On-line Hierarchical Graph Drawing

Stephen C. North and Gordon Woodhull

north,gordon@research.att.com
AT&T Labs - Research
180 Park Ave. Bldg. 103
Florham Park, New Jersey 07932-0971 (U.S.A.)

Abstract. We propose a heuristic for dynamic hierarchical graph drawing. Applications include incremental graph browsing and editing, display of dynamic data structures and networks, and browsing large graphs. The heuristic is an on-line interpretation of the static layout algorithm of Sugiyama, Togawa and Toda. It incorporates topological and geometric information with the objective of making layout animations that are incrementally stable and readable through long editing sequences. We measured the performance of a prototype implementation.

1 Introduction

Graph layout is effective for visualizing relationships between objects. Static layout techniques are well understood, but some applications display graphs that change. Examples include:

- interactive graph editors
- displays of intrinsically dynamic graphs, such as Internet router BGP announcements, or data structures in a running program
- browsers for large graphs based on adjustable subgraphs [10]

The browsing application is motivated by the need for better techniques for visualizing massive graphs [14]. For example, a finite state machine for continuous speech recognition can have more than 5×10^6 transitions. Graphs of Internet structures and biological databases can be even larger. Static layout of large general graphs does not seem feasible: even when the layout computation is tractable, it is difficult for a human to make sense of many thousands of objects unless they are arranged in a regular, predictable structure and it does not seem possible to do this for arbitrary graphs. A helpful alternative could be to show the neighborhood around a movable focus node, or a simplification of the base graph, adjusted interactively.

Informative dynamic graph displays should direct attention to changes while also revealing the graph's global structure. When static layout algorithms based on global optimization are employed, insertion or deletion of even one node or edge can dramatically change the layout. Such instable changes disrupt a user's sense of context, and are uninformative because they do not direct attention to changes in structure in the underlying graph [9]. An incremental approach is needed.

Hierarchical drawings are often useful in practice. They seem to provide a good match between visual perception and common data analysis tasks such as identifying

ancestor-descendant relationships or locating articulation points and bridges. Efficient hierarchical layout algorithms have been devised. Dynamic hierarchical drawing potentially has many of the same benefits, and would be useful in many situations where static layout is being employed. We propose a heuristic, Dynadag, that maintains on-line hierarchical graph drawings.

2 Layout server model

Dynadag uses a client-server model with communication based on a shared, managed graph that holds geometric coordinates and other layout attributes. Client and server send changes to each other via *insert*, *modify*, and *delete subgraphs* of the shared graph. The order of these changes is recorded when objects are inserted into subgraphs. The client accumulates changes in the subgraphs and eventually calls the server's *Process* method to obtain a new layout. The server further appends to the *modify subgraph* as it generates a new layout. After the *Process* method finishes, the client may update its display to reflect the new state of the shared graph.

Table 1 lists graph object layout attributes. Coordinates are dimensionless and computed to a client-specified precision (*e.g.* nearest pixel or millimeter). Position attributes are optional in requests. If an insert or modify request does not have a valid position, the server may arbitrarily determine the object's placement. On the other other hand, when the client gives a position (such as when editing a graph interactively, or importing a saved diagram), it is a strong indication to place the object as closely as possible to the request coordinate. Every object also has a flag to request pinning or fixing its position. Thus, some graph objects may be placed manually while others are being managed automatically. Edges can also be given a minimum length and a weight indicating the cost of stretching it. The client further controls the spacing of objects via the separation parameter, which states a minimum horizontal and vertical distance.

The *insert-modify-delete subgraph* mechanism allows the client to make large or small changes to the managed graph. For instance, a client may load an entire external graph into the *insert subgraph* before invoking the server's *Process* method, resulting in a globally optimized layout. Or, a large subgraph may be selected manually and its layout re-computed. At the other extreme, an on-line editor providing direct manipulation may call the *Process* method after every operation.

The change-subgraph interface between clients and servers makes almost no assumptions about how each behaves, and supports multiple layout algorithms. A server is assumed only to make a best effort to process requests and generate a new layout. It is allowed to modify or even ignore requests incompatible with its algorithm. For example, it could align nodes and edges to grid coordinates, or reject non-planar edges or parallel multi-edges. Servers are not responsible for graphical effects such as inbetweening or fading. So animation techniques such as those of of Eades and Friedrich [8] are complementary to our proposal. A significant limitation is that our system does not exploit look-ahead, though doubtless it could produce better off-line animations.

3 Dynadag Heuristic

Most hierarchical graph layout programs use variants of a well-known batch heuristic due to Sugiyama, Tagawa and Toda [16]. This heuristic (STT) draws directed graphs in phases that reduce the search space by solving sub-problems that optimize objectives such as total edge length and crossing number, subject to constraints that edges point downward, nodes not overlap, etc.

The phases of STT are:

1. convert input graph into a directed acyclic graph (DAG) by reversing any cyclic edges
2. assign nodes to discrete levels (ranks), *e.g.* placing root nodes on level 0, their immediate descendants on level 1, etc.
3. convert edges that span multiple levels into chains of model nodes and edges between adjacent levels
4. assign the order of nodes in levels to avoid crossings
5. assign geometric coordinates to nodes and edges, keeping edges (represented as polylines or splines) short and avoiding bends

This ordering of phases prioritizes the aesthetic properties of the resulting layouts. For example, computing a level assignment before determining edge crossings reflects a decision that emphasizing flow is more important than avoiding crossings. Adopting the aesthetic priorities of STT, our dynamic version has the same phases. STT readily lends itself to this modification because each phase depends only on limited information computed by previous phases, and because we can maintain its framework of topological and geometric constraints incrementally as described here.

Dynadag combines the static layout aesthetics of STT and decisions about how to make on-line layouts stable. Measuring how well a dynamic diagram preserves a user's "mental map" is the topic of ongoing research. Without a firm foundation of experimental studies to rely on, we simply assume that basic geometric and topological properties contribute to the a layout's visual stability and readability. Of course, other things being equal, a drawing is more readable when its edges are short and don't have many crossings. These goals often conflict with obvious measures of stability: objects should not move far, and neither the sequence of nodes within a hierarchical level nor the angular order of edges incident on a given node should change much. Because these measures are not comparable, and are handled by different parts of the algorithm, they are not combined into any sort of unified layout quality or stability metrics. We admit that our notion of stability is purely heuristic.

3.1 Main algorithm

Dynadag maintains an internal *model graph* that satisfies the one-level edge constraint of STT phase three, and holds internal information such as the integer rank assignments of nodes. It also stores the model graph's nodes in a two-dimensional array (or *configuration*) for efficient access. Dynadag's *Process* or main work procedure applies the STT phases incrementally. Each phase examines the insert, modify, and delete subgraphs

Value	Type	Explanation
$G = (V, E)$	graph object	graph
$u, v, w, \ldots \in V$	node object	node
$e, f, \ldots \in E$	edge object	edge
$\Delta(G)$	coord	min node separation
$L_{i,j}$	node object	jth node in ith level
r_x,r_y	float	precision
$\lambda(v)$	integer	level (rank) assignment
$X(v), Y(v)$	coord	position of node center
$\hat{X}(v), \hat{Y}(v)$	coord	client node position request
$X'(v), Y'(v)$	coord	previous node position
$B(v)$	coord	node shape bounding box
$fixed(v)$	boolean	node movable
$tail(e), head(e)$	node object	endpoints
$C(e)$	coord list	layout spline
$\hat{C}(e)$	coord list	client request spline
$\omega(e)$	float	weight ≥ 0
$\delta(e)$	float	minimum length ≥ 0
$strong(e)$	boolean	strong level constraint

Table 1. shared graph objects and their attributes

and updates the model graph, configuration, or objects in the shared graph accordingly. Each phase must perform these computations in a way that is stable with respect to the previous layout, while preserving the layout invariants (*e.g.* hierarchical edges point downward). The objectives and constraints for each phase are shown in table 2.

The main steps of $Process$ (algorithm 5) act on the request subgraphs. We will describe each in detail. *Preprocess* conditions the input subgraphs. Some requests trivially fold or cancel, such as an inserting an object and then modifying or deleting it, or modifying the same object multiple times. Likewise, deleting a node implies deleting its incident edges.

Phase	Objective	Constraints
rerankNodes	$\min \sum_{e=(u,v)\in E} w(e)(\lambda(v) - \lambda(u))$	$\lambda(v) \geq \lambda(u) + \delta(u, v)$
reduceCrossings	crossings	$X(v) = X(u) + 1$
updateGeometry	$\min \sum_{e=(u,v)\in E} w(e)\lvert X(v) - X(u)\rvert$	$X(v) \geq X(u) + \Delta(u, v)$

Table 2. objectives and constraints

3.2 Rerank nodes

This phase assigns integer levels to the nodes of the graph to maintain the hierarchy, preserve stability, and minimize total edge length, prioritized in that order. The following discussion assumes the hierarchy is drawn top-to-bottom; of course it is simple to orient the hierarchy in other ways by pre- and post-processing. Level assignment employs an integer network simplex solver previously developed for the *dot* layout program [12]. In applying this solver, Dynadag maintains an auxiliary graph CG_y whose nodes are interpreted as variables and edges as constraints, as shown in tables 3 and 4.

Variable	Explanation
$\forall v \in V : \lambda(v)$	level of v or $Y(v)$
$\forall v \in V : \tau(v)$	stable level assignment of v
$\forall e \in E \vert \neg strong(e) : \rho(e)$	lower endpoint of weak edge
$\lambda_{\min}, \lambda_{\max}$	lowest and highest levels

Table 3. variables in CG_y

Constraint edge	Weight	Explanation
$\forall v \in V : \lambda(v) - \lambda_{\min} \geq 0$	0	maintain min level
$\forall v \in V : \lambda_{\max} - \lambda(v) \geq 0$	0	maintain max level
$\forall e = (u, v) \in E \vert\ strong(e) : \lambda(v) - \lambda(u) \geq \delta(e)$	$\omega(e)$	strong edge constraint
$\forall e = (u, v) \in E \vert \neg strong(e) : \rho(e) - \lambda(u) \geq 0$	$\omega(e)$	weak edge constraints
$\rho(e) - \lambda(v) \geq \delta(e)$	$c_{\text{rev}}\omega(e)$	

Table 4. constraints in CG_y

Dynadag treats client edges as either *strong* or *weak* level assignment constraints. A strong edge is always hierarchical: it points downward so its head is on a higher-numbered level than its tail. A *weak* edge is unconstrained and may point downward, upward or sideways across the same level. To favor hierarchical drawing, weak edges usually have a high cost c_{rev} associated with a non-downward orientation, although a client can explicitly set $\omega(e)$ to be small or zero to defeat this bias. Edges are strong unless the client marks them as weak or pins an endpoint. If the algorithm encounters a cycle, it marks the last inserted edge of the cycle as weak; if the cycle is later broken, this edge will point downward.

The network simplex solver we use does not have any intrinsic stability. To compensate, we add explicit variables and constraints that penalize level assignments by their variance from some given assignment (usually the previous layout or a client-suggested coordinate). Adjusting the penalty edge weights changes the tradeoff between minimizing edge length and maintaining geometric stability.

There are a few additional details to obtaining good level assignments. One is that stability constraints are removed on un-pinned nodes when the client inserts the first incident edge: the previous location of a disconnected, unpinned node is assumed unimportant. Also, Dynadag ensures that nodes with slack in their level assignments are brought up near their parents, by adding non-zero weight constraints with reference to a global anchor node.

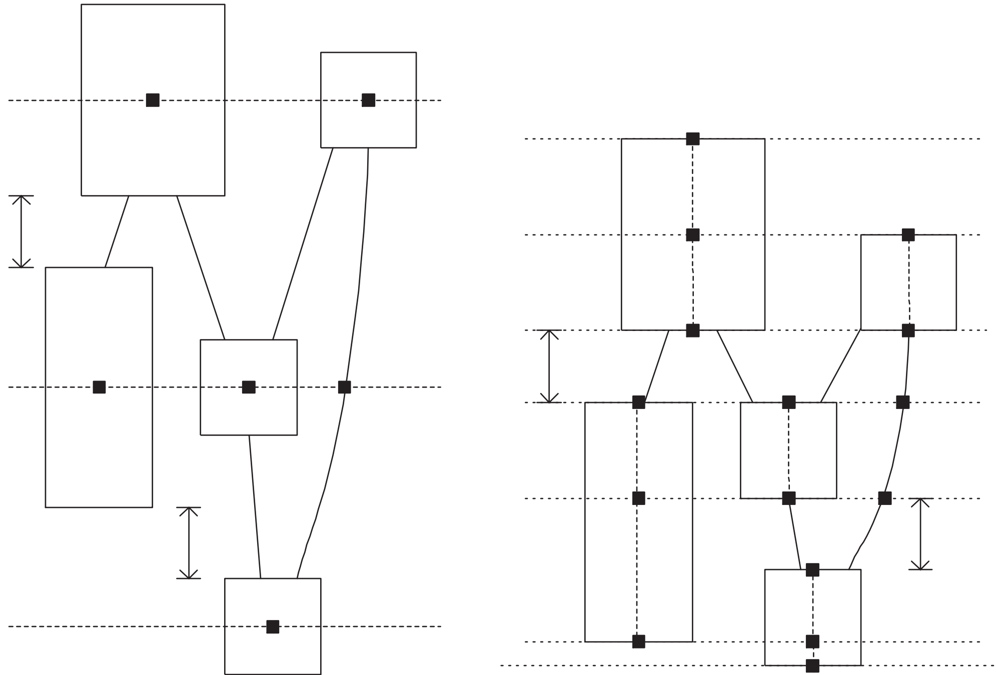

Fig. 1. multiheight ranking system in CG_y

Dynadag supports two ranking systems. The first, familiar from STT, is intended when nodes all have about the same height. In this case, nodes are aligned on ranks, and ranks are separated enough to fit the nodes. The second system better accommodates large differences in node heights by assigning independent levels to the top and bottom of each node. In this case, ranks simply encode Y coordinates and nodes can potentially span many ranks. The second system is more general than the first, but results in larger model graphs, and can allow edges to cross large nodes unless other heuristics are added to the edge router. When nodes are of the same size, the model graph is approximately twice the input graph's size. Both models are of complexity $O(N^2)$ in the size of the input graph.

At the end of this phase, every node is labeled with its new level assignment, $\lambda(v)$.

3.3 Restore Configuration

This phase updates the configuration $L_{i,j}$ according to the new levels $\lambda(v)$. Long edges will be converted into chains of model nodes, all one level apart. We will write the model node chain of $e = (u, v)$ as $u, \phi_0, \phi_1, \ldots, v$. (Self-arcs and flat edges within the same level are ignored in this phase.) When using the multirank node system, nodes that span ranks are also converted to chains. Thus $L_{i,j}$ gives the order j of all nodes and edges appearing in level i.

Dynadag first moves the pre-existing nodes or node chains to match the new ranks assigned in the previous phase. Then it moves edges by moving the chains to the new ranks, lengthening or clipping them as necessary. If the user has specified coordinates for an edge, Dynadag honors the request by placing the model edge for each rank at the X position determined by intersecting the user path with the rank Y. Otherwise it simply draws the chain in a straight line.

3.4 Minimize Crossings in Configuration

At this point, the configuration fully depicts the requested layout. However, edges may be tangled, and in the multirank node system, may even cross nodes.

Dynadag uses a variant of the *dot* crossing minimization heuristic to eliminate node crossings and avoid edge crossings. First it determines which model graph objects are candidates for adjustment. To the nodes and edges corresponding to objects from the *insert* and *modify* shared subgraphs, it adds edges incident on nodes in these subgraphs. (This neighborhood could be extended to try to improve readability at the expense of stability.)

The crossing minimization heuristic scans the configuration and applies two different sorts: *median sort* and *transposition sort*. As it runs, it records the best configuration found so far; if after some number of passes k the configuration has not improved, it restores the best assignment. Its scans alternate between left-to-right and right-to-left, top-to-bottom and bottom-to-top, to avoid built-in bias.

Median sort rearranges nodes according to the median position of incident nodes in the adjacent rank last visited. Transposition sort exchanges adjacent nodes if this reduces the crossing number. As an optimization, transpose sort employs a sifting matrix [5] to avoid re-counting crossings each scan.

On every scan, either the median sort or the transposition sort may reorder nodes even if the crossing number does not decrease. This allows the heuristic to sometimes escape local minima even when no immediate benefit is evident. We have observed that the transposition sort tends to propagate these attempts up or down edge chains in the graph until it eventually eliminates crossings.

It is important to ignore node crossings on the first scan, because nodes must temporarily move across edges to reduce crossings. Weighting node crossings too heavily prevents the transposition sort from trying these steps. Instead, the heuristic first optimizes the model, ignoring whether model edges belong to real nodes. Then it changes the scoring system to penalize edge-node crossings and especially node-node crossings, and scans the graph with the transposition sort to eliminate most node crossings.

It is not always possible to eliminate edge-node crossings in a strictly hierarchical layout with multirank nodes. If any edge-node crossings are left, Dynadag should specially route these edges non-hierarchically in the last phase, but we have not yet implemented this heuristic.

3.5 Update Geometry

This phase computes the coordinates $X(v)$ for model nodes, re-using the integer network simplex solver from step 2. The linear program's variables and constraints are listed in tables 5 and 6 and are represented in an auxiliary graph CG_x.

Variable	Explanation
λ_{left}	the left boundary of the layout
$\forall v \in G : \chi(v)$	X coordinate of node v
$\forall e \in G : \rho(e)$	left point of e
$\forall v \in G : \tau(v)$	stable anchor of v
$S(L_{i,j})$	width of $L_{i,j}$

Table 5. CG_x variables

Variable	Explanation
$\forall i : \chi(L_{i,0}) \geq \lambda_{\mathrm{left}}$	maintain left boundary
$\forall i,j : \chi(L_{i+1,j}) \geq \chi(L_{i,j})) + \Delta_x(G) + \frac{S(L_{i,j})+S(L_{i+1,j})}{2}$	separate adjacent nodes
$\forall v \in G : \tau(v) \geq \lambda_{\mathrm{left}}$	maintain left boundary
$\forall e \in G : \chi(head(e)) \geq \rho(e)$	maintain leftmost node of e
$\forall e \in G : \chi(tail(e)) \geq \rho(e)$	maintain leftmost node of e

Table 6. CG_x constraints

The objective is:

$$\min \sum_{e=(u,v)\in E} c\,\omega(e)(\chi(v) - \rho(e) + \chi(u)) - \rho(e)) + \sum_{v\in V}(1-c)(\chi(v) - \tau(v))$$

which has a term for the total weighted edge length (measured by the L_1 norm), and a term for the total distance that nodes move from certain given positions (previous placements or client-requested positions). The constant c trades off stability and edge length minimization.

After node position assignment, Dynadag recomputes edge routes $C(e)$ where needed. New edges must always be routed and existing edges are re-routed when an endpoint

has moved or the edge has a model node whose distance is less than $\Delta_x(G)$ from a neighbor in the same level. Edges are drawn one at a time by a 2-D spline fitter [7] whose input is a simple path and a list of barrier segments. It returns a piecewise cubic Bezier curve that is close to the path and does not cross any barrier. For Dynadag to provide these arguments to the spline fitter, it takes the model node path of the edge to be drawn, and computes a constraint polygon that contains the path nodes extended horizontally to $\Delta_x(G)$ from neighboring model nodes on the same ranks, ignoring model nodes of edges that cross the one being routed. (Thus, crossings do not appear artificially "forced" to a certain point.) We also compute the shortest path within the constraint polygon, and provide that, along with the constraint polygon as a list of barrier segments, to the spline fitter.

A final detail is the updating of X coordinates of model nodes ϕ_i of an edge e to reflect the intersections of $C(e)$ with the centerlines of levels that it crosses. In other words, model nodes are moved to match the computed spline.

The $UpdateGeometry$ algorithm follows from these details; its listing is omitted to save space.

4 Performance

The asymptotic complexity of the proposed heuristic is dominated by the network simplex algorithm invoked in the first and third phases. Its complexity is $O(IVE)$ per refresh; although I is not provably polynomial, it is often nearly linear in practice. In the second phase, the crossing minimization heuristic is also $O(IVE)$ where I is a small constant that we determine. The edge spline fitter is $O(V^3)$, but often performs quadratically.

We measured the performance of an implementation of the proposed heuristic running on an 1 GHz Intel Pentium PC. The rest graphs were Forrester's World Dynamics graph and the Unix family tree circa 1988, available as `world.dot` and `unix.dot` in the graphviz package from `www.graphviz.org`. To interpret these dynamically, in our experiments we serially inserted nodes (each with its incident edges) until the whole graph was built. Nodes were ordered by breadth-first and depth-first search. We also ran a simple random graph generator. Figures 2 and 3 are animation sequences. Figure 4 shows running time per iteration versus number of graph objects, for each phase. We made full measurements of running time, static layout quality and stability, and expect to report these in a full version of this paper. We noticed that layouts remain readable throughout long editing sequences. This is fortunate, as we had suspected that they could deteriorate so much as to require frequent instable global reoptimization.

The performance of our implementation depends upon the density and complexity of the input graph. For sparse graphs with 95% of insertions being leaves, $Process$ takes less than a second for each of the first 150 insertions. As the graph gets denser, the greater proportion of long edges increases the cost of crossing optimization and X coordinate assignment. With 90% leaves, Dynadag updates graphs of up to 120 nodes in a second. At 80%, graphs inserted incrementally have many crossings, and Dynadag only runs adequately on graphs with no more than 90 nodes.

Since each stage of the algorithm takes time proportional to the model graph size, the batch performance of Dynadag is only slightly worse than the incremental case. This means that when the graph gets too unreadable, re-layout of the graph or part of it is always an option. Although more tuning is needed for interactive use, the prototype is suitable to incremental display of reasonable sized graphs, and to the generation of off-line animations. Improving the asymptotic behavior of our heuristic remains a fascinating goal.

5 Related Work

Newbery-Paulisch and Bolinger proposed augmenting the batch STT algorithm with constraints that preserve the order of nodes staying within the same hierarchical level between successive layouts [3]. This is a good idea, but doesn't preserve placement when nodes change levels. Eades and Sugiyama identified the general problem of stable incremental graph layout and proposed using the global left-to-right scan order of vertices as the stability criterion [9].

In the other main layout families, Tamassia et al and Biedl and Kauffman propose sophisticated incremental algorithms for orthogonal layout [2, 4, 15]. In contrast, force-directed layout algorithms often rely on incremental local search algorithms that can easily drive animated displays [1, 6, 10]. There is a straightforward way of defining additional forces to anchor nodes near intended stable positions, as reported in experiments by Eades and Huang [13].

6 Conclusions

The heuristic has been implemented in an experimental testbed [11, 17] that produced the sequences shown in the figures. Short videos can be seen at `www.research.att.com/~north/videos/gd2001`.

There are several ways the heuristic and its implementation might be improved:

- implement edge labels as model nodes
- consider flat edges in counting crossings
- improve the $ReduceCrossings$ heuristic
- improve sensitivity to the ordering of $moveOldNodes$
- support nested diagrams
- exploit look-ahead in off-line layout
- invent an output-sensitive heuristic

A final remark is that the STT heuristic has proven surprisingly flexible, having accommodated many variants of both static and dynamic layout.

7 Acknowledgments

John Ellson, Emden Gansner, and John Mocenigo shared many ideas with us and made key contributions to our implementations and user interfaces. We also thank the referees of Graph Drawing 2001 for their suggestions.

References

1. Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: algorithms for the visualization of graphs*. Prentice-Hall, 1999.
2. Therese Bield and Michael Kaufmann. Area-efficient static and incremental graph drawings. In *Proc. 5th European Symposium on Algorithms (ESA'97)*, volume 1284 of *Lecture Notes in Computer Science*, pages 37–52. Springer-Verlag, 1997.
3. K. Bohringer and F. Newbery Paulisch. Using constraints to acheive stability in automatic graph layout algorithms. In *Proceedings of ACM CHI'90*, pages 43–51, 1990.
4. S. S. Bridgeman, J. Fanto, A. Garg, R. Tamassia, and L. Vismara. Interactive Giotto: An algorithm for interactive orthogonal graph drawing. In G. Di Battista, editor, *Graph Drawing '97*, volume 1353 of *Lecture Notes in Computer Science*, pages 303–308, Rome, Italy, 1998. Springer-Verlag.
5. R. Schonfeld C. Matuszewski and P. Molitor. Using sifting for k-layer crossing minimization. In Jan Kratochvíl, editor, *Graph Drawing '99*, volume 1731 of *Lecture Notes in Computer Science*, pages 217–224. Springer-Verlag, 2000.
6. J. Cohen. Drawing graphs to convey proximity: an incremental arrangement method. *ACM Trans. on Computer-Human Interfaces*, 4(11):197–229, 1997.
7. D. Dobkin, E. Gansner, E. Koutsofios, and S. North. Implementing a general-purpose edge router. In G. Di Battista, editor, *Graph Drawing '97*, volume 1353 of *Lecture Notes in Computer Science*, Rome, Italy, 1998. Springer-Verlag.
8. P. Eades and C. Friedrich. The Marey graph animation tool demo. In Joe Marks, editor, *Graph Drawing '00*, volume 1984 of *Lecture Notes in Computer Science*, pages 396–406. Springer-Verlag, 2001.
9. P. Eades, W. Lai, K. Misue, and K. Sugiyama. Layout adjustment and the mental map. *Journal of Visual Languages and Computing*, 6:183–210, 1995.
10. Peter Eades, Robert F. Cohen, and Mao Lin Huang. Online animated graph drawing for web navigation. In G. Di Battista, editor, *Graph Drawing '97*, volume 1353 of *Lecture Notes in Computer Science*, Rome, Italy, 1998. Springer-Verlag.
11. J. Ellson and S. North. TclDG - a Tcl extension for dynamic graphs. In *Proc. 4th USENIX Tcl/Tk Workshop*, pages 37–48, 1996.
12. E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo. A technique for drawing directed graphs. *IEEE Trans. Software Engineering*, 19(3):214–230, 1993.
13. M. Huang and P. Eades. A fully animated interactive system for clustering and navigating huge graphs. In Sue H. Whitesides, editor, *Graph Drawing '98*, volume 1547 of *Lecture Notes in Computer Science*, pages 374–383, Montreal, Canada, 1999. Springer-Verlag.
14. Tamara Munzner. *Interactive visualization of large graphs and networks*. PhD thesis, Stanford University, 2000.
15. A. Papakostas, J. M. Six, and I. G. Tollis. Experimental and theoretical results in interactive orthogonal graph drawing. In S.C. North, editor, *Graph Drawing '96*, volume 1190 of *Lecture Notes in Computer Science*, pages 101–112, 1997.
16. K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical systems. *IEEE Trans. on Systems, Man and Cybernetics*, SMC-11(2):109–125, 1981.
17. G. Woodhull and S. North. Montage - an ActiveX container for dynamic interfaces. In *Proc. 2nd USENIX Windows NT Symposium*, 1998.

Algorithm *Process*(inG)
Input: inG: client requests
Output: outG: layout server's updates
($*$ main procedure to process layout requests $*$)
1. outG $\leftarrow$*Preprocess*(inG)
2. outG $\leftarrow$*RerankNodes*$(outG)$
3. outG $\leftarrow$*ReduceCrossings*$(outG)$
4. outG $\leftarrow$*UpdateGeometry*$(outG)$
5. **return** outG

Algorithm *RerankNodes*(inG)
($*$ top level of phase 1- compute new levels $\lambda(v)$. See table 1 $*$).
1. **for** $e \in edgeDeletions(G)$
2.   **if** e is a strong constraint **then**
3.   remove constraint arc representing e in CG_y
4.   **else** remove $\rho(e)$ and incident arcs from CG_y **for**
5.   **for** $v \in nodeDeletions(G)$
6.   remove $\lambda(v)$, $\tau(v)$ and incident arcs in CG_y
7.   **for** $v \in nodeMoveUpdates(G)$
8. $\lambda(v) \leftarrow mapToRank(RequestCoord(v))$
9.   **if** $isAStrongMove(v)$ **then**
10.   **for** e incident on v
11.   remove constraint arc representing e in CG_y
12.   create edge $aux_0 = \tau(v), tail(e)$ with $\omega(aux_0) = c_{\mathrm{rev}}\omega(e)$
13.   create edge $aux_1 = \tau(v), head(e)$ with $\omega(aux_1) = \omega(e)$
14.   stabilize $\lambda(v)$

Algorithm *ReduceCrossings*(M, S)
($*$ reduce crossings on edges incident to nodes in S $*$)
1. pass $\leftarrow$0
2. best $\leftarrow$crossings(M)
3. **while** pass $<$ NPASSES and best $>$ 0
4.   ntrials $\leftarrow$0
5.   **while** pass $<$ NPASSES and ntrials $<$ PATIENCE
6.   leftward $\leftarrow$pass mod 2 == 0
7.   downward $\leftarrow$pass mod 4 $<$ 2
8.   equalPass $\leftarrow$pass mod 8 $<$ 4
9.   BubbleSortPass(S,leftward,HasMedian(downward),MedianCompare(downward))
10.   **while** crossings(M) decreases
11.   BubbleSortPass(S,leftward,downward,true,CrossingsCompare)
12.   current $\leftarrow$crossings(M)
13.   **if** $current < best$ **then**
14.   save configuration
15.   best $\leftarrow$current
16.   ntrials $\leftarrow$0
17.   **else**
18.   ntrials $\leftarrow$ntrials $+1$
19.   **if** $current > best$ **then**
20.   restore configuration

Algorithm *BubbleSortPass*$(S, leftward, downward, comparable, compare)$
1. **for** r in S
2.   **for** u in r according to leftward
3.   **if** not *comparable*(u) **then** continue
4.   **for** v after u in r
5.   **if** not $(u \in S$ or $v \in S)$ **then** break
6.   **if** not *comparable*(v) **then** continue
7.   **if** *compare*(u, v) **then**
8.   put u after v according to leftward

Fig. 2. Unix history graph. The first 48 frames drawn by serially inserting nodes with their incident arcs in breadth-first search order. Each frame is a readable diagram and the sequence appears somewhat stable.

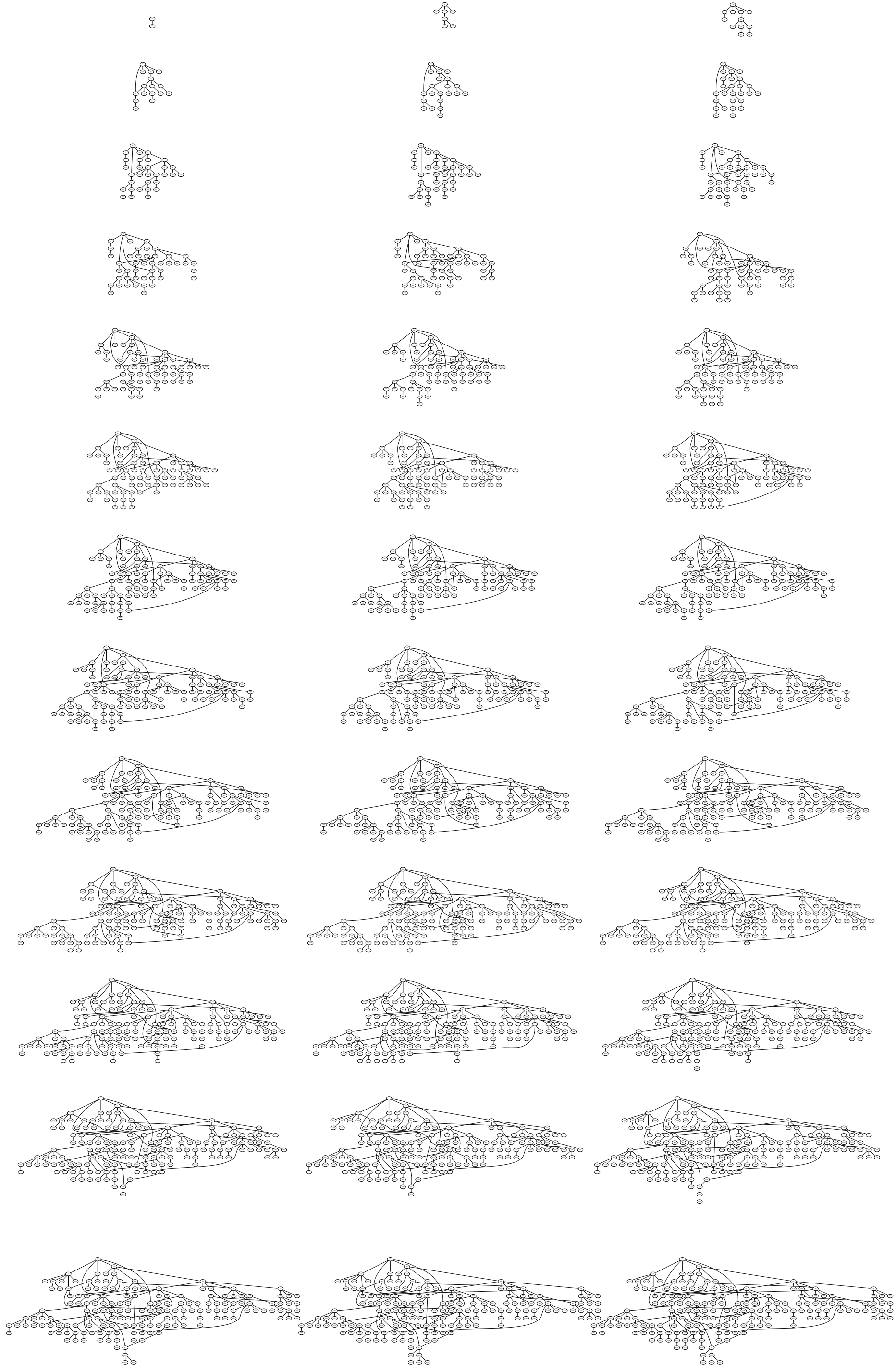

Fig. 3. Sparse random graph, $p(leaf) = 0.90$. Every 5th frame from a sequence of 200 is shown.

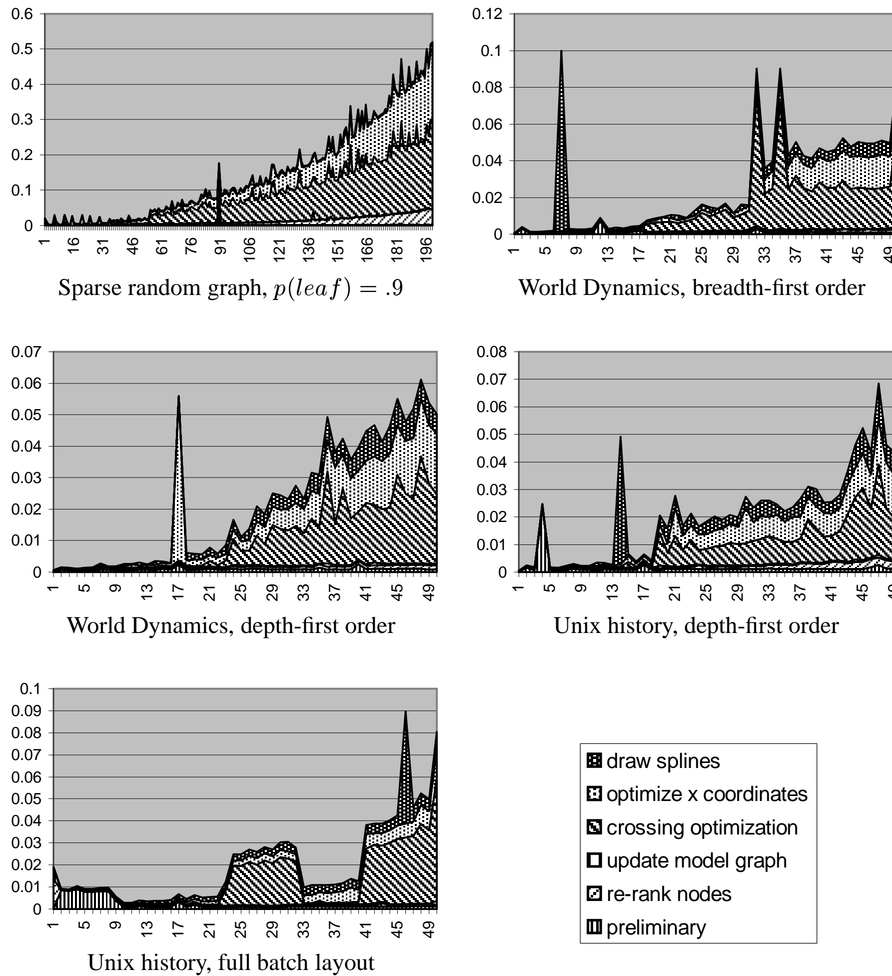

Fig. 4. Time per update in seconds vs. number of nodes for random graphs and two other example graphs. Most updates were less than 0.1 sec. The heuristic was tuned to balance the cost of the different phases.