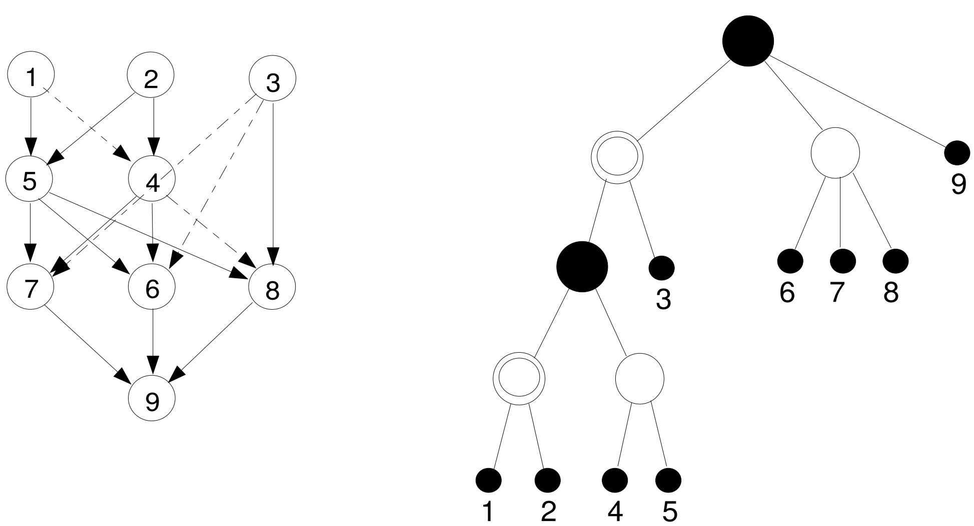
Specifically, let $S_1 \ldots S_k$ be the source nodes of primitive clan C. For each pair S_i, S_j with children $C_i = \{C_{i1}, C_{i2}, ..C_{ik} \mid (S_i, C_{ip}) \in E, 1 \le p \le k\}$, $C_j = \{C_{j1}, C_{j2}, ..C_{jq} \mid (S_j, C_{jp}) \in E, 1 \le p \le q\}$ where $C_i \cap C_j \neq \emptyset$, add edges (S_i, C_{jk}) and (S_j, C_{ik}) where $(S_i, C_{jk}) \notin E$ and $(S_j, C_{ik}) \notin E$. The subgraph, $S_i \cup S_j$ is then a pseudo-parallel clan linearly connected to the union of the children. The primitive excluding $S_i \cup S_j$ is recursively parsed until no primitives remain. Figure 8. shows the steps in decomposing the primitive graph G



(a) graph & parse tree (with primitive clan)



(b) edge (1,4) augmented to create pseudo-parallel clan {1,2}



(c) edges (3,6), (3,7), & (4,8) augmented to create pseudo-parallel clan {3,{1,2,4,5}}

Figure 8. Decomposing Primitives

After the primitives are decomposed, the interior nodes of the parse tree labels show only series and parallel clans. A bounding box attribute for each tree node, *x*, specifies the length and width of a rectangle containing the graph nodes in *x*'s subtree. The length and width are computed for each series and each parallel clan from the attributes of the clan's children. The children of a series clan are displayed vertically and the children of an parallel clan are displayed horizontally in our example layout.

5. Spatial Analysis for Node Layout

The parse tree of the graph is used to provide geometric interpretations to the graph. A "bounding box" with known dimension 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 of the bounding boxes and inherited attributes compute the bounding box and node locations. The attributes can vary with the application to present a customized visual representation of the graph. For illustrative purposes we choose simple attributes we call *natural* that give a balanced layout, both vertically and horizontally.

5.1 Bounding Boxes

A bounding box, associated with each node of the parse tree, specifies the allotted area for a subgraph. That area need not be rectangular, but the natural attributes will illustrate this case. An algebra defines the synthetic attributes that specify the bounding boxes by computing the parameters of the bounding boxes at a tree node from the bounding boxes of its children. The intrinsic attributes of singleton DAG nodes (or equivalently 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 give the unit square. A series 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 parallel 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 **series** 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 **parallel** node with children C_1 ...C_k.

As an example consider the parse tree in Figure 7. The bounding box dimensions are indicated by the (i,j) pairs in Figure 9.

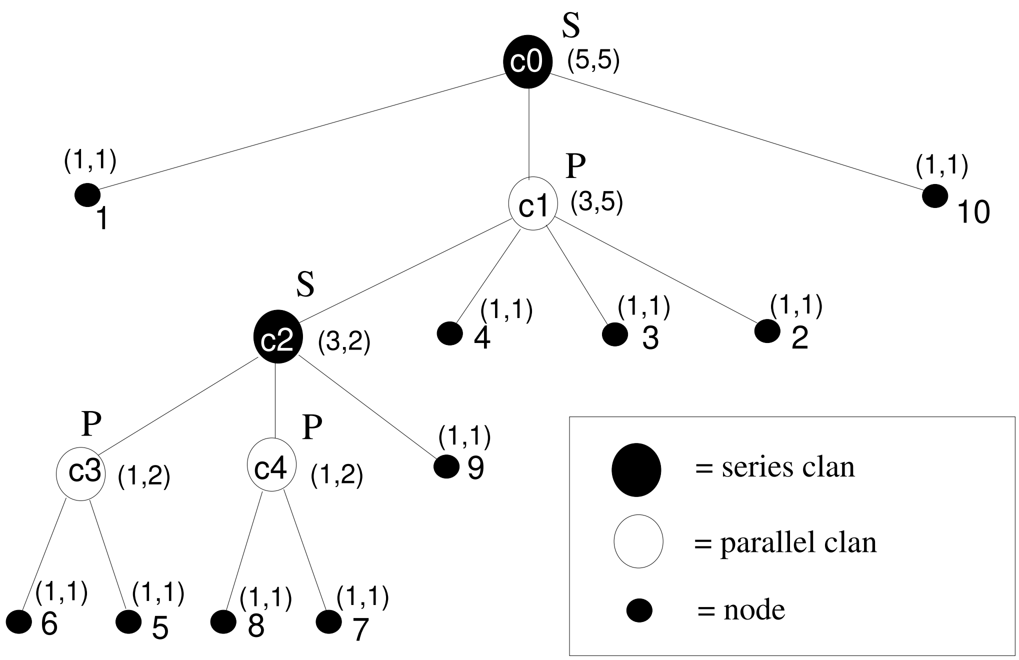


Figure 9. Parse tree with labeled bounding box labels

5.2 Bounding Box Placement

After the spatial requirements for each bounding box have been computed through the synthetic attribute, the bounding box must be mapped onto coordinates in the planar window. Inherited attributes that describe a bounding box's location place a nodes's bounding box within the bounding box of the parent and insures 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. To achieve an aesthetically pleasing 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 denote (0,0), the location (x,y) where x, y > 0, 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 series 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. Similarly, a tree node N that is a child of an parallel 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 **series** 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 **parallel** 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 series 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 parallel node I, the rectangle in which D is to be centered has length I.l and width D.w. Figure 10. shows the node layout for the parse tree of Figure 9. and the graph of Figure 6.

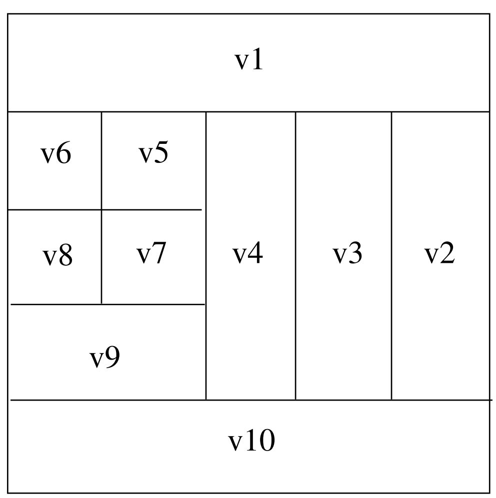


Figure 10. Node Layout

6. Drawing edges

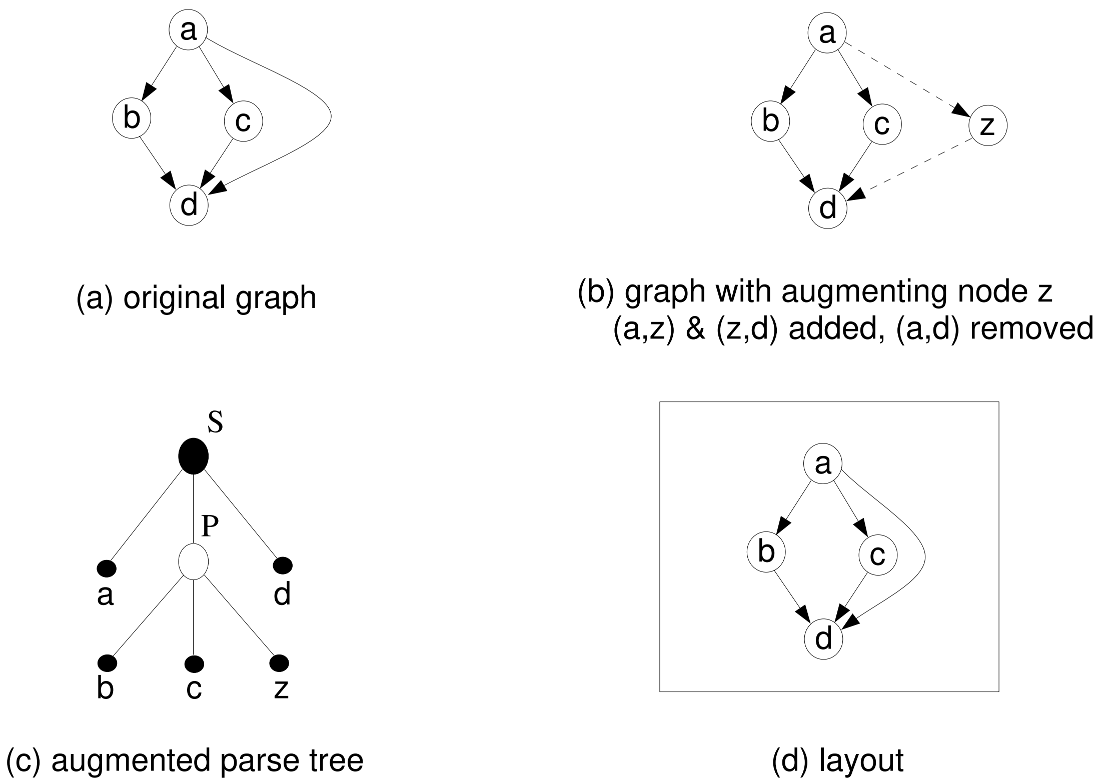
Using the parse tree to place nodes is simple and elegant and provides for an aesthetically pleasing balanced placement. If adjacent nodes are connected by straight edges, several unacceptable visualizations may occur when the nodes are placed according to the location attributes.

- Edges could pass through nodes on their path.
- Edges might be superimposed upon other edges
- Unnecessarily long edges may be drawn.
- There may be an unnecessary number of edge crossings.

The first 3 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[16.]. 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 and may cross other edges unnecessarily. CG provides a set of heuristics to solve the long edge problems. Edge lengths are reduced when possible and long edges are routed through a few dummy nodes whose location is determined through the parse tree.

6.1 Transitive edges

Transitive edges are necessarily long edges. Before parsing, a long transitive edge, (x,y), is replaced by a dummy node z and edges (x,z) and (z,y). When the augmented graph is parsed, z is a component in an parallel subclan of the clan containing x and y. The example graph in Figure 11. illustrates this situation. The only transitive edge is (a,d). Here, node z and edges (a,z) and (z,d) are added, and edge (a,d) is deleted. The augmented parse includes new parallel clan, {b,c, z} with components {b,c} and {z}. The augmented parse tree is shown in figure 6(c). Since the only role of z is that of place holder for the dummy node, a reasonable modification of the attribute grammar would be to give the dummy node a smaller width. The layout in 6(d) illustrates the case where the bounding box of z is assigned the dimensions (1, 0.5).



(a) original graph

(b) graph with augmenting node z
(a,z) & (z,d) added, (a,d) removed

(c) augmented parse tree

(d) layout

Figure 11. Transitive edges

6.2 Reducing lengths of unnecessarily long edges

The parse tree and natural location attributes create unnecessarily long edges when a source node is adjacent to nodes whose levels are greater than one. Since the location attribute, y, is equivalent to the node level, we modify the y attribute of source node S. S.y = min(C.y) - 1 where the minimum is taken over all the children, C, of S.

6.3 Routing Long edges

The subclans of a series clan can have edges between them. If the connected subclans are not adjacent children in the clan, the edges will span more than one level. However, within parallel clans, no two nodes may be connected by an edge. For these reasons, the algorithm needs to consider only routing edges between nodes within a common series clan. Because nodes in parallel clans are not connected, there are no edges intersecting intermediate subclans and parallel clans can be created for edge routing.

Two heuristics modify the parse tree by adding dummy nodes through which the long edges are routed. The *short clan heuristic* routes long edges within a series clan and *inter-clan heuristic* routes long edges between different subclans.

The short clan heuristic is invoked when node or series clan C has bounding box height less than the bounding box height of its parent. Dummy nodes are added both at the top and bottom of the clan. For each clan source, dummy nodes are added for each in-coming edge, and for each clan sink, dummy nodes are added for each out-going edge. Figure 12. shows a graph with the dummy nodes generated by this heuristic.

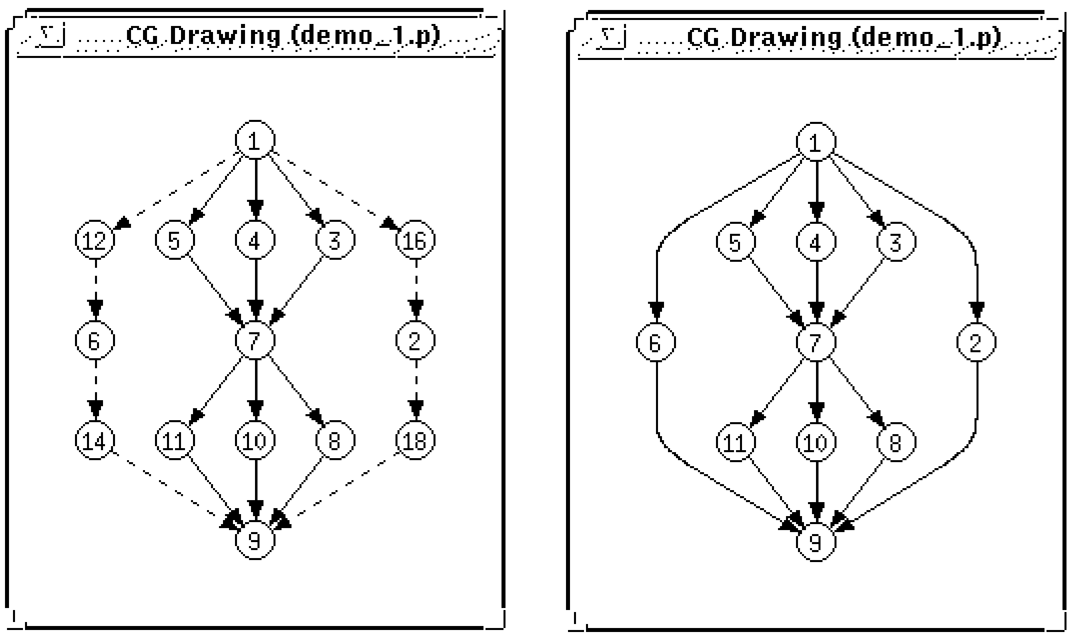


Figure 12. Dummy Nodes Added by the Short Clan Heuristic

The inter-clan heuristic routes edges between nodes in different series clans. For edge (x,y), let lca(x,y) be the least common ancestor of nodes x and y in the parse tree. By definition, lca(x,y) must be a series node. Let i_x and i_y be the parallel children of lca(x,y) that are ancestors of x and y, respectively. The inter-clan heuristic adds three sets of dummy nodes to the parse tree:

1. For all series clans in the traversal from x to lca(x,y), dummy nodes are added as children in each clan to the right of the ancestor of x.
2. For all parallel clans that are children of lca(x,y) between i_x and i_y, a dummy node is added in the appropriate location.
3. For all series nodes in the traversal down the tree from lca(x,y) to y, dummy children are added for each node to the left of y's ancestor.

Figure 13. illustrates this situation. The singleton dummy nodes will be added as children to the blackened nodes.

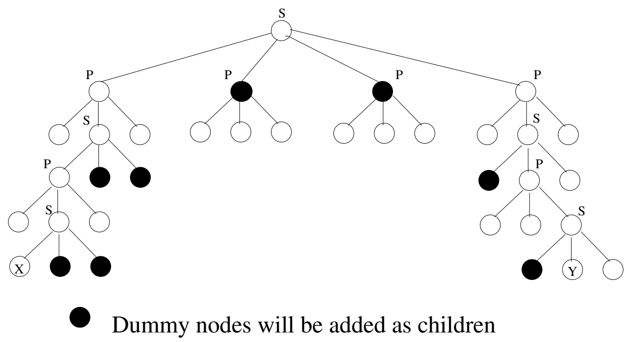
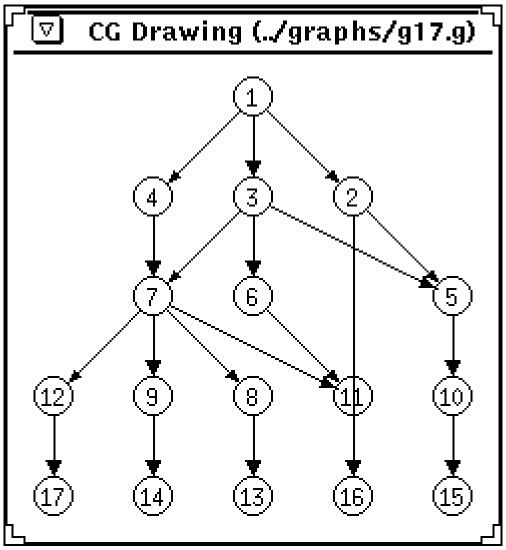
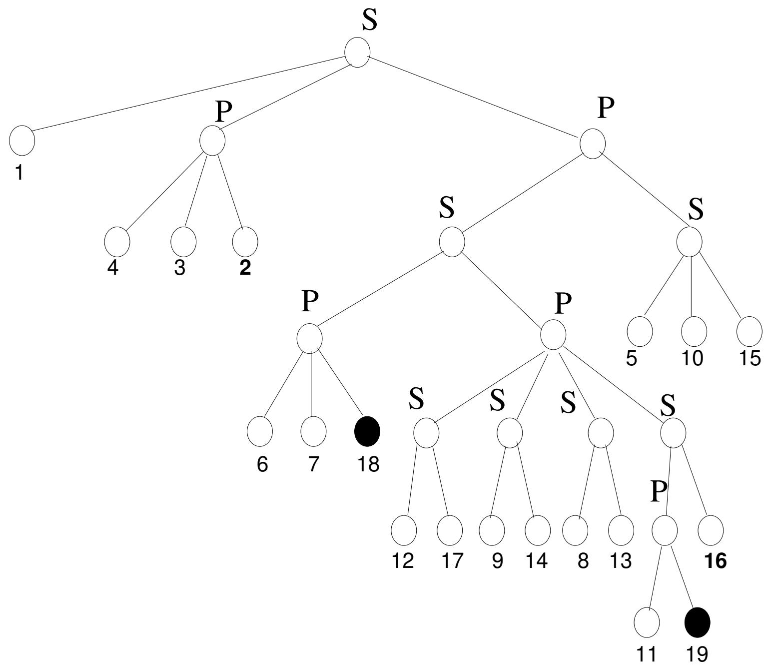


Figure 13. Dummy Nodes Added by Heuristic B

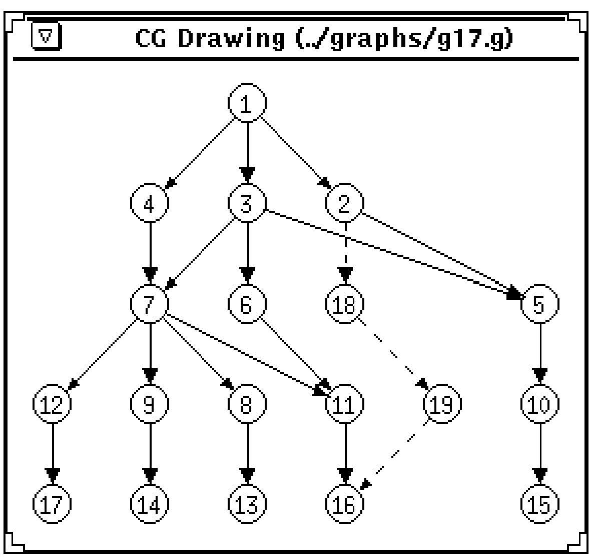
The typical way to route long edges in digraphs is to place dummy nodes to guide the edge on each level of the graph between the head and tail nodes for the edge. The inter-clan heuristic is an improvement of this process in several ways. There are fewer dummy nodes required because CG adds only one node per intermediate clan rather than one per intermediate level. Also there are fewer edge bends in this scheme because the dummy nodes are inserted in the best available spot horizontally. That is, if the x-coordinates of the edge head and tail are X_1 and X_2, resp., the dummy nodes are inserted between nodes whose x-coordinates are between X_1 and X_2. This horizontal placement also reduces the number of potential edge crossings. Figure 14. shows a graph where the dummy nodes were generated by this heuristic.



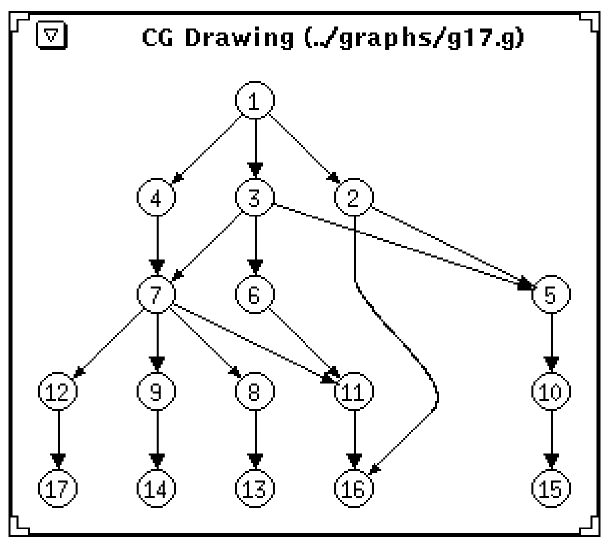
(a) original graph (edge (2,16) goes through node 11)



(b) dummy nodes (18 & 19) added for long edge (2,16)



(c) dummy nodes 18 & 19 added



(d) layout

Figure 14. Graph with Dummy Nodes Added by Inter-clan Heuristic

6.4 Reducing Bends and Unnecessary Spacing

The addition of dummy nodes by the inter-clan heuristic can cause unnecessary bends in the long edges. After the dummy nodes are identified and all nodes are placed, CG checks the added dummy nodes of each long edge. In a long edge from node n_0 to n_k passing through dummy nodes $n_1..n_k$, if there are no actual nodes in the rectangular region bounded by the n_{i-1} and n_{i+1}, the dummy node n_i is removed to reduce the unnecessary bend. After reducing bends, some spaces may be created due to the dummy nodes' removal. CG shifts part of graph to reuse those spaces and reduce unnecessary spacing.

6.5 Smoothing edges.

The CG system eliminates sharp corners by using B-splines. When an edge connects two nodes (an actual node with a dummy node, or two dummy nodes), the spline technique described by Enns[8.] and Farin[9.] is used. During the addition of the control points for the spline, if an edge is too close to a node, extra control points are added to avoid intersecting the edge with the node.

7. Extensions to General Directed Graphs

The techniques described in this paper can be used to draw any directed graph. Cycles can temporarily be removed.

7.1 Extension to Cyclic Graphs

A simple transformation is required to apply the graph decomposition method to cyclic graphs. Cycles can be found in a depth-first graph traversal. To break a cycle, the edge that identifies the cycle is given the reverse orientation. When the layout is ready, its orientation will be corrected. This method of breaking cycles will show a cycle not as a circular arrangement of nodes, but as a vertical line of nodes with an edge connecting the bottom to the top. This view is consistent with some applications such as the visualization of program control flow graphs. The general philosophy of a top to bottom flow for directed graphs is supported by this layout, with only few edges reversing that direction.

For a cycle with only two nodes has edges (x,y) and (y,x). To insure that these are placed in different locations, a dummy nodes is required to guide the back edge. In longer cycles, the back edge is a transitive edge and a dummy node is added as described in section 4.

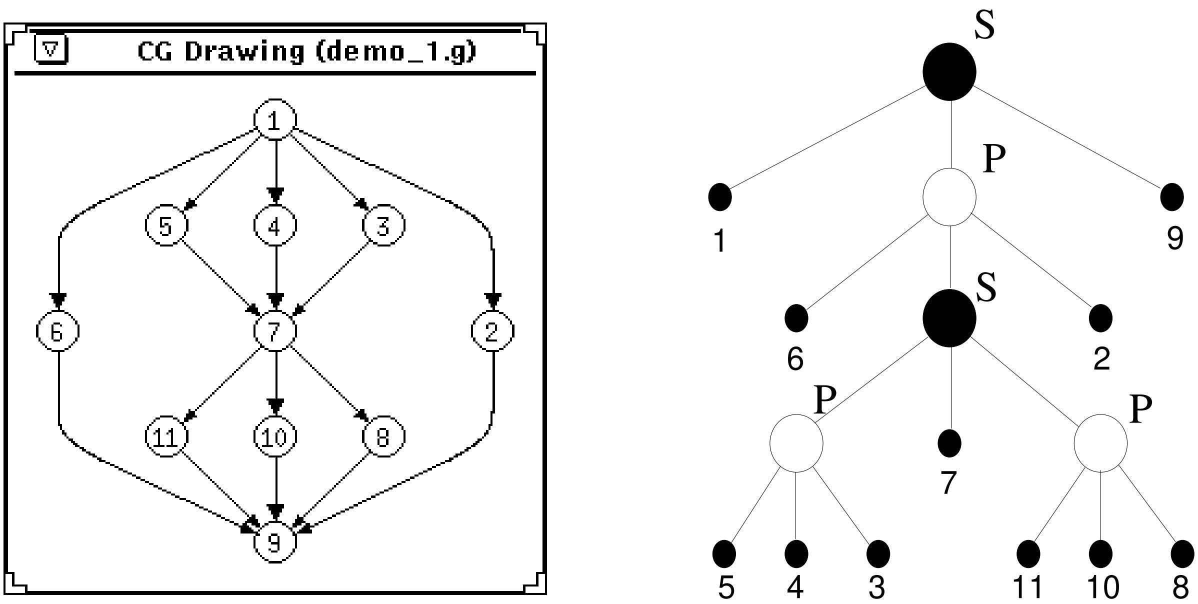
8. Viewing Subgraphs

Because the graph layout is based on a parse tree created by extracting subgraphs, it is possible to abstract those subgraphs and represent them by a single node. CG supports contraction and expansion.

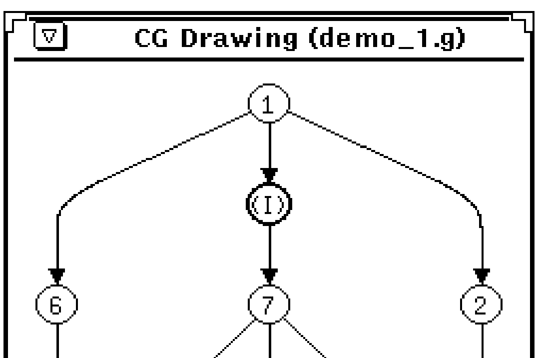
8.1 Clan Expansion and Contraction

Since clans are defined as sets of nodes with identical ancestors and descendants within the rest of the graph, clans can easily be contracted to a single node. By selecting a single node, the user can contract the smallest non-trivial clan containing that node into a single node. See Figure 15. Any node not in the clan that was connected to a clan source or sink will be connected to the contracted node. By allowing segments of the graph to be contacted, the user can simplify graphs for viewing by contracting those parts which are not relevant to the investigation. Contracted nodes can be expanded to show the original clan configuration. Similarly, it is possible to display only the clan, ignoring the rest of the graph.

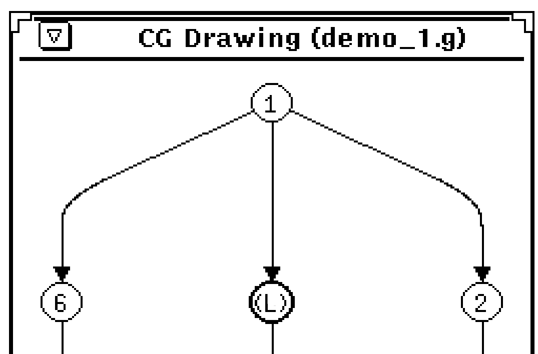
The contracted clans can be selected exactly like the nodes. The expand button can then be used to expand selected clans to reveal their contents. This expansion and contraction of clans allows the user to concentrate on specific subgraphs while browsing the digraph.



(a) original graph & parse tree



(b) selecting node 3, 4, or 5 will contract parallel subclan {3, 4, 5}



(c) selecting node 7 will contract series subclan {3, 4, 5, 7, 8, 10, 11}

Figure 15. Clan Contraction

9. Example Application

One application area that is currently interested in this work is that of social networks. Figure 15. is an example of data reported in the 1940's by anthropologists [3]. They tallied the co-attendence of 18 women over a series of 14 small informal social events. Freeman and White began with their person by event matrix and constructed a Galois lattice that represents the person-person, the event-event and the person-event dependencies[10.]. It shows that there are two pretty clear-cut sub-groups of women, and three kinds of events: those involving one group of women, those involving the other group, and those events that bridge the two. The 65 points in the graph picture the overall dependency structure. Each point represents some collection of women and some collection of events. The uppermost point is the collection of all women and the null set of events. The lowermost point is the universal set of events and the null set of women.

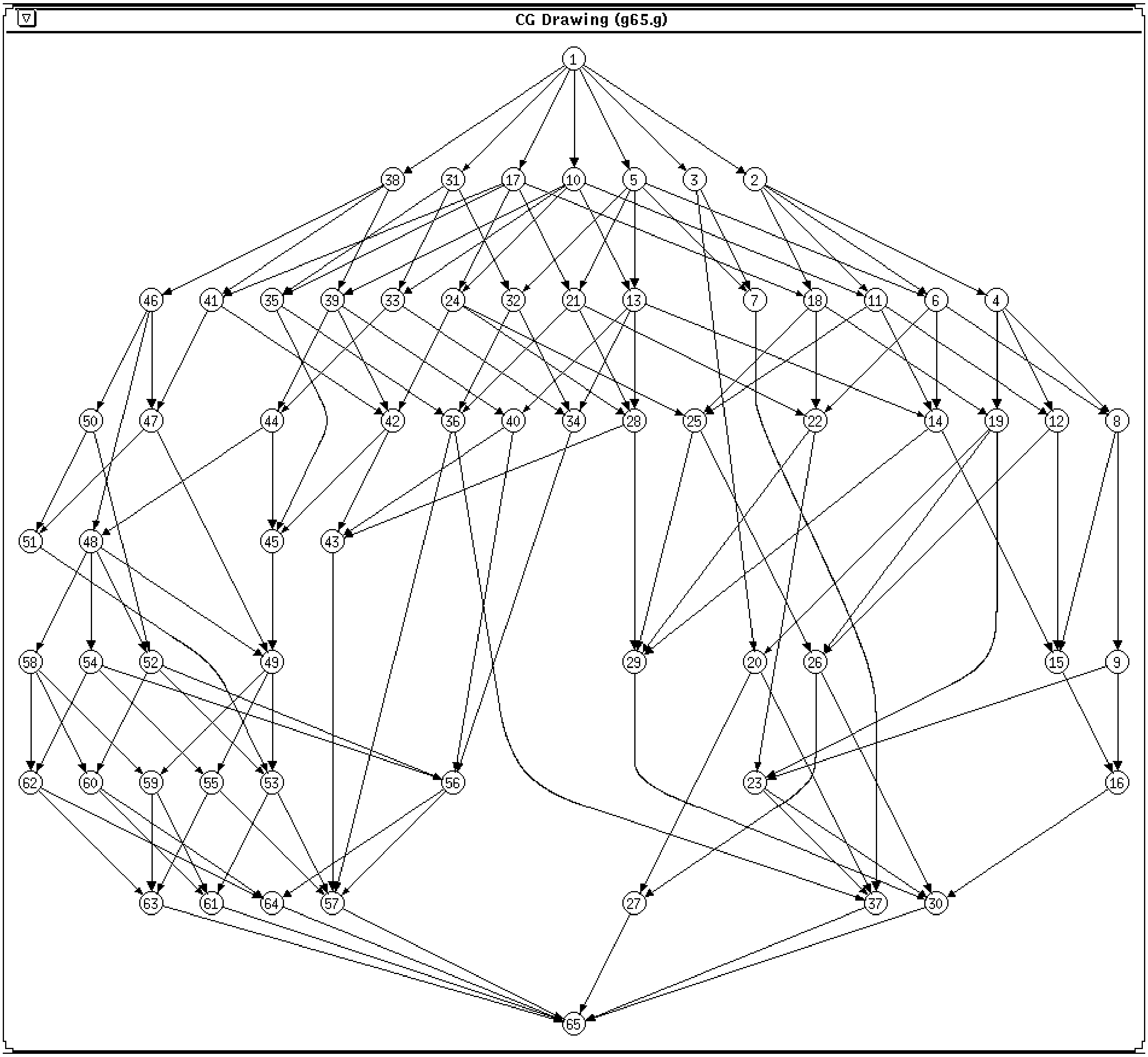


Figure 16. Social Networks Lattice

10. Conclusions

One of the major differences between the graphs drawn by CG and hierarchial systems [12][2] is that the nodes in CG's graphs use both horizontal and vertical attributes to determine their location while hierarchial systems use only a vertical attribute. Hierarchical drawings partition nodes into levels, and all nodes of the same level are placed on the same horizontal axis. By using our graph decomposition technique, CG is able to determine balanced vertical spacing as well as balanced horizontal spacing.

CG's process of contracting subgraphs is a capability not found in any other system we've studied. This tool can help the viewer to remove unnecessary details while studying subsections of the graph. Large graphs can be viewed more clearly with this organizational scheme.

The graph drawing community discusses finding "meaning" within graphs. That is, when given a purely textual description of a graph, present it visually so that the viewer will be able to organize her/his thoughts about it. Because of the property that a clan is a subgraph whose nodes are connected to nodes outside the subgraph in an identical way, a clan is a cohesive set of nodes, one which the viewer can recognize. A clan is a natural grouping which CG displays coherently to the viewer and which helps organize the graph for the viewer.

By labeling the parse tree with attributes that will be used in layout, CG has the potential to display graphs of great variety. The basic concepts of parsing a graph and defining a layout attribute grammar can be used to generate varying views of graphs. Node shapes and sizes can be viewed as intrinsic attributes and new rules can be formulated to evaluate inherited and intrinsic attributes.

References

1. F. J. Brandenburg, "Layout Graph Grammars: the Placement Approach", *Lecture Notes in Computer Science, 532, Graph Grammars and Their Application to Computer Science,* Springer-Verlag, Berlin, 1991..

2. M. J. Carpano, "Automatic Display of Hierarchized Graphs for Computer Aided Decision Analysis", *IEEE Trans. on Systems Nab and Cybernetics,* vol. SMC-10, no. 11, pp 707-715.

3. A. Davis, B. Gardiner and M. Gardiner, 1941, *Deep South*, Chicago: U. of Chgo. Press

4. G. Di Battista, P. Eades, R. Tamassia, I. Tollis, "Algorithms for Drawing Graphs: an Annotated Bibliography", available via ftp from wilma.cs.brown (128.148.33.66) in file pub/papers/compgeo/gdbiblio.tex.Z.

5. P. Eades and N. Wormald, "Edge Crossings in Drawings of Bipartite Graphs", *Algorithmica,* Vol. 11, no. 4, April 1994, pp. 379-403..

6. 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.

7. 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.

8. S. Enns, “Free-Form Curves on Your Micro,” *BYTE*, vol. 11, no. 13, pp. 225-230, Dec. 1986.

9. G. Farin, *Curves and Surfaces for Computer Aided Geometric Design*. Boston: Academic Press, 1988.

10. L. C. Freeman and D.R.White, 1993, "Using Gaiois lattices to representnetwork data," in P.V. Marsden, ed., *Sociological Methodology* 1993, Oxford: Blackwell, 127-146.

11. 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.

12. 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.

13. M. Himsolt, "A View to Graph Drawing Algorithms through GraphEd," *Proceedings of the ALCOM International Workshop on Graph Drawing,* 1993, pp. 80-81.

14. Koutsofios and S. North, “Drawing Graphs with *dot*”, *Dot User’s Manual,* AT&T Bell Labs, Murray Hill, N. J., 1994

15. C. L. McCreary and A. Reed“A Graph Parsing Algorithm and Implementation,” Tech. Rpt. TR-93-04, Dept. of Comp. Sci and Eng., Auburn U. 1993.

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

17. J. N. Warfield, “Crossing Theory and Hierarchy Mapping,” *IEEE Trans. on Syst., Man, and Cybernetics*, vol. 7, no. 7, pp. 505-523, Jul. 1977.

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

19. J. Valdes, R. E. Tarjan, E. L. Lawler, "The Recognition of Series Parallel Digraphs," *SIAM J. Comput.,* vol. 11, No. 2, May 1982., pp. 298-313.

20. G. Zinssmeister and C. McCreary, "Drawing Graph with Attribute Graph Grammars," *Graph Grammar Workshop,* Williamsburg ’94, pp. 355 - 360.