Graph Layout for Displaying Data Structures

Vance Waddle

IBM Thomas J. Watson Research Center,
P.O.Box 704
Yorktown Heights, NY 10598
waddle@us.ibm.com

Abstract. Displaying a program's data structures as a graph is a valuable addition to debuggers, however, previous papers have not discussed the layout issues specific to displaying data structures. We find that the semantics of data structures may require constraining node and edge path orderings, and that nonhierarchical, leveled graphs are the preferred data structure display. We describe layout problems for data structures, and extend the Sugiyama algorithm to solve them.

1 Introduction

Displaying data structures graphically is a valuable addition to debuggers, especially for large collections of heap allocated data forming trees, graphs, etc. which are naturally displayed as directed graphs. Such a display gives a global view of the data and its structure; without it, the view is limited to a small portion of the data, like the seven blind men examining an elephant.

Previous efforts [12, 13, 16, 19, 24, 25] have explored the architectural and user interface facilities in tools displaying data structures, but have left layout as an architectural feature into which multiple, generic, layout algorithms could be substituted. This leads to the approach in [24] in which each layout algorithm requires a particular type of data structure, e.g., a tree, list, or DAG. However, as a program executes, its the data structures may evolve from one shape to another, and such a layout algorithm may produce bad layouts if given a different structure than it expects. Instead, we add constraints to a general purpose layout algorithm [20] to impose local "layout styles" on the graph.

We are developing a tool to display data structures, which we call DART (for "Data ARTist"). We originally intended to use our implementation of a Sugyiama-style algorithm [23] to lay out data structures with only some "minor tuning." However, when we built a prototype to determine the additional requirements, we discovered problems that were either different than those solved by published algorithms, or were more serious in this context:

1) **Ports - crossings inside nodes**: most layout algorithms were developed to lay out graphs in which the nodes are points. However, when data structures are displayed as in Figure 1 with structure members shown as nested fields, and edges due to pointers end inside the node, edge crossings can now occur inside (and near) the nodes, rather than just between them (Figure 2A, 2C).

J. Marks (Ed.): GD 2000, LNCS 1984, pp. 241–252, 2001.

The graph becomes unreadable much more quickly due to crossings inside the relatively smaller node area than it does for crossings between nodes. [6] calls these "ports", but only solves the problem when the fields are parallel to graph levels. This problem has similarities to the compound graphs of [21], but we are not free to rearrange the node's interior structure.

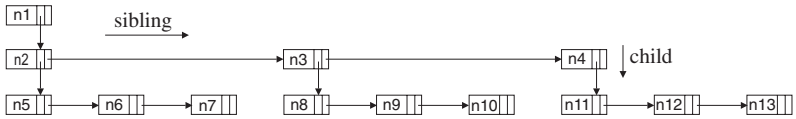


Fig. 1. Knuth-style Drawing of Tree

2) **User control over layout**: Layout algorithms assume more degrees of freedom in laying out the graph than may be allowable in displaying data structures. The two most serious examples uncovered by our prototype were:

i) *Assigning nodes to levels*: The common style of drawing data structures in [10] (Figure 1) uses edges within levels to show pointers to sibling nodes, visually organizing the graph by separating child and sibling links. Without this separation, sibling and child links can become tangled and hard to trace through the diagram. This style requires a consistent direction of edge flow and reduced edge crossings within a nonhierarchical, leveled graph. Orthogonal layout algorithms [3] produce nonhierarchical layouts, but do not preserve the edge flow. Sugiyama-style [20] algorithms reduce edge crossings and preserve edge flow, but their heuristics can not effectively handle edges within a level. Hierarchical layouts are also taller: A balanced M-ary tree (M > 1) of depth K (through *child* links - the tree in Figure 1 has depth of 3) requires max(K, K + (K-2) * (M-1)) levels. For K=10 and M=3, the tree takes 26 levels, or about 3 times as much zooming as a nonhierarchical graph, further degrading readability.

ii) *Reordering nodes and edges*: Layout algorithms reorder nodes and edges to reduce the number of edge crossings. However, the ordering of edges in a data structure may carry important semantic information. Consider a graph representing the expression "A - B", in which "-" is the node root, and "A" and "B" its children. A layout algorithm that reorders child nodes will eventually convert the expression "A - B" to "B - A". Few users appreciate this change.

We provide user control over the layout algorithm by adding constraints to specify the allowable relationships among nodes and edges. We extend the Sugiyama algorithm to process the supported constraints.

3) **Stability between successive drawings**: Debuggers display successive views of a data structure, where each view is slightly modified from its predecessor. Maintaining the stability of the unchanged parts of the display makes it easier to see the changes between views. This is the motivation for incremental layout algorithms [14]. However, it is difficult to create incremental layout algorithms, even without constraints. Although we have an incremental version of the Sugiyama algorithm [20], we considered adding constraints to its already complex processing to be too heroic an effort. Instead, the ordering constraints we required provided a simpler means of imposing stability on the graph.

2 Ports - Crossings Inside Nodes

Algorithms such as [20] are designed to lay out graphs whose nodes have no internal structure. The edges stop at the borders of their end nodes, and barycenters are computed as though the edges end at the node center. When data structures are represented as nodes with nested fields (as in Figures 1 and 2), edges ending at a fixed port inside the node generate crossings inside and around the node of which the algorithm is ignorant. These edge crossings are concentrated in a small area of the graph (Figures 2A and 2C), and quickly becomes crippling problem for data structures containing arrays of pointers like hash tables and B-trees. (We have omitted the field names, which are shown as gray fields in Figure 2.)

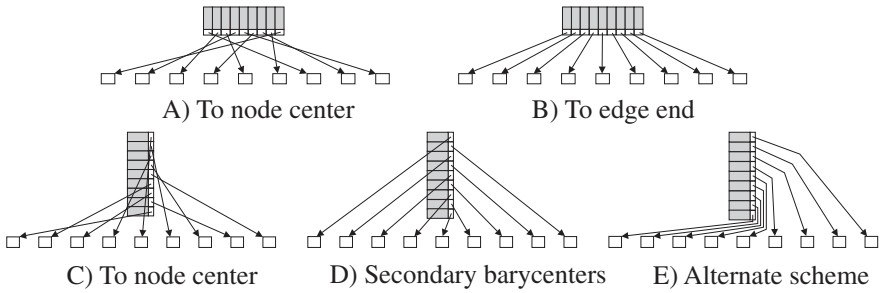


Fig. 2. Fields aligned parallel and perpendicular to levels

This problem divides into two cases: In the simpler case of Figure 2A, the pointer fields are parallel to the graph's levels. This layout could result from a naive application of [20]. This problem can be fixed by using coordinates for the actual size of the node (rather than the ordinal coordinates of [20]), and using the coordinate of the edge's endpoint within the node. It can also be solved by the auxiliary graph method [6], which minimizes total edge length in addition to positioning end nodes. Figure 2B shows the result of these revisions.

Fields stacked perpendicular to the levels have the same horizontal coordinate. Figure 2C shows the problem, and Figure 2D the desired result. ([12, 13, 19] avoid this problem by only displaying the fields parallel to levels.) We use the fact that [4, 20] are based on sorting, and a sort can have both primary and secondary sort keys, with the secondary sort key only significant for comparisons within the same primary key. We define a secondary barycenter, SBc, to produce the graph of Figure 2D. The secondary barycenter is assigned to the child nodes of the compound node. The lowest field goes to the center node to avoid crossing another edge, and edges above it go to nodes on alternate sides to maximize the distance between the edge paths. Given fields $F_0, \ldots, F_k$ in a column, where F_0 is the lowest field, and $i \in \{0, \ldots, k\}$ and $\delta > 0$, then $\mathrm{SBc}(F_i)$ is:

$$\mathrm{SBc}(F_0) = 0$$
$$\text{if } i \text{ is odd then } \mathrm{SBc}(F_i) = \tfrac{-(i+1)}{2}\delta$$
$$\text{if } i \text{ is even and } i \neq 0 \text{ then } \mathrm{SBc}(F_i) = \tfrac{i}{2}\delta$$

This scheme causes edges to left of center child nodes to overlap the field names, which could be objectionable for large graphs. An alternate scheme (Figure 2E) avoids this problem by extra edge routing and longer edges. The parent node is centered over its children, although the graph would be visually improved by shifting it to the left. This is another instance in which traditional layout heuristics (center parent over children, minimize edge lengths) don't apply well. Where $m = \frac{k}{2}$, the secondary barycenters for this scheme are:

$$\mathrm{SBc}(F_m) = 0$$
$$\text{if } i > m \text{ then } \mathrm{SBc}(F_i) = \mathrm{SBc}(F_{i-1}) + \delta$$
$$\text{if } i < m \text{ then } \mathrm{SBc}(F_i) = \mathrm{SBc}(F_{i+1}) - \delta$$

3 User Control via Constrained Layout

Constrained layout schemes [1, 2, 3, 7, 8, 9, 17] typically use a constraint solver to perform layout by solving an optimization problem on some class of equations (linear in [1], linear or quadratic in [7, 8], [9, 17] are rule-based). Our constraints are different because they control the choices made by the layout algorithm within its existing layout style, providing a mechanism to override the default, global style with alternate, local criteria. From the results of our prototype, and surveying drawings of data structures, we determined the minimal set of constraints we needed to support were 1) constraining nodes to the same level, 2) ordering nodes within a level (and ordering edge paths), and 3) forcing edge reversals.

Abstractly, our constraints impose a total order on the layout algorithm's level assignment and crossing reduction phases. Although there are potentially a large variety of layout constraints, only a few are needed to impose a total order on these algorithm phases. The level assignment phase assigns nodes to levels, and reverses cycles. The constraints "node a is on the same level as node b;" "node a is on a level above node b;" and "reverse edge e" are enough to control these decisions. The crossing reduction phase orders nodes within their level, and is controlled by the constraint "node a precedes node b." However, the constraint systems of [1,7] also apply to the algorithm's final positioning phase, which we leave unconstrained. Supporting constraints on the positioning phase might require a more general framework for constraint solving.

In the following, we first give the rationale for each constraint, and then its processing. A constraint on a set of objects S is specified in the layout input by giving a list of the objects in S and a code for the constraint.

Constraining nodes to the same level: The *SameLevel(n, p)* constraint constrains nodes n and p to be on the same level. For convenience, we also provide a constraint that an edge is an in-level edge, which is converted internally into

a *SameLevel* constraint. The edge constraint is the primary constraint that the user specifies.

Processing: The Sugiyama algorithm [20] provides crossing minimization on directed, leveled graphs, but is unable to handle the in-level edges in nonhierarchical, leveled graphs. [18] saw the same need for UML diagrams, and used the Sugiyama algorithm to layout subgraphs not containing in-level edges, then added in-level edges in a separate step. This kept the Sugiyama algorithm from seeing the in-level edges, with the result that crossing minimization was not performed on these edges. We have instead generalized the algorithm to handle in-level edges.

The problem with [20] handling in-level edges is that it sorts nodes in a level based on their barycenters. The barycenter for a node, n, is the average position of a set of nodes, S, related to n through edges. If the nodes in S are in a different level than n, their positions (and hence the barycenter) are constant during the sort. However, introducing in-level edges makes the value of the barycenters change during the sort, since they depend on the position of nodes that are being rearranged by the sort.

Note that this problem exists for the entire Sugiyama algorithm: If one attempted to use the barycenter heuristics without first partitioning the nodes into levels, the same circular dependency would exist of the barycenters on nodes whose positions are changing during the sort. The level structure is not an independent feature of the layout algorithm, but enables the barycenters to function properly. Within each level we create a secondary system of "virtual levels" by performing a topological sort on the level's in-level edges. These virtual levels do not show up directly in the display, but are used to enable barycenter sorting on in-level edges. Abstractly, the process of sorting a level with in-level edges now works by expanding the level's nodes into multiple levels whose structure is dictated by its virtual levels, sorting the resulting sequence of levels, and then re-embedding these temporary levels back to form the original level.

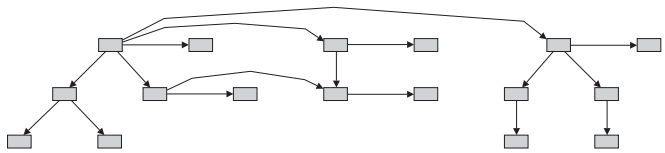


Fig. 3. Nonhierarchical graph showing routing scheme

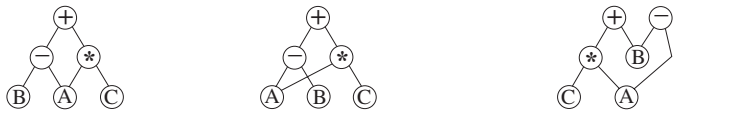
A full description of the extensions to lay out nonhierarchical graphs is beyond the scope of this paper. For a full description, see [22]. Here, we describe processing the *SameLevel* constraint in the level assignment algorithm, after describing the routing scheme for in-level edges. Unless the end nodes of an in-level edge are neighbors in the level, the path of the edge must be routed around the intervening nodes. As shown in the first level of the graph in Figure 3, we route these edges by creating intermediate routing levels between the level containing in-level edges, and its predecessor level.

The algorithm to assign nodes to levels must also be extended to handle *SameLevel* constraints. These constraints are an example of equality constraints, and naturally give rise to equivalence classes. The extended level assignment algorithm first processes the constraints to produce the equivalence classes, i.e., sets of nodes that are on the same level. Each equivalence class is replaced by a single proxy node, and the normal level assignment algorithm is performed to create levels for the graph with proxy nodes. The proxy nodes are then expanded. Finally, for each level containing in-level edges, a variant of the level assignment algorithm is performed to assign the end nodes of in-level edges to the set of virtual levels associated with the level:

1) Create equivalence classes $N_1, \ldots, N_k$, where each N_i is the maximal set of nodes constrained to be on the same level.
2) Replace nodes in each N_i by a proxy node p for N_i. Mark edges between nodes in N_i as in-level edges. For edges e between a node $q \in N_i$ and $r \notin N_i$, replace q in e by p.
3) Perform level assignment on the revised graph, ignoring in-level edges.
4) Replace each proxy node p by nodes in its equivalence class N_i, and replace edges involving p by the original end node.
5) For each level, L, containing in-level edges, perform level assignment on its nodes that are end-nodes of in-level edges to assign them to L's vitual level structure.

Ordering nodes and edge paths: In graphs representing programming language structures like arithmetic expressions, the ordering of nodes and edges can carry important semantic information. Since layout algorithms often reposition the graph's components to achieve aesthetic heuristics (reduce edge-crossings, minimize edge length), we introduce ordering constraints to preserve this information. The constraints then preserve the semantic ordering at the possible cost of degrading the graph's aesthetics.

Consider the "sub-expression reuse graph" for the expression "A-B+A*C" shown in Figure 4A. (This type of graph shows the reuse of variables and subexpressions in a calculation. Each variable occurs once as a leaf node, and arithmetic operations are non-leaf nodes. A node for a subexpression that is reused has multiple parent nodes.) Without constraints, the Sugiyama algorithm changes "A-B" into "B-A" in order to reduce edge crossings and edge lengths. (The semantically correct version in Figure 4B has an additional crossing and a longer edge.)



A) Bad "A-B+A*C" B) Correct "A-B+A*C" C) Bad "C*A+B+A-B"

Fig. 4. Subexpression reuse graphs

Previous efforts have provided constrained layout by incorporating a constraint solver into the layout process, e.g., [1] added linear constraints to the Sugiyama algorithm. This approach is attractive in that it adds a whole class of constraints via a single mechanism. However, when we examined how it would work in practice, we decided it did not solve our problem. [1] imposes an ordering constraint on the sub-expression "A-B" by the linear constraint "A.x<B.x". The graph in Figure 4C illustrates some problems with this scheme:

1) The positions of nodes A and B satisfy the constraint, but the expression A-B has still been reversed because node A has been moved to a lower level. The ordering could be enforced by constraining A and B to be in the same level, but the level assignment in the graph is the one we desire.

2) The constraint we really want is on the intermediate bend between "-" and "A", but the constraint mechanism specifies relationships on objects input to the layout algorithm. The only way to know that there will be a bend is to precompute the layout. This is particularly unattractive.

3) [1] uses the Sugiyama algorithm to generate an initial ordering that is input to the constraint solver as lower priority constraints than the explicit input constraints. This leaves crossing minimization as an artifact of the constraint solver's input, about which it is ignorant. A crossing reduction algorithm that solves constraints should be more effective in reducing edge crossings than separate algorithms.

These problems led us to implement "procedural constraints," in which each constraint has code dedicated to handling it in the layout algorithm. This allows our constraints to be expanded as necessary to apply to objects internal to the layout process like bend points. They may also contain implicit "guard predicates" such that the constraint condition only applies if the predicate is satisfied. (The constraints in pure constraint systems such as [1] always apply.) These properties allow our constraints to circumvent the problem of precomputing the layout. Unfortunately, each new constraint requires added programming.

We provide three constraints to order nodes and edge paths:

1) *NodePrecedes(x, y)* - node x precedes node y only when they are in same level. If they are in different levels, there is no constraint on their relative positions. The condition that the nodes are in the same level handles cases where the nodes have been shifted to different levels. For example, if x and y are children of node z, but y is involved in a cycle that was reversed, y may now be on the level above z. In this case, the ordering constraint may no longer make sense.

2) *PathPrecedes(e, f)* - this constraint orders the bend points and end nodes composing the paths of edges e and f with a common end node. Let e be an edge in a leveled graph between nodes x and y, whose k bend points are in the ordered set B_e. (If there are no bend points, $k = 0$.) Similarly, f is an edge between nodes x and z, with j bend points in the ordered set B_f. We define:

$\text{Path}(e) =$ the ordered set $\{B_e[1], \ldots, B_e[k], y\}$
$\text{Path}(e, i) = B_e[i]$ if $i \in \{1, \ldots, k\}$ else y if $i = k + 1$
$\text{PathLength}(e) = k + 1$

PathPrecedes(e, f) iff for all $i \in \{1, \ldots, \min(k+1, j+1)\}$ then
NodePrecedes(Path(e, i), Path(f, i))
EdgeSuccessor(e, f) iff NodePrecedes(Path$(e, 1)$, Path$(f, 1)$) and
Path$(e, 1)$ is the left neighbor of Path$(f, 1)$

The *PathPrecedes* constraint is converted into *NodePrecedes* constraints when edges spanning multiple levels are converted to a path of short (single-level) edges.

3) *PathPrefixPrecedes(length, e, f)* - this variant of the *PathPrecedes* constraint only applies for the first min(*length*, min(PathLength(e), PathLength(f)) nodes in the path. We use this constraint in our stabilization scheme when the initial part of e's path precedes f's path, but the edges cross.

Processing: After assigning nodes to levels, we convert *PathPrecedes* and *PathPrefixPrecedes* constraints into *NodePrecedes* constraints. Next, for the set of nodes N involved in ordering constraints, we create a precedence graph, $G = <N, E, L>$. For each constraint $NodePrecedes(x, y)$, there is an edge $e = <x, y>$ in E. G is a leveled graph, with the levels in L created by the same style of cycle-breaking topological sort used in the Sugiyama algorithm, except that we do not shorten long edges. The level assignment on G checks consistency by reversing unsatisfiable constraints. G also has the property a node's parents precede it in the constraint ordering, and its children follow it. This gives us a quick check for violations of ordering constraints.

We use the idea of a stable sort to incorporate the constraint ordering into the barycenter sort. Given a node n, let Bc(n) be its barycenter, and Pos(n) its level coordinate. The barycenter sort is stable [11] if given nodes x and y in a level, Bc(x) = Bc(y), and Pos$(x) <$ Pos(y) before the sort, then Pos$(x) <$ Pos(y) after the sort, i.e., the relative ordering of nodes with the same sort key is unchanged. A stable sort places a node n in the proper order if, before the sort:

CO: For all nodes x such that $NodePrecedes(x, n)$ then either Bc$(x) <$ Bc(n) or Bc(x) = Bc(n) and Pos$(x) <$ Pos(n)

Since we have implemented a stable sort, we only need to reorder nodes that fail **CO**. After computing the barycenters of nodes in a level, L, but before the barycenter sort, we perform a presort pass over L's order constrained nodes to find nodes failing condition **CO**. We give these nodes a new position and barycenter to make the sort place them in the proper order.

We check the constraint ordering by traversing the levels in the constraint graph G for the constrained nodes in L, from the root level downward. Any node, n, that violates **CO** with respect to its parent nodes in G will cause the sort to violate the constraint ordering by placing n in front a node it should follow. Let p be the rightmost node in L that is a parent of n in G. We force n into the proper order by placing it after p, and assigning n the barycenter and position of p. However, if several nodes are inserted after the same node, we then have a sequence of nodes with the same position. The comparison in **CO** requires using the position to determine the node order.

Recomputing the positions of all the nodes in the level with each insertion produces an $O(v^2)$ algorithm, where v is the number of nodes in the level. However, the duplicate positions only exist within a sequence of nodes inserted after a common ancestor node in G. We introduce a secondary positioning scheme which we call a collision sequence number. A node's collision sequence number is a secondary sort key, and is only used when two nodes have the same position. With the addition of the collision sequence number, Col, condition **CO** becomes

COS: For all nodes x such that $NodePrecedes(x, n)$ then either $\mathrm{Bc}(x) < \mathrm{Bc}(n)$ or $\mathrm{Bc}(x) = \mathrm{Bc}(n)$ and $\mathrm{Pos}(x) < \mathrm{Pos}(n)$ or $\mathrm{Bc}(x) = \mathrm{Bc}(n)$and $\mathrm{Pos}(x) = \mathrm{Pos}(n)$ and $\mathrm{Col}(x) < \mathrm{Col}(n)$

Initially, all nodes have a collision sequence number of 0. When a node n repositioned to satisfy a constraint is inserted between nodes p and q, $\mathrm{Col}(n) = \mathrm{Col}(p) + 1$. We add 1 to the collision number of the nodes following n with non-zero collision numbers until reaching a node with a 0 collision number, which is not involved in a collision sequence.

This scheme is still $O(v^2)$ in the worst case, but now only within the sequence of nodes that have been repositioned after a common node, and only if nodes get inserted into the middle of the collision sequence. (If nodes are only added to the end of the sequence, there is no iteration.) If this is a problem in practice, we believe it is possible to develop a more sophisticated scheme that could avoid the need for renumbering. The systems of constraints we use to stabilize the layout do not produce this problem. Note that after the sort, the positions of nodes in the level are recomputed to their actual positions. Thus, the collision sequence scheme only exists temporarily, while sorting the level.

Forced edge reversal: Structures like doubly linked lists contain reverse pointers that we want the layout to reverse. However, the layout algorithm may choose to break cycles by reversing the edges representing forward pointers. The constraint $Reverse(e)$ prevents that by explicitly marking the edges to be reversed. This is mainly a convenience, since the tool could implement similar behavior by reversing the edges before giving them to the layout algorithm.

Level above: The constraint $LevelAbove(n, p)$ constrains node n to be placed on a level above the level assigned to node p. The combination of the *LevelAbove* and *SameLevel* constraints can impose a total ordering on node level assignments.

Processing: We form a precedence graph for nodes constrained by *LevelAbove* constraints, whose level structure is input to the level assignment phase as the initial level assignment for the constrained nodes. This forces the resulting level assignments to obey the constraint.

4 Graph Stability Using Ordering Constraints

The minimum goal for an incremental layout algorithm is to preserve the stability [5] of the unchanged portions of a graph that is modified between successive views. Additionally, it may minimize processing costs by reusing the unchanged

parts of the previous layout, thus providing fast lay out on a large graph to which small changes have been made. (See [14] for a fuller discussion.)

Alternatively, stability can be imposed [1] by adding constraints to the layout. This requires the entire graph to be laid out, even for small changes. However, it is easier to implement than an incremental algorithm, particularly since the required constraints were already implemented to preserve "semantic" ordering. Thus, we decided to provide stability via constraints. We add to the layout process the steps: 1) *constraint annotation* - constraints are added to the current layout, before any graph objects are modified, and 2) *modification phase* - nodes and edges are added to, and removed from, the graph to form the next view. The layout cycle is now: layout, constraint annotation, modification, next layout, etc.

1) *Constraint annotation*: We preserve the order of downward edges via *PathPrecedes* constraints when all of an edge's path precedes that of another edge, and *PathPrefixPrecedes* constraints when only a prefix of an edge's path precedes that of another edge. This preserves the inner ordering of subgraphs as they shift between levels. We order the root nodes on the graph's top level to preserve this order if they remain on the top level, even for unrelated subgraphs. Finally, for each edge $e = <r, n>$, where r is a root node, we constrain e to remain in order with its immediate predecessor and successor edges on n. This helps preserve the relative positioning on root nodes as they move in and out of the root level. Given a leveled graph, G, the annotation phase is given by **AddStabilityConstraints**:

```
CreateEdgePrecedes( e, f){
  find depth n to which nodes in e's path precede the nodes in f's path;
  if all nodes in e's path precede all nodes in f's path up an end node
    create constraint PathPrecedes(e, f);
  else
    create constraint PathPrefixPrecedes( n, e, f);
}
AddStabilityConstraints(G){
  for each node, n, on G's top level with a right neighbor p
    create constraint NodePrecedes(n, p);
  for each node n in G /* Constrain downward edges */
    for each edge e from n (in order of the EdgeSuccessor relation), with
                successor edge f, i.e., EdgeSuccess(e,f)
      CreateEdgePrecedes(e, f);
  for each root node r in G
    for each edge e = <r, n>{
      if exists edge f such that EdgeSuccessor(f, e)
        CreateEdgePrecedes(f, e);
      if exists edge F such that EdgeSucessor(e, f)
        CreateEdgePrecedes(e, f);
    }
}
```

2) *Modification phase*: Nodes and edges added to the graph are unconstrained when first laid out. When a node y being removed has constraints $NodePrecedes(x, y)$ and $NodePrecedes(y, z)$, the constraints are replaced by $NodePrecedes(x, z)$, and similarly for edges and $PathPrecedes$ constraints. $PathPrefixPrecedes$ constraints are similarly replaced, but the length of the common path between the edges in the constraint is recalculated.

5 Conclusions

Previous papers on graphically displaying data structures have not examined the layout problems specific to data structures. Data structure displays have requirements and strong preferences that have not been recognized in the graph drawing literature as significant problems. Among these are the "port" problem; the need to display nonhierarchical graphs; and constraining the order of nodes and edge paths in the layout. These are only a minimal set of requirements. While other constraints could be provided, e.g., on node positioning, we do not know which ones are practically important.

In addition to these requirements, many tools displaying data structures will also benefit from stabilizing the unchanged parts of a graph between successive layouts. We have given an algorithm to do this using ordering constraints. Although its processing is not incremental, it is simple to build using node and path ordering constraints.

The algorithms we have described are implemented in the NARC graph tool kit [23], which is DART's engine for graph display and layout. The next stage of work will concentrate on the developing the other mechanisms required in a working tool.

References

1. K.-F. Boehringer, and F. N. Paulisch Using Constraints to Achieve Stability in Automatic Graph Layout Algorithms, *ACM CHI '90 Proceedings*, pp. 43-51.
2. A. Borning, The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory, *ACM Transactions on Programming Languages and Systems*, 3(4), pp. 252-387, 1981.
3. G. D. Battista, P. Eades, R. Tomassia, and I. G. Tollis, *Graph Drawing: Algorithms for the Visualization of Graphs*, Prentice Hall, 1999.
4. P. Eades and D. Kelly, Heuristics for Reducing Crossings in 2-Layered Networks, *Ars Combin.*, 21.A, 89-98, 1986.
5. P. Eades, W. Lai, K. Misue, and K. Sugiyama, Preserving the Mental Map of a Diagram, *Proceedings Compugraphics '91*, pp. 24-33, 1991.
6. 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. 1993.
7. W. He and K. Marriott, Constrained Graph Layout, *Proceedings of Graph Drawing GD'96*, pp. 217-232, Springer, 1996.
8. T. Kamps, J. Kleinz, and J. Read, Constraint-Based Sping-Model for Graph Layout, *Proceedings of Graph Drawing GD '95*, pp. 349-360, Springer.

9. C. Kosak, J. Marks, and S. Shieber, Automating the Layout of Network Diagrams with Specified Visual Organization, *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 24, No. 3, pp. 440-454.
10. D. E. Knuth, *The Art of Computer Programming, Vol 1: Fundamental Algorithms*, Second Edition, Addison-Wesley, 1973.
11. D. E. Knuth, *The Art of Computer Programming, Vol 3: Sorting and Searching*, Addison-Wesley, 1973.
12. J. Korn, A. W. Appel, Traversal-based Visualization of Data Structures, *IEEE Symposium on Information Visualization (InfoVis '98)*, pp 11-18.
13. B. Myers, INCENSE: A System for Displaying Data Structures, *Proc. SIGGRAPH 1983*, pp. 115-125.
14. S.C. North, Incremental Layout in DynaDAG, *Proc. of Graph Drawing GD '95*, pp. 409-418, Springer.
15. S.C. North and E. Koutsofios, Applications of Graph Visualization, *Graphics Interface '94*, pp. 235-245.
16. S.P. Reiss, *The Field Programming Environment: A Friendly Integrated Environment for Learning and Development*, Kluwer, 1995.
17. K. Ryall, J. Marks, and S. Shieber, An Interactive System for Drawing Graphs, *Proc. Graph Drawing GD '96*, pp. 387-393, Springer.
18. Jochem Seeman, Extending the Sugiyama Algorithm for Drawing UML Class Diagrams: Towards Automatic Layout of Object-Oriented Software Diagrams, pp. 415-427, *Proc. Graph Drawing '97*, Giuseppe DiBattista, ed. Springer.
19. T. Shimomura and S. Isoda, Linked-List Visualization for Debugging, *IEEE Software*, Vol. 8, No. 3, pp. 44-51, May 1991.
20. Sugiyama, K., Tagawa, S., and M. Toda, Methods for Visual Understanding of Hierarchical Structures, *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. SMC-11, No. 2. Feb. 1981.
21. K. Sugiyama and K. Misue, Visualization of Structural Information: Automatic Drawing of Compound Digraphs, *IEEE Transactions on Systems, Man, and Cybernetics*, Vol 21, No. 4, pp. 876-892, July/August, 1991.
22. V. Waddle, A Sugiyama-Style Layout Algorithm for Nonhierarchical, Leveled Graphs, in preparation.
23. V. Waddle, and A. Malhotra, An E log E Line Crossing Algorithm for Leveled Graphs, *Proc. of Graph Draw GD '99*, pp. 59-71, Springer.
24. J. Yang, C.A. Shaffer, and L. S. Heath, SWAN: A Data Structure Visualization System, *Proc. of Graph Drawing GD '95*, pp 520-523.
25. A. Zeller and D. Luetkeaus, DDD - A Free Graphical Front-end for UNIX Debuggers, *SIGPLAN Notices*, 31(1):22-27, January 1996.