A DIVIDE-AND-CONQUER ALGORITHM FOR THE AUTOMATIC LAYOUT OF LARGE DIRECTED GRAPHS

by

Eli B. Messinger, Lawrence A. Rowe, and Robert H. Henry

Memorandum No. UCB/ERL M89/23

17 February 1989

A DIVIDE-AND-CONQUER ALGORITHM FOR THE AUTOMATIC LAYOUT OF LARGE DIRECTED GRAPHS

by

Eli B. Messinger, Lawrence A. Rowe, and Robert H. Henry

Memorandum No. UCB/ERL M89/23

17 February 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

A DIVIDE-AND-CONQUER ALGORITHM FOR THE AUTOMATIC LAYOUT OF LARGE DIRECTED GRAPHS

by

Eli B. Messinger, Lawrence A. Rowe, and Robert H. Henry

Memorandum No. UCB/ERL M89/23

17 February 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

A Divide-and-Conquer Algorithm for the Automatic Layout of Large Directed Graphs [1]

Eli B. Messinger [2]
Lawrence A. Rowe [3]
Robert H. Henry [4]

Abstract

Pictures are a useful medium for communicating abstract information. Pictures exploit a person's natural abilities to recognize and understand visual patterns. However, the differences between an effective and a confusing presentation of the same data can be subtle, and often depend on the layout.

This paper discusses methods for automatic presentation of relational information in the form of directed graphs. An overview of the graph-layout problem is presented, along with a summary of several different layout algorithms. A new divide-and-conquer layout algorithm is described in detail. This algorithm produces especially good results on large graphs of several hundred vertices.

1. Introduction

The increasing acceptance of the bit-map display and mouse paradigm for user-interfaces (e.g., on an X-Windows workstation or an Apple Macintosh) allows information to be presented to users in pictorial, as opposed to textual, representations. An advantage of these pictorial representations is that they exploit a user's natural ability to recognize graphic patterns [Rob87b]. A commonly used form of graphic representation is a vertex and edge graph.

Many computer applications allow the direct manipulation of human generated graphs. For example, Batini, *et. al*, have developed a system for creating and editing database schema through the manipulation of *Entity-Relationship Diagrams* [BTT84a]. Other graph systems automatically display application-generated graphs, such as compiler-generated parse trees [AHY88]. Lastly, several interactive systems, such as *MacProject* [APP84] allow a user to manually draw and manipulate graphs, such as PERT Charts or organizational trees.

Although the electronic-manual approach to graph layout offers a small improvement over actual paper-and-pencil drawing, two problems are very evident. First, experience has shown

that people find it difficult to lay out graphs with many vertices and edges. Second, it is difficult for a user to produce a layout when the data is generated by an application (e.g., a parse tree generated by a compiler). An algorithm is needed to produce graph layouts automatically. That is, given a formal vertex and edge set representation of a graph, an algorithm is needed that automatically assigns planar coordinates to the vertices and edges.

In making these coordinate assignments, one must keep in mind that the layouts are intended for visual consumption, and thus must be easily "read" and aesthetically pleasing to the eye. Recent work on automatic layout algorithms has focused on graph-syntactic and geometric aesthetics (or *constraints*) such as minimized edge crossings and balanced distribution of graph elements in the plane. Tamassia, Batini and Di Batista [TBD87] and Messinger [Mes88] have each produced surveys of the field and derived taxonomies of layout criteria from them.

This paper describes a new layout algorithm, called *COMPOZE*, that uses a divide-and-conquer approach to graph layout. Our motivation for studying graph layouts was borne from a need to examine program call-graphs. We produced an implementation of Sugiyama, Tagawa and Toda's algorithm (henceforth referred to as the *STT algorithm*) [STT81], called *GRAB*. [5] *GRAB* combines the automatic layout of the STT algorithm with an interactive graphical interface, and runs on a Sun Microsystem workstation [RDM87].

After devising several improvements to the basic STT algorithm [Dav85] and conducting several experiments to measure its performance (both layout time and layout quality), we concluded that the algorithm was ineffective on graphs larger than 50-100 vertices [Mes88]. Graphs of a few hundred vertices required several minutes of layout time on a single-user Sun-3/75. Additionally, the layout quality for large graphs was generally unacceptable, containing numerous edge crossings and no overall structure.

These two problems motivated the development of the divide-and-conquer scheme. By subdividing a large graph into subgraphs, laying them out separately, and pasting them together to create a layout of the total graph, *COMPOZE* is able to produce layouts of higher quality (as measured by objective constraints) in as little as 20% of the time used by the STT algorithm. Additionally, the partitioned nature of the final layouts is preferred by users, who suggest that the geographic separation of subgraphs allows the eye to break down a complex graph into manageable pieces. Finally, the partitioning of the layout process suggests ways to perform incremental layout changes and to develop a parallel version of the algorithm.

The remainder of this paper is organized as follows. Section 2 surveys related work. Section 3 describes the *COMPOZE* layout algorithm. Section 4 presents the results of a series of

experiments conducted to compare *COMPOZE* and *GRAB*. Finally, section 5 summarizes the results presented in this paper and suggests directions for future research.

2. Related Work

This section presents a brief overview of previous work on graph layout algorithms. Comprehensive surveys are given elsewhere (e.g., Tamassia *et al.* [TBD87] or Messinger [Mes88]). Readers unfamiliar with graph terminology are referred to basic graph theory books [BoM76, Har69].

Many automatic layout algorithms have been developed, reaching back at least as far as Knuth's suggested system for automating the layout of flow charts [Knu63]. Most layout algorithms have been developed for special subclasses of graphs, and with particular application-dependent layout constraints in mind.

For example Reingold and Tilford's algorithm for trees [ReT81] works only for directed tree graphs, and ensures a number of customary constraints, such as the alignment of equal depth vertices on a horizontal level, and centering parent vertices between their children horizontally. Table 2-1 lists several notable layout algorithms.

This paper describes an algorithm to produce hierarchical layouts of directed graphs. A *hierarchical layout* of a graph divides the graph's vertices into a number of subsets called *levels*. The vertices in each level are laid out on a horizontal line, and the levels are stacked vertically. Figure 2-1 depicts one hierarchical layout of a directed graph.

The goal of the hierarchical layout is to clearly display the ancestral relationships between vertices. This goal is accomplished by selecting the level subsets so as to maximize the edge-flow in a particular direction (e.g., downwards in figure 2-1). This positioning allows a reader to identify ancestor/descendant relationships by identifying physical above/below relationships between connected vertices. That is, in a perfect hierarchy, all of a vertex's ancestors appear physically above it, and all of a vertex's descendants appear physically below it.

Many algorithms have been developed to solve the hierarchical layout problem. Warfield published an early algorithm that assigned vertices to levels to insure hierarchical positioning (i.e., all of the graph's edges directed in one direction) [War76]. The algorithm starts by finding the subset of vertices with no successors and assigns them to the *bottom level*. These vertices are removed from the graph and the process is repeated to find the next level up. The algorithm continues until all vertices have been placed.

Author(s)	Graph Domain	Special Attributes	Reference
Woods	Undirected Planar	Non-Manhattan Grid Embeddings	[Woo81]
Chiba, Yamanouchi, and Nishizedi	Undirected Planar	Convex Layouts	[CYN84]
Tamassia	Undirected Planar	Manhattan Embeddings with Minimum Edge Bends	[Tam87]
Lipton, North, and Sandberg	General Undirected	Symmetric Drawings	[LNS85]
Makinen	General Undirected	Circular Drawings	[Mak88b]
Reingold and Tilford	Directed Trees	Hierarchical Tree Drawings	[ReT81]
Sugiyama, Tagawa, and Toda	General Directed	Hierarchical, Reduced Edge-Crossings	[STT81]
Rowe, *et al.*	General Directed	Hierarchical, Cycles, Interactive Graph Browser	[RDM87]
Robbins	General Directed	Hierarchical, Crossing Reduction/Speed Trade-off	[Rob87b]

Table 2-1 — Selected Automatic Layout Algorithms

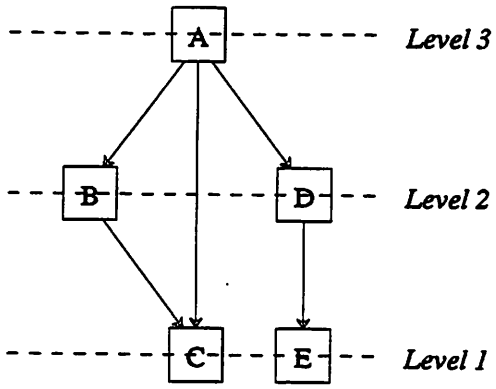


Figure 2-1 — A Hierarchical Layout of a Directed Graph

For cyclic graphs Warfield described a method that identifies maximal-cycles and compresses them into *proxy vertices*. These proxies act as super-vertices, taking on all of the predecessors and successors of the cycle's vertices, and reducing the graph to an acyclic graph. Carpano discussed a post-processing step in which the proxies are expanded in a three-dimensional layout [Car80]. Meyer suggested using pre-determined canonical forms to draw proxies of different sized cycles [Mey83]. Davis devised an alternate method that involved removing cycles by temporarily reversing selected edges [Dav85].

After assigning the vertices to levels, the next problem is to reduce the number of edges that cross other edges. Warfield developed a heuristic method for 2-level graphs that is based on alternately permuting the vertices of the top and bottom levels [War77]. Eades and Kelley showed that the crossing minimization problem is NP-hard for 2-level hierarchies with the vertices on the second level held fixed [EaK86], so Warfield's use of heuristics was warranted.

Delarche developed a faster crossing reduction heuristic for 2-level hierarchies [Del79], which was also described by Carpano [Car80]. This algorithm combined Warfield's permutation strategy and a sorting measure called a *barycenter* [Tut63]. The barycenter of a vertex is the average ordinal position of the vertex's neighbors. For example, in figure 2-2 the barycenter of vertex A is the average position of its neighbors E and G, or 1. Similarly the barycenter of vertex H is the average position of vertices B and C, or 1.5.

Delarche's crossing-reduction algorithm alternately sorts the top and bottom levels according to their barycenters. As each level is sorted, the barycenters of the opposite level are

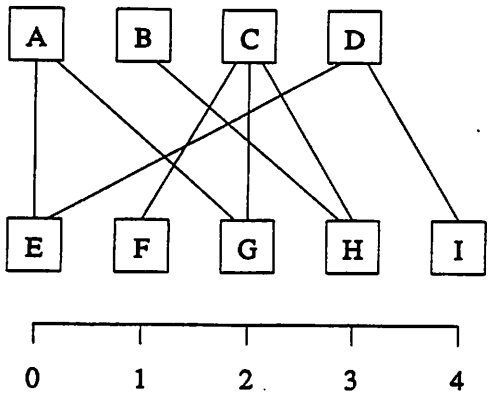


Figure 2-2 — Computing the Barycenters

recalculated. This process continues until a stable configuration is reached. Carpano reported that Delarche showed convergence "under the usual convergence assumptions" [Car80, pp. 710].

The idea is that the barycenters tend to align connected vertices vertically, to create vertical edges that do not cross. Although the convergence results do not show that the final configuration will be minimal in terms of the number of edge crossings (or in fact better than the starting configuration!), Delarche was able to reduce the number of crossings by 30 to 50 percent across several sample graphs.

Carpano generalized Delarche's 2-level sorting to reduce crossings in k-level hierarchies. Generalizing the algorithm involved three steps: 1) reducing the k-level hierarchy to a series of connected 2-level hierarchies, 2) generalizing the barycenter measure, and 3) generalizing the sorting iterations.

The first transformation reduces long-edges that span non-adjacent levels to a series of connected single-level spanning segments. This is accomplished by inserting new *dummy vertices* at intermediate levels. For example, in the hierarchy of figure 2.1, the edge $A \rightarrow C$ would have a *dummy vertex* inserted at level 2.

The second generalization replaces the single barycenter with a pair of measures, the *up*-barycenter and the *down*-barycenter. The *up*-barycenter is the average ordinal position of a vertex's predecessors, while the *down*-barycenter is the average ordinal position of a vertex's

successors.

The third generalization modifies the order in which levels are sorted. Carpano's k-level algorithm iteratively sorts levels 1 and 2 as described above, and then sorts levels $3, 4, \cdots, n$ in ascending order. For $3 \leq i \leq k$, the vertices in level i are sorted according to their *up*-barycenters. After level i has been sorted, the new vertex positions are propagated back through the graph by sorting level j (j ranging from $i-1$ down to 1) according to the *down*-barycenter of each vertex.

Sugiyama, Tagawa and Toda extended Carpano's algorithm in two ways [STT81]. First, they altered the sorting order so that the levels are sorted $2, ..., k$ by their *up*-barycenter and then they are sorted $k-1, ..., 1$ by their *down*-barycenter. And second, they added a third phase to the algorithm that attempts to minimize edge lengths and straighten long-edges.

Meyer [Mey83] and Davis [Dav85] further extended the algorithm. They generalized the barycenter measure by introducing an *up-down*-barycenter. This measure averages the position of a vertex's predecessors and successors. As noted earlier, they also introduce a heuristic for handling graph cycles without resorting to proxy vertices. This algorithm was used in an interactive graph browser, called *GRAB*, developed by Rowe *et al.* [RDM87]. Experience with *GRAB* motivated the development of the compositional layout system described below in section 3.

Other researchers have investigated various aspects of this algorithm, including alternatives to barycentric sorting [EaK86, GNV88, Mak88a, Mak88c], an improved version of Sugiyama's first-phase level-assignment problem [GNV88] and an exact solution to Sugiyama's third-phase edge-length minimization problem [GNV88].

Several other approaches to hierarchical layout have been investigated, including Robins' LISP-based *ISI Grapher* [Rob87a, Rob87b], May, Iwainsky and Mennecke's hierarchical circuit layout system [MIM83] and Majewski, Krull, Fuhrman and Ainslie's schematic layout system [MKF86].

3. COMPOZE: A Compositional Graph Layout Algorithm

Experience with the *GRAB* implementation of the Sugiyama algorithm showed that the layout scheme was too slow for interactive applications. Graphs of only a few hundred vertices required several minutes of layout time on a Sun-3/75 workstation. Dynamic profiles of the system showed that over 95% of the total layout time was spent sorting and resorting the levels during the crossing minimization phase (i.e., phase 2). After several failed attempts to find a

faster heuristic solution that would produce layouts of equivalent quality, we decided to utilize a divide-and-conquer algorithm.

The *COMPOZE* system lays out graphs by partitioning them into subgraphs, laying out the subgraphs separately, and composing the constituent sub-layouts into a layout of the original graph. To design *COMPOZE*, four problems had to be solved: 1) how to partition the input graph into subgraphs, 2) how to layout the subgraphs, 3) how to position the subgraph layout relative to each other in the composition phase, and 4) how to layout graph edges that span between different subgraphs.

The input to *COMPOZE* is a graph, $G = (V, E)$, and a partition of the graph into subgraphs specified as non-overlapping subsets $\{V_1, \cdots, V_n\}$. These subsets induce a partition of the entire graph:

$$\{(V_1, E_1), ..., (V_n, E_n)\} \cup E_{INTER}$$

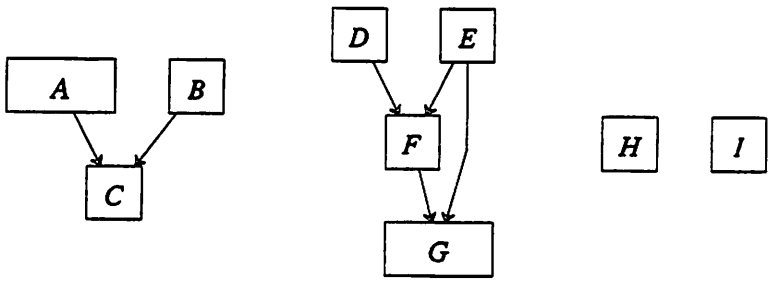
where E_i are the edges connecting vertices in vertex subset V_i, and E_{INTER} are the inter-subgraph edges connecting vertices in different vertex subsets. The *COMPOZE* algorithm operates in three phases:

1. *Subgraph Layout.* Each subgraph is laid out using a modified version of the Sugiyama algorithm.
2. *Composition.* These subgraph layouts are positioned in the final layout plane.
3. *Interedge Routing.* The interedges, those edges with endpoints in two different subgraphs, are added to the final layout.

Figure 3-1 shows an example of this process. In the top figure the subgraphs are laid out individually. Using the bounding boxes of these layouts and the inter-subgraph edges listed in the center figure, a *metagraph* is computed as shown in the bottom figure. This metagraph, when laid out, acts as a template for the composition of step 2. After composing the subgraphs, the inter-subgraph edges are added, using the metagraph edges as a guide.

The remainder of this section is organized as follows. Section 3.1 describes several graph partitioning techniques that we investigated. Section 3.2 discusses the technique used by *COMPOZE* to recombine the subgraph layouts into a layout of the input graph. Section 3.3 describes the techniques used to route the inter-subgraph edges.

The Subgraph Layouts



The Intersubgraph Edges

Interedge	Equiv. Meta-Edge
(C, H)	(Sub 1, Sub 3)
(G, I)	(Sub 2, Sub 3)

The Metagraph

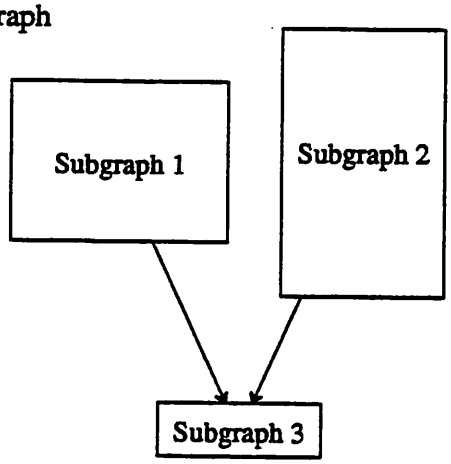


Figure 3-1 — The Metagraph as Layout Template

3.1. Graph Partitioning

In preparing a graph for layout with the *COMPOZE* system, the graph's vertex set must be partitioned in some manner. Two types of partitions were investigated: 1) application-specific partitions suggested by the semantics of the input graph (e.g., a module-based partition of a

function call-graph) and 2) graph-theoretic partitions based on syntactic partitioning algorithms such as Kernighan and Lin's [KeL70].

Each type of partition has its own set of advantages and disadvantages. In looking at semantic partitions it is clear that the primary advantage is the semantic coherency of the induced subgraphs. These groupings can be of great value in displaying graphs, especially when different partitions are used to provide contrasting views of the same graph. For example, a call-graph might be partitioned according to module-membership, data-bindings or paging behavior. The relationship between the different partitions might be used to learn more about the static and dynamic structure of a program.

On the other hand, semantic partitions are often very poor from the standpoint of display. Module partitions, for example, are often very uneven, with module size ranging from one or two to hundreds of functions. Uneven partitions produce poor layouts because they do not play to the strength of the compositional strategy, which is to use the underlying subgraph layout algorithm to layout out graphs of moderate size (e.g., 20-40 vertices).

A second problem with uneven partitions is that they increase layout time. Layout time is the sum of the times to layout the subgraphs, route the inter-subgraph edges, and recombine the subgraphs (i.e., composition overhead). An uneven partition often has an increased number of subgraphs, which increases the overhead component. Similarly, the polynomial running time of the subgraph layout algorithm is larger on unevenly sized subgraphs. For example, the algorithm takes longer to lay out two unevenly sized subgraphs of 1 and $n-1$ vertices than two evenly sized subgraphs of $n/2$ vertices.

Finally, many semantic partitions contain a large number of inter-subgraph edges. Although not inherently a problem in a divide-and-conquer layout scheme, experience with *COMPOZE* has shown that the routing of these edges is difficult, and that a large number of them quickly makes a layout unreadable.

The second type of partition that was considered is based on graph syntactic algorithms such as Kernighan and Lin's [KeL70]. The goal of this algorithm is to partition a graph of kn vertices into k evenly divided subsets of n vertices each, with a minimal number of inter-subgraph edges. The obvious advantage of such a partitioning scheme is that the goals directly relate to the requirements of the divide-and-conquer layout scheme, as discussed above.

For graphs with no natural, semantic partition, the syntactic partitioning scheme is ideal. On the other hand, for graphs with a semantic partition (e.g., a call-graph with a module partition), repartitioning the graph syntactically might be misleading. That is, a layout based on a syntactic graph, with vertices grouped by their degree of connectivity, might imply some non-

existent semantic relationship to the user.

Several other semantic graph partitioning schemes are available, including *clustering algorithms* [HuB85, SCC77] in which new clusters of highly connected vertices are "grown," or algorithms based on the technique of *simulated annealing* [JAM87, KGV83, Rom86]. Each has disadvantages when compared with the Kernighan and Lin scheme in this context. In particular, clustering algorithms have a tendency to trade subgraph evenness for minimized inter-subgraph edges, and simulated annealing is too slow.

For applications in which overall layout quality, in terms of layout time, number of edge crossings, evenness of vertex distribution, etc. is the primary concern, then a syntactic partitioning algorithm is, at this point, a must. Future divide-and-conquer layout schemes might handle uneven partitions more gracefully than *COMPOZE*, but likely never as well as they will handle higher quality syntactic partitions.

3.2. Composition

After the subgraphs have been laid out, the subgraph layouts are composed into a layout of the total graph. *COMPOZE* achieves this through the use of a *metagraph*. The metagraph consists of a *metavertex* for each subgraph of the partition, and a *metaedge* for each set of edges directed from one subgraph to another. The original input edges underlying the metaedges are called *actual edges*.

Each metavertex is sized to match the dimensions of the underlying subgraph layout. Similarly, each metaedge is width-scaled to represent the number of underlying actual edges. The metaedges are essentially channels in which the actual edges are later routed, as shown in figure 3-2.

The metagraph is laid out by a modified version of the Sugiyama algorithm. The modifications allow the algorithm to handle vertex-icons of arbitrary heights (the original formulation assumed that each vertex was represented by a dimensionless point-icon) and insures that edges enter at the top and exit from the bottom of vertex-icons. Further details are given elsewhere [Mes88].

Once the metagraph has been computed and laid out, the composition involves three steps: 1) copying the subgraph vertices, 2) merging the level structures, and 3) copying the intra-subgraph edges.

The first step copies the vertices of each subgraph into the total graph, using the coordinates of the vertices within their subgraph layouts, and the subgraph's coordinates within the

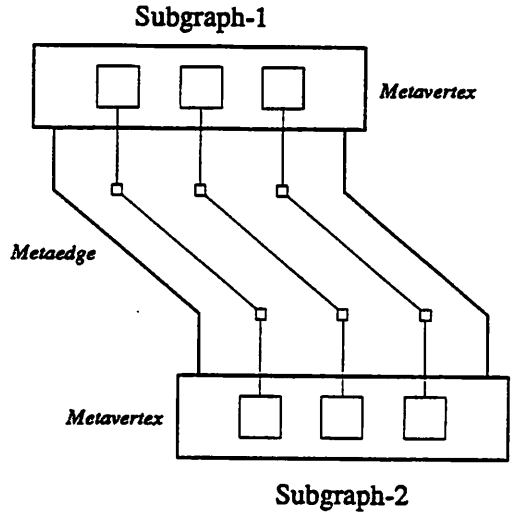


Figure 3-2 — Encoding a Multigraph with Width-Scaled Edges

metagraph layout. Figure 3-3 shows an example. Vertex A has its center placed at the point $(10,25)+(0,40)$, or, $(10,65)$ in the total graph. This location represents the vertex's offset within the subgraph, and the base (lower-left) coordinates of the metavertex within the metagraph, respectively. Similarly, vertex F is centered at $(15,25)+(50,30)$ or $(65,55)$.

The second composition phase merges the individual level structures of the subgraph layouts into a coherent level structure for the layout of the entire graph. In doing so it is often useful to combine nearby levels from horizontally adjacent subgraphs, so as to align more vertices in the total layout, and simultaneously to reduce the number of levels. For example, as shown in figure 3-4 the leftmost figure represents the level misalignment that can result from a proscribed metagraph placement. The rightmost figure shows an adjusted placement that allows the first two levels in each subgraph to be combined. The heuristic for coalescing levels is very simple, merging closely spaced levels that do not overlap horizontally.

Finally, once the vertices have been copied and the level structures merged, the intra-subgraph edges are copied. This copying is a trivial matter, since the edge endpoints (i.e., the vertices) are already fixed in place.

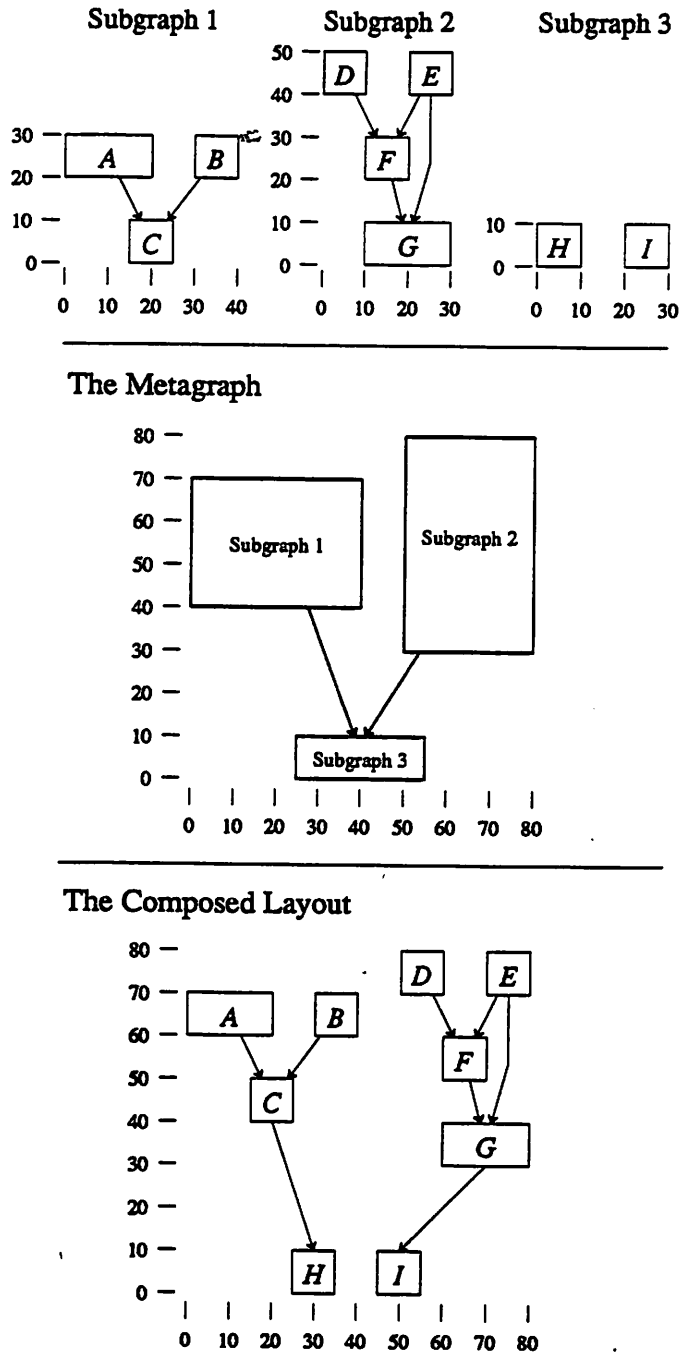
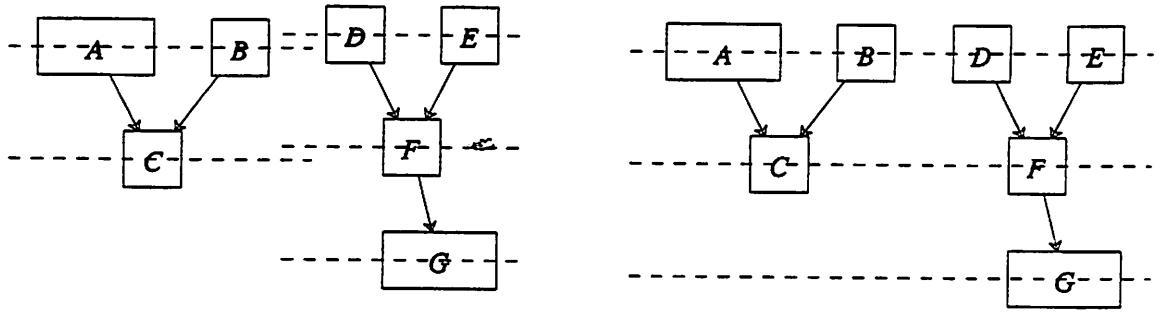


Figure 3-3 — Composing the Vertex Structure



Unaligned Levels

Evenly Aligned Levels

Figure 3-4 — Aligning Subgraph Level Structures

3.3. Interedge Routing

The next step in producing a layout of the total graph is to add the inter-subgraph edges. This task is performed in two steps: 1) inserting the two intra-subgraph segments and 2) inserting the connecting inter-subgraph segment. Figure 3-5 shows this breakdown pictorially.

Although this routing problem is similar in some respects to VLSI routing problems (e.g., the obstacle avoidance routing schemes of Hamachi and Ousterhout [HaO84] and Larson and Li [LaL81]), there are also some key differences. First, these VLSI solutions produce rectilinear routings that may include edges that do not flow monotonically downwards. Second, the algorithms do not necessarily attempt to minimize edge bends. Neither of these problems rule out using a VLSI algorithm for the interedge routing problem, but in the case of *COMPOZE*, a simpler solution was chosen. The rest of this section describes the method used by *COMPOZE*.

The initial method used by *COMPOZE* to add an intra-subgraph segment was to weave the segment into any available space in the corresponding finished subgraph layout. This method proved inadequate, as the subgraph layouts were rather compact, and left little space for later insertions. The current method involves inserting the intra-subgraph segment into the subgraph before it is laid out. This allows the layout algorithm to position the edge. Inserting the intra-segment requires the addition of dummy vertices to the subgraph, along with the connecting edge-segments. The dummy vertex nearest the subgraph border is call a *port vertex*, or simply a *port*. For example, figure 3-6 shows the intra-segment insertions needed to accomodate the inter-subgraph edge (A, C).

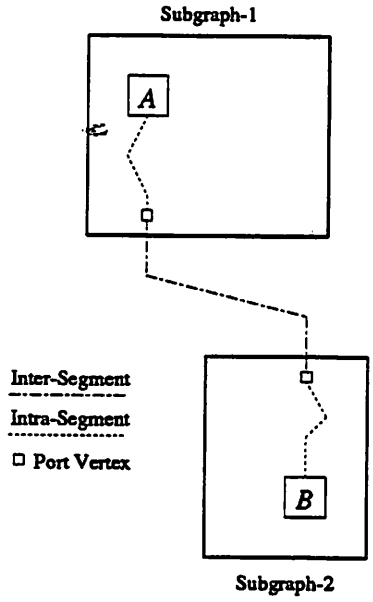


Figure 3-5 — Routing Inter-Subgraph Edges

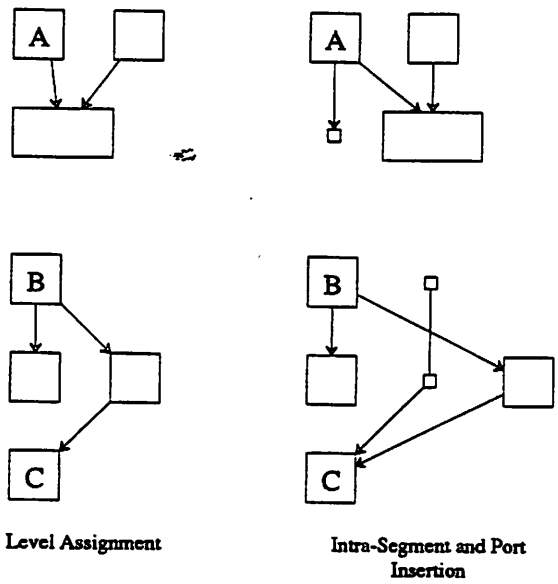


Figure 3-6 — Inserting Ports for the Edge (A, C)

This scheme quickly revealed a new problem: the positions of the ports, relative to the positions of the subgraphs, could cause further problems, as shown in figure 3-7.

The solution to this problem is to bias the placement of the port vertices, and hence the underlying subgraph layouts, using the known positions of the subgraphs. This biasing is accomplished by adding an extra level to the subgraph with *anchored bias vertices*, as shown in figure 3-8. The effect of these anchored bias vertices is to pull the attached port vertices in the desired directions. This, in turn, influences vertices *A* and *B* to swap positions. Note that the ports themselves are not locked in place, as considerations internal to the subgraph layout may outweigh those external to it.

At this point the reader may have noticed a circularity in this method: the subgraph layouts depend on the intra-segments, the intra-segments depend on the metagraph, and the metagraph depends on the subgraph layouts. *COMPOZE* breaks this circularity by estimating the metagraph, and using this estimate as the basis for inserting the intra-segments. Details of this procedure can be found elsewhere [Mes88].

Notice also that certain types of crossings will still not be eliminated by the bias vertex technique. Because the subgraphs are not laid out in any predetermined order (and may in fact

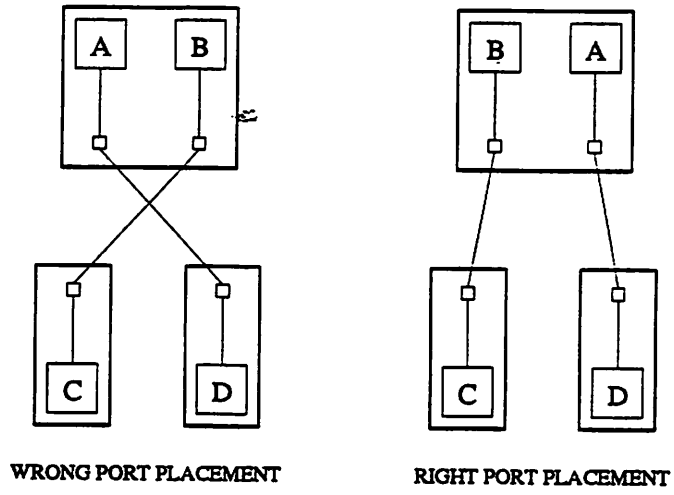


Figure 3-7 — Poor Port Placement Leading to Unnecessary Edge Crossing

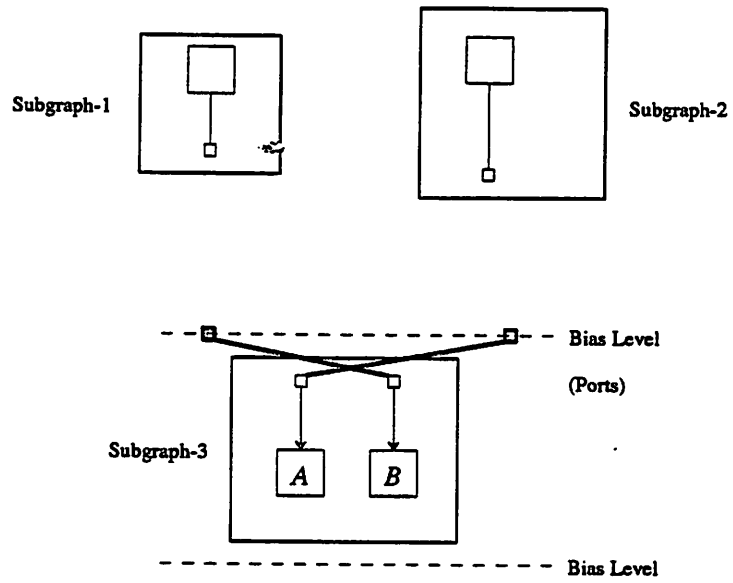


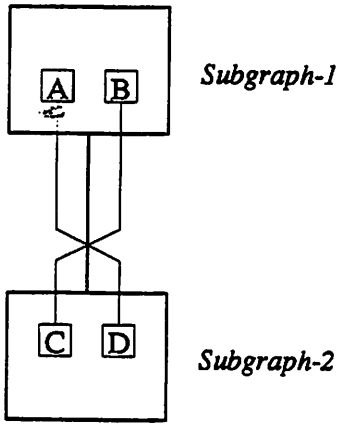
Figure 3-8 — Biasing Port Vertices to Accomodate Inter-Subgraph Connections

be layed out in parallel), a situation such as the one shown in figure 3-9 might occur. In such a case, the bias heuristics would be unable to communicate the relative positions of each pair of vertices. This problem can be avoided if the subgraphs are laid out sequentially and the internal vertex positions are used in biasing connected subgraphs.

Once the intra-segments are inserted, the subgraphs are laid out and composed according to the metagraph template, all that remains is to insert the inter-segments of the inter-subgraph edges. This insertion is accomplished by connecting the appropriate pairs of port vertices. Each inter-segment is routed along the channel cut by its representitive metaedge, as depicted in figure 3-2.

4. Comparative Experiments

This section summarizes the results of experiments conducted to compare the performance of *COMPOZE* and the *GRAB* implementation of the Sugiyama algorithm. The algorithms were compared both objectively, using several layout measurements such as the number of edge crossings, and subjectively, using a panel of expert users. Section 4.1 discusses the graphs used to compare the algorithms, as well as the objective measures that were used. Section 4.2



TYPE 2

Figure 3-9 — Unchecked Intersubgraph Edge Crossing

presents the results of the objective comparison, and section 4.3 summarizes the results of the subjective evaluations.

4.1. Sample Graphs and Objective Measures

The objective experiments examined dual layouts (i.e., one computed by *GRAB* and one computed by *COMPOZE*) of a collection of fourteen graphs, four of which are discussed in this paper. These graphs are characterized in table 4-1.

The graph *shortlist* is a type hierarchy from the *Newspeak* algebraic manipulation system [Fod83]. The graph *world1* is a version of the World Dynamics graph created by Forrester [For71] and used by Sugiyama, Tagawa and Toda to illustrate their algorithm [STT81]. The final two graphs, *grabst2* and *ui2*, are function call-graphs taken from the *GRAB* system. *Grabst2* represents the call-graph of the layout subsystem and *ui2* represents the call-graph of the user-interface.

The objective evaluations compared ten different layout measures, four of which are discussed in this paper. These four measures are:

Graph Name	Vertices	Edges	Graph Type
shortlist	37	63	Type Hierarchy
world1	43	60	Geographic Map
grabst2	169	233	Function call graph
ui2	342	685	Function call graph

Table 4-1 — Graph Set For Statistical Tests

1. *Layout Time.* The number of seconds for the graph to be laid out (not including input-output time) on an unloaded Sun-3/75 workstation with 8-megabytes of main memory and a local swapping disk.
2. *Number of Crossings.* The number of edge crossings.
3. *Total Edge Length.* The sum of the length of all edge segments. The measurement scale is the underlying graphic coordinate system in which vertex-icons have a height of 1.0.
4. *Number of Levels.* The number of distinct graph levels.

These four measure were chosen from the full-set as they seemed to best predict subjective quality. Other measures are analyzed in detail elsewhere [Mes88].

4.2. Objective Experiments

A number of factors must be held constant to compare *GRAB* and *COMPOZE*. First, as each algorithm uses hill-climbing heuristics to minimize edge-crossings, the starting configuration of the input graph can greatly influence the final configuration. A hill-climbing heuristic can be envisioned to be moving along a cost curve, always trying to move from a higher cost configuration (i.e., larger number of crossings) to a lower cost configuration (i.e., smaller number of crossings). When a local minimum is reached, and all of the available transformations lead to higher cost configurations, the algorithm halts. The difficulty in evaluating the performance of this type of heuristic is that the local minimum that is eventually found is a direct product of the starting point on the cost curve, and hence the initial configuration.

Experiments with both *GRAB* and *COMPOZE* show that widely differing final results can be produced by randomly permuting the starting configuration of the graph. Thus, to judge the true nature of each algorithm, and to derive numbers that might usefully be compared, the tests in this section compare the mean statistics over 100 randomly permuted starting configurations

of each graph.

The second factor to be considered is the actual graph partition itself. As discussed in section 3.1, some graphs have application-specific partitions, while others have no such natural partition. In the cases where no semantic partition exists, or where the semantic partition is unimportant or produces especially poor results, a graph-syntactic algorithm such as that of Kernighan and Lin is employed.

In this study (i.e., in which partitioning was not a primary focus of the research), each *COMPOZE* graph is partitioned using a modified k-way version of the Kernighan and Lin algorithm [KeL70]. Each input graph is partitioned into k subgraphs, each of which has a maximum of 30 vertices. This partitioning is handled by recursively dividing large subgraphs in half, until the maximum subgraph size requirement is satisfied. Although it does not produce optimal k-way partitions, the partitions it produces are of sufficient quality for this study. Table 4-2 shows the mean partitioning statistics for each of the four sample graphs. The *Subgraphs* column gives the mean number of subgraphs in a partition. The *Min Subgraph Size* and *Max Subgraph Size* columns give the mean minimum and maximum subgraph sizes, respectively. Finally, the *Intra Edges* and *Inter Edges* give the mean numbers of intra-subgraph and inter-subgraph edges, respectively.

Table 4-3 shows the mean layout time for each of the four sample graphs. Comparing the means, *COMPOZE* reduces layout time by 22% (i.e., *world1*) to 78% (i.e., *ui2*). Overall, *COMPOZE* reduces the layout time by an average of 47%. Notice that the improvements increase

Table 4-2 — Mean Statistics for 100 Partitions of Sample Graphs

Graph Name	# of Vertices	# of Subgraphs	Min Subgraph Size	Max Subgraph Size	# of Intra Edge	# of Inter Edges
shortlist	37	2	18	19	55	7.9
world1	43	2	21	22	51	9.1
grabst2	109	4	27	28	170	66
ui2	342	16	21	22	420	270

with graph size, indicating that the overhead of the compositional process is small in comparison to the polynomially increasing layout time of the underlying Sugiyama, Tagawa and Toda algorithm.

Although *COMPOZE* produces layouts several times more quickly than *GRAB*, it still remains to be seen how the quality of the layouts compare. Table 4-4 shows the comparative edge crossing statistics for the two algorithms. As with the comparison of layout times, *COMPOZE* shows improvement over *GRAB*, with the improvements increasing along with graph

Table 4-3 — Layout Time (Seconds)

Graph Name	*GRAB*				*COMPOZE*				Change
	Min	Mean	SD	Max	Min	Mean	SD	Max	in Mean
shortlist	6.0	6.9	0.4	8.0	3.0	4.2	0.6	6.0	-39%
world1	4.0	5.1	0.3	6.0	3.0	3.9	0.5	5.0	-22%
grabst2	36	42	2.6	49	17	21	1.9	27	-49%
ui2	410	470	26	530	80	100	15	150	-78%
							Avg. Change in Mean		-47%

Table 4-4 — Number of Edge Crossings

Graph Name	*GRAB*				*COMPOZE*				Change
	Min	Mean	SD	Max	Min	Mean	SD	Max	in Mean
shortlist	28	32	4.7	57	39	59	9.1	81	85%
world1	37	54	8.0	77	32	53	15	107	-1.9%
grabst2	1600	1800	190	2300	850	1100	150	1600	-39%
ui2	11000	12000	1000	15000	5800	7000	790	10000	-43%
							Avg. Change in Mean		0.0%

size. Over the original fourteen graph test set *COMPOZE* reduces crossings on eleven by an average of 35%. Across the eight sample graphs with more than 100 vertices this improves to 45%, and across the three graphs with more than 200 vertices, the reduction in crossings is 53%.

The last two measures to be considered reflect aspects of the layout size. First, the total edge length, compared in table 4-5 demonstrates the ability of each algorithm to place connected vertices close together. Over the four graph test set *COMPOZE* produces layouts with an average of 66% smaller total edge length. This reduction is due in large part to the clustered nature of the partitioned layouts. By grouping highly connected vertices into subgraphs, the number of long-edges, and hence the total edge length is reduced.

Secondly, the number of levels in the layouts, compared in table 4-6, demonstrates the ability of each algorithm to organize the vertices into a small number of levels. Across the four graph sample set, *COMPOZE* shows an average increase of 210%. This figure is reduced slightly to 170% across the entire fourteen graph test set.

The enormous increase in the number of levels is produced by two factors. First, new heuristics, implemented to handle variable-height vertex-icons, often introduce extra levels that are used to bend problem edges around icons. These *bend levels* do not contain any input graph vertices and are essentially invisible in the rendered layout. Secondly, the simple strategy used to compose the subgraph layouts does not coalesce as many levels as possible. For example, given two subgraphs, such as the ones shown in figure 4-1a, *COMPOZE* makes no use of the internal structure of the subgraphs to determine their placement. Thus, *COMPOZE* produces

Table 4-5 — Total Edge Length

Graph Name	*GRAB*				*COMPOZE*				Change
	Min	Mean	SD	Max	Min	Mean	SD	Max	in Mean
shortlist	710	1100	290	2300	410	550	120	950	-51%
world1	370	700	250	1400	290	370	36	480	-47%
grabst2	14000	34000	11000	58000	4600	7100	1400	12000	-79%
ui2	190000	400000	79000	610000	37000	53000	8800	83000	-87%
						Avg. Change in Mean			-66%

Table 4-6 — Number of Levels

Graph Name	*GRAB*				*COMPOZE*				Change
	Min	Mean	SD	Max	Min	Mean	SD	Max	in Mean
shortlist	15	15	0.0	15	11	15	0.9	18	2.5%
world1	7.0	7.0	0.0	7.0	8.0	12	1.3	14	71%
grabst2	10	10	0.0	10	20	26	1.7	30	160%
ui2	12	15	1.7	20	95	110	4.7	120	620%
						Avg. Change in Mean			+210%

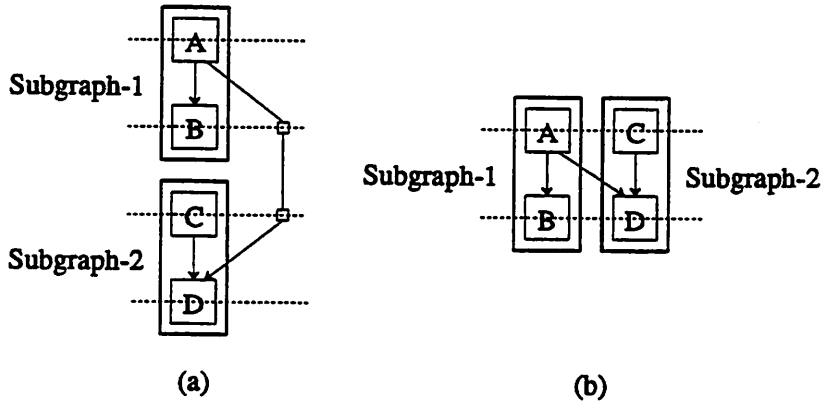


Figure 4-1 — Non-Coalescing Subgraph Levels

the four-level layout of figure 4-1a, instead of the more compact two-level layout of figure 4-1b. This strict adherence to the partition structure causes *COMPOZE* to create many extra levels, as connected subgraphs are not allowed to overlap vertically.

A final measure that can be used to compare the two programs is their actual code size. Both programs are written in *C* [KeR78], and operate in the *SunWindows* environment [SUN85b] . Each system is fairly cleanly divided into a layout module and a user-interface module. Table 4-7 shows the relative sizes of each module in both *GRAB* and *COMPOZE*. The increased size of the layout module is due to the increased complexity of the compositional layout scheme. The increased size of the user-interface module is due to this increased complexity and the less-than-perfect division of the program into two modules. The extra lines of code in the header files define new data structures used by the compositional scheme.

Table 4-7 — Comparative Program Size of *GRAB* and *COMPOZE*

	Lines of *C* Code	
	GRAB	*COMPOZE*
Layout	4,100	7,200
User-Interface	11,000	11,500
Header Files	1,000	1,100

4.3. Subjective User Comparisons

This section briefly summarizes a human factors study that compared layouts produced by *GRAB* and *COMPOZE*. The experiment used dual layouts of seven graphs from the original fourteen used in the objective experiments, including graphs on which each algorithm produced better objective measurements.

The layouts were shown to a panel of six users, three Computer Scientists, and three non-Computer Scientists who used graphs in other disciplines (e.g., planning, document preparation, etc.). In preparing this experiment it was decided that the *GRAB/COMPOZE* browser was not sufficiently powerful to usefully display large graphs, so the layouts were printed on 32-inch wide paper by a Versatec printer.

Each test subject performed their evaluations separately, and did not have any prior knowledge of the particular graphs or layouts. For each pair of layouts, a subject was asked to identify which, if any, of the two layouts was preferred. The subject was then asked to provide a general reason for this selection. Finally, the subject was asked to give specific comments, both positive and negative, about the layouts. Both singular and comparative comments were solicited. Users were encouraged to be especially critical about the layout they choose as better.

Over the seven graph test set the subjects strongly preferred the *COMPOZE* layouts. On only one graph did the subjects prefer the *GRAB* layout, and this preference was predicted by the objective statistics.

The subjects found the *COMPOZE* layouts to be "more balanced" and read as a diagram of related vertices. Most found the clustered nature of the *COMPOZE* layouts to be an aid to understanding the graphs. Particularly, for the large graphs, the clustering allowed the subjects to focus on subsections, without the distraction of the entire graph. One subject noted that the clustering would be misleading if the physical vertex groupings did not represent semantic

groupings. This observation suggests possible conflicts in repartitioning graphs for *COMPOZE*, such as program-call graphs. [6] Although one subject disliked the physical separation of the subgraphs, the remaining subjects all expressed positive opinions about the juxtaposition of semi-dense subgraph layouts with sparse inter-subgraph spaces. That is, they suggested that white-space can be used strategically to improve layout readability.

On the negative side, several subjects pointed out that the large number of parallel edges that flow between subgraphs quickly become impossible to separate visually. Figure 4-2 shows an example of such a layout. Although the edges are physically distinct, they cannot be traced without the use of a pointer (e.g., the subject's finger). Several subjects also pointed out that many edges have unnecessary bends and flow in circuitous routes between subgraphs, also evident in the layout shown in figure 4-2. These effects are primarily a product of the simple meta-graph modeling that is used by *COMPOZE*, and not endemic to a divide-and-conquer layout scheme.

Lastly, as the sample graphs became larger, several problems of scale became evident, most notably overlapping edge labels and vertex labels that do not fit within their icon. These latter problems were equally evident with *GRAB*.

For most of the test graphs the subjects found the *GRAB* layouts to be "disorganized." These problems became more evident as the sample graphs became larger, with subjects noting that the layouts were virtually unusable. Subjects noted that the larger layouts had "edges all over the map," "large, unnecessary spaces" that made it "hard to trace the connectivity" and had in general become "muddled" and "too dense."

When the subjects were queried about the value of specific objective measures, such as edge crossings, they did not seem to feel these layout attributes universally affect graph readability. Instead, they indicated that attributes such as crossings are highly context sensitive, and that edge crossings in sparse layout areas impair readability much less than crossings in dense areas. Similarly, edge crossings appeared to be a problem mainly when their number reaches a certain threshold, relative to other layout attributes. A secondary aspect of edge crossings that the subjects felt affects readability is the angle of crossing. Edges that cross perpendicularly were easier for the subjects to comprehend than edges crossing at acute angles.

Similarly, edge bends, as an objective measure, appeared to have context sensitive importance. The subjects felt that extra edge bends, in an edge running through a sparse area of the layout, did not severly impact readability. On the other hand, the subjects said they found straight edges easier to trace visually. This preference for straight edges appears to be due to a subject's ability to extrapolate the opposite end of a straight edge without having to visually

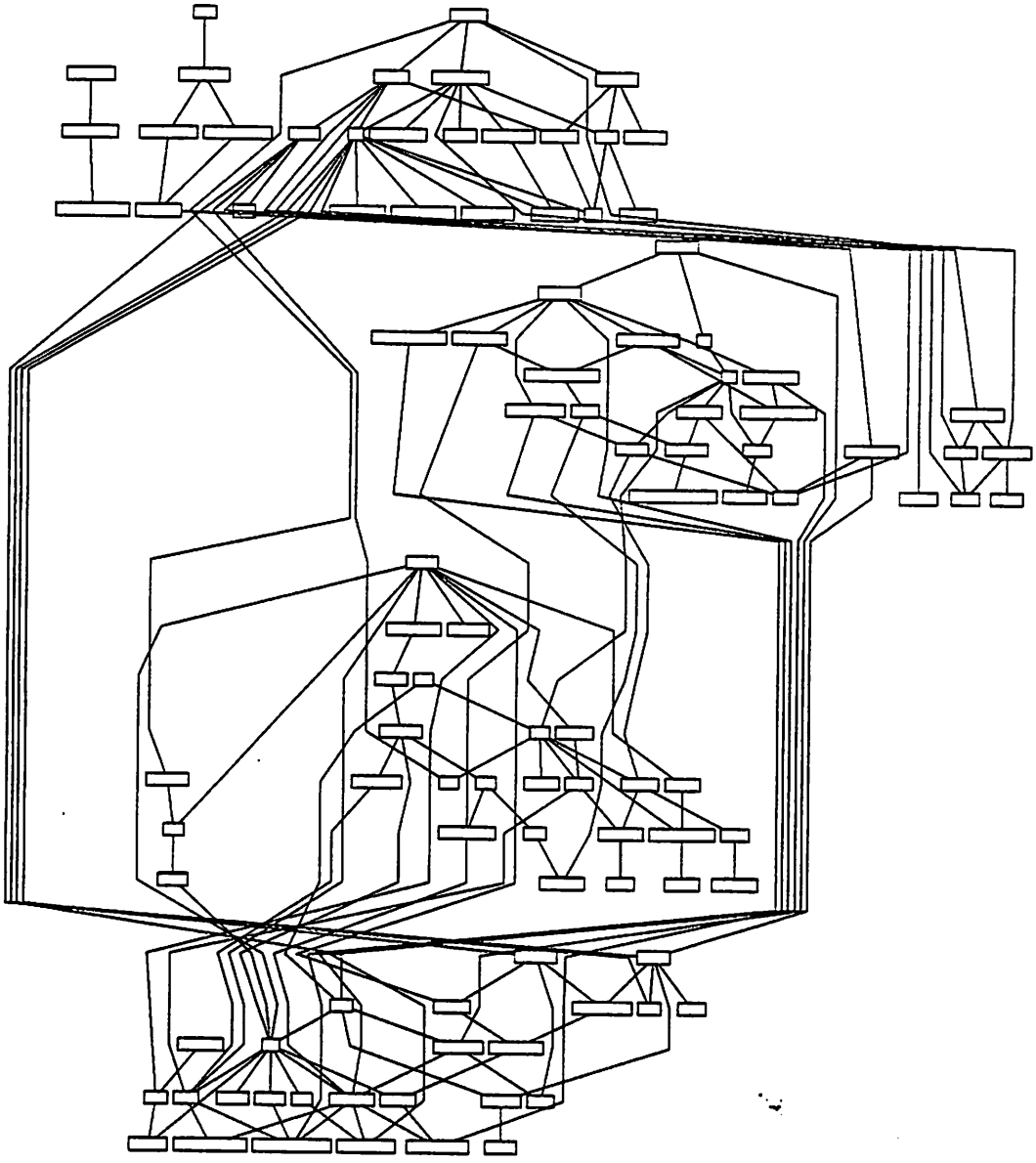


Figure 4-2 — A Sample *COMPOZE* Layout with Merging Parallel Edges

trace its entire length. As noted above, however, this ability is severly hampered when several straight edges are laid close together in parallel.

4.4. Summary

The objective comparisons show clearly that the *COMPOZE* algorithm outperforms the *GRAB* algorithm on a number of layout measures. Most importantly *COMPOZE* is shown to produce graph layouts several times faster than *GRAB*, while simultaneously improving most of the objective measures that were calculated. Thus, from a purely statistical viewpoint, *COMPOZE* provides an example of the promise of divide-and-conquer strategies for graph layout algorithms.

From a subjective perspective, the comparison is not quite as clearly drawn. There is a gap between the magnitudes of the objective statistics and the enthusiasm of the subjective evaluations. Although the objective measures appear to predict the corresponding subjective evaluations, they do so with too large an emphasis, making it unclear whether the correspondence is real or coincidental. What is clear is that the objective measures do not fully capture the subjective measures. The test subjects rarely mentioned notions such as edge crossings or edge length. The actual subjective concerns have yet to be discovered, as has any real link between these concerns and the objective constraints currently in use.

It is easy to see that each objective measure, taken by itself, leads to an improved layout. Thus, if all other layout attributes are held fixed, removing an edge crossing will improve the layout. This hypothetical situation is unfortunately unlikely to occur. In shifting graph elements to alter one layout attribute other attributes will change. These ancillary attributes may be stated layout constraints, or, more likely they will be unstated subjective constraints. What this demonstrates is that the user's perception of a graph is based on both the selected constraints used in computing the layout and the user's own unstated constraints.

5. Conclusions and Future Work

Graphs are a useful notation for relational information. Many applications manipulate relational information either explicitly or implicitly, and would benefit from the ability to display this information to the user in graphic form. As the number of relational applications increases, so do the class and size of the graphs they manipulate. These increases create a need for automatic graph layout algorithms.

The current generation of algorithms generally use one of two approaches to graph layout: 1) general algorithms such as the that of Sugiyama, Tagawa and Toda's system for hierarchical drawings of directed graphs [STT81], or 2) application specific algorithms such as Batini, Talamo and Tamassia's system for entity-relationship diagrams [BTT84a].

Application-specific algorithms produce layouts that most closely reflect the *customary drawing constraints* of the application. For example, a layout algorithm designed to draw finite state automata will automatically follow the conventions of placing the start state on the left, final state(s) on the right, and maximizing edge flow from left to right. A disadvantage of the application-specific approach is that these algorithms rely on special properties of the limited input domain and are thus not easily transferred to new application areas.

General layout algorithms present the opposite balance. The strategies used in these algorithms are applicable to a much wider class of graphs, and can more easily be transferred to new input domains. However, by not utilizing special knowledge about the input domain and the output conventions, the final layouts do not conform as well to the customary drawing constraints of a specific application.

An intermediate approach worth exploring are general algorithms that are easily customized for different applications. Tamassia, Batini and Di Battista refer to this as the need to develop *parametric algorithms* [TBD87].

More research is also needed to understand the gap between the objective layout constraints that are currently in use, and the subjective criteria used in the human interpretation of graph layouts. The current generation of layout algorithms are built on top of an unproven collection of objective constraints.

Most noticeably, as graph size increases, user concerns appear to hinge on a changing set of criteria. Where in a small graph a reduction from 20 edge crossings to 10 might greatly improve readability, a larger graph's reduction from 2000 to 1990 crossings is not likely to have any impact. Further, users note that they interpret most layout attributes in the context of the actual layout. For example, users generally regard edge crossings as unimportant when they are located in sparse areas of a layout. The current use of constraints in a contextual vacuum produces less than optimal results.

More thought also needs to be put into what a graph layout it meant to communicate, and how this is best displayed. Present display technology does not allow large graphs (e.g., 1000's of vertices and edges) to be displayed in their entirety, and so some sort of display/browser interface must be used. This leads again to the dichotomy between general algorithms for wide classes of graphs, and application-specific algorithms for very narrow classes of graphs, with much the same trade-offs as with the layout algorithms themselves. In either case, browsers will have to effectively deal with issues of scale, including powerful navigational functions (e.g., overviews, multiple-views, view-histories) and hierarchical abstractions.

Another interesting problem is providing incremental graph layouts. With this function, a user could specify a *focus vertex*, and the layout system would incrementally add neighbors, ancestors and descendents to the layout. This has the advantage of keeping extraneous information off the screen, and allowing a user to incrementally develop a model of the graph. It appears to be a very difficult problem to solve, as the need to keep graph elements stable between incremental insertions often conflicts with the desired layout aesthetics.

Footnotes

[1] Manuscript received:

[2] Current address: IBM—Almaden Research Center, 650 Harry Road, San Jose, CA 95120-6099.

[3] Current address: Computer Science Division—EECS Department, University of California, Berkeley, CA 94720.

[4] Current address: Computer Science Department, University of Washington, Seattle, WA 98195.

This research was sponsored in part by the U.S. Defense Advanced Research Projects Agency (DoD) under Arpa Order No. 4871 and monitored by the Naval Electronic Systems Command under Contract No. N00039-84-C-0089.

[5] A *GRA*ph *B*rowser.

[6] One subject suggested that secondary cues, such as color, could be employed in repartitioned graphs to designate the original partition. This idea has some interesting applications in studying the logical and physical partitions of graphs.

Bibliography

[SUN85b] —, Programmer's Reference Manual for SunWindows, Sun Microsystems, Inc., Mountain View, CA, April 1985.

[AHY88] K. Andrews, R. R. Henry and W. K. Yamamoto, "Design and Implementation of the UW Illustrated Compiler", Technical Report 88-03-07, March 1988.

[APP84] *MacProject*, Apple Computer, Inc., Cupertino, CA, 1984.

[BTT84a] C. Batini, M. Talamo and R. Tamassia, "Computer Aided Layout of Entity Relationship Diagrams", *Journal of Systems and Software 4* (1984), 163-173.

[BoM76] J. A. Bondy and U. S. R. Murty, *Graph Theory with Applications*, American Elsevier, New York, NY, 1976.

[Car80] M. Carpano, "Automatic Display of Hierarchized Graphs", *IEEE Transactions on Systems, Man, and Cybernetics SMC-10*, 11 (November 1980), 705-715.

[CYN84] N. Chiba, T. Yamanouchi and T. Nishizedi, "Linear Algorithms for Convex Drawings of Planar Graphs", in *Progress in Graph Theory*, J. A. Bondy and U. S. R. Murty (editor), Academic Press, Orlando, FL, 1984, 295-305.

[Dav85] M. B. Davis, "A Layout Algorithm for a Graph Browser", Master's Project Report, Computer Science Division, EECS, UCB, Berkeley, CA, May 1985.

[Del79] M. Delarche, "Quelques Outils Infographiques Pour l'Analyse Structurale de Systemes", Dr. Ing. Thesis, University Grenoble, June 1979.

[EaK86] P. Eades and D. Kelley, "Heuristics for Drawing 2-Layered Graphs", *Ars Combinatoria 21-A* (1986), 89-98.

[Fod83] J. K. Foderaro, "The Design of a Language for Algebraic Computation Systems", Ph.D. Dissertation, Computer Science Division, EECS, UCB, Berkeley, CA, 1983.

[For71] J. W. Forrester, *World Dynamics*, Wright-Allen Press, Cambridge, MA, 1971.

[GNV88] E. R. Gansner, S. C. North and K. P. Vo, "DAG—A Program that Draws Directed Graphs", *Software—Practice & Experience*, (To Appear).

[HaO84] G. T. Hamachi and J. K. Ousterhout, "A Switchbox Router with Obstacle Avoidance", *IEEE 21st Design Automation Conference*, 1984, 173-179.

[Har69] F. Harary, in *Graph Theory*, Addison-Wesley, Reading, PA, 1969.

[HuB85] D. H. Hutchens and V. R. Basili, "System Structure Analysis: Clustering with Data Bindings", *IEEE Transactions on Software Engineering SE-11*, 8 (August 1985), 749-757.

[JAM87] D. S. Johnson, C. R. Aragon, L. A. McGeoch and C. Schevon, Optimization by Simulated Annealing: An Experimental Evaluatio (Part 1), (Unpublished Manuscript).

[KeL70] B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs", *Bell System Technical Journal*, February 1970, 291-307.

[KeR78] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1978.

[KGV83] S. Kirkpatrick, C. D. Gelatt, Jr. and M. P. Vecchi, "Optimization by Simulated Annealing", *Science 220*, 4598 (May 1983), 671-680.

[Knu63] D. E. Knuth, "Computer-Drawn Flowcharts", *Communications of the ACM 6*, 9 (September 1963), 555-563.

[LaL81] R. C. Larson and V. O. K. Li, "Finding Minimum Rectilinear Distance Paths in the Presence of Barriers", *Networks 11*, 3 (1981), 285-304.

[LNS85] R. J. Lipton, S. C. North and J. S. Sandberg, "A Method for Drawing Graphs", *Proceedings 1st Symposium on Computational Geometry*, Baltimore, 1985, 153-160.

[MKF86] M. A. Majewski, F. N. Krull, T. E. Fuhrman and P. J. Ainslie, "Autodraft: Automatic Synthesis of Circuit Schematics", *IEEE International Conference on Computer-Aided Design*, Santa Clara, CA, 1986, 435-438.

[Mak88a] E. Makinen, "Experiments on Drawing 2-Level Hierarchical Graphs", Report A-1988-1, Department of Computer Science, University of Tampere, Tampere, Finland, January 1988.

[Mak88b] E. Makinen, "On Circular Graphs", *International Journal on Computer Math 24* (1988), 29-37.

[Mak88c] E. Makinen, "A Note on the Median Heuristic for Drawing Bipartite Graphs", Report A-1988-4, Department of Computer Science, University of Tampere, Tampere, Finland, May 1988.

[MIM83] M. May, A. Iwainsky and P. Mennecke, "Placement and Routing for Logic Schematics", *Computer Aided Design 15*, 3 (May 1983), 115-122.

Bibliography

[Mes88] E. B. Messinger, "Automatic Layout of Large Directed Graphs", Ph.D. Dissertation, University of Washington, 1988.

[Mey83] C. Meyer, "A Browser for Directed Graphs", Master's Project Report, Computer Science Division, EECS, UCB, Berkeley, CA, December 1983.

[ReT81] E. M. Reingold and J. S. Tilford, "Tidier Drawings of Trees", *IEEE Transactions on Software Engineering SE-7*, 2 (March 1981), 223-228.

[Rob87a] G. Robins, "The ISI Grapher: a Portable Tool for Displaying Graphs Pictorially", *Symboliikka '87*, Helsinki, Finland, August 1987.

[Rob87b] G. Robins, *The ISI Grapher*, Information Sciences Institute, Marina Del Ray, CA, June 1987.

[Rom86] F. I. Romeo, "Probabilistic Hill Climbing Algorithms: Properties and Applications", Master's Project Report, Computer Science Division, EECS, UCB, Berkeley, CA, December 1986.

[RDM87] L. A. Rowe, M. Davis, E. Messinger, C. Meyer, C. Spirakis and A. Tuan, "A Browser for Directed Graphs", *Software—Practice & Experience 17*, 1 (January 1987), 61-76.

[SCC77] A. Sangiovanni-Vincentelli, L. Chen and L. O. Chua, "An Efficient Heuristic Cluster Algorithm for Tearing Large-Scale Networks", *IEEE Transactions on Circuits and Systems CAS-24*, 12 (December 1977).

[STT81] K. Sugiyama, S. Tagawa and M. Toda, "Methods for Visual Understanding of Hierarchical System Structures", *IEEE Transactions on Systems, Man, and Cybernetics SMC-11* (February 1981), 109-125.

[Tam87] R. Tamassia, "On Embedding a Graph in the Grid With the Minimum Number of Bends", *Siam Journal on Computing 16*, 3 (1987).

[TBD87] R. Tamassia, C. Batini and G. Di Battista, "Automatic Graph Drawing and Readability of Diagrams", *IEEE Transactions on Systems, Man, and Cybernetics*, To Appear 1987.

[Tut63] W. T. Tutte, "How to Draw a Graph", *Procedeings of the London Mathematical Society 3rd Series, 13*, 52 (1963), 743-768.

[War76] J. N. Warfield, *Societal Systems*, John Wiley & Sons, New York, 1976.

[War77] J. N. Warfield, "Crossing Theory and Hierarchy Mapping", *IEEE Transactions on Systems, Man, and Cybernetics SMC-7*, 7 (July 1977), 505-523.

[Woo81] D. R. Woods, "Drawing Planar Graphs", Ph.D. Dissertation, Computer Science Department, Stanford University, June 1981.