On-line Hierarchical Graph Drawing

Stephen C. North and Gordon Woodhull

AT&T Labs - Research
180 Park Ave. Bldg. 103
Florham Park, New Jersey 07932-0971 (U.S.A.)

Abstract. We propose a heuristic for maintaining dynamic hierarchical graph layouts. 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 both incrementally stable and readable through long editing sequences. We present measurements of an experimental prototype using generated test data. Applications include incremental graph browsing and editing, the display of dynamic data structures and networks, and browsing large graphs.

1 Introduction

Graph layout is an effective technique for visualizing relationships between objects. Substantial progress has been made on effective static layout algorithms and systems. But some applications need to display graphs whose structure changes over time. Some examples include:

- display of intrinsically dynamic graphs, such as data structures in a running program
- interactive graph editors
- browsers for exploration of large graphs based on views of adjustable subgraphs [ECH98]

The browsing application is motivated by the need for better techniques for visualizing massive graphs [Mun00]. For example, a finite state machine for continuous speech recognition can have more than 5×10^6 transitions, and graphs of the Internet or biological databases can be even larger. Static layout of large general graphs does not seem practical: even if the layout computation is tractable, it is difficult for a human to make sense of many thousands of objects unless they are arranged in some regular, predictable structure and it does not seem possible to do this for arbitrary graphs. For many tasks, a practical alternative could be to display small subgraphs whose contents are adjusted for example, by addition or deletion of nodes near a focus node in the display.

Informative displays of dynamic graphs should focus attention on individual updates, while also revealing the graphs' global structure. When conventional static layout algorithms based on global optimization are employed, the 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 focus attention on specific graph structure changes [ELMS95]. So an incremental approach is needed.

We propose a heuristic, Dynadag, that maintains on-line hierarchical graph drawings. Hierarchical drawings are one of the most useful layout types in practice. They provide a good match between visual perception and some important data analysis tasks such as understanding ancestor-descendent relationships or identifying articulation points and bridges, and efficient hierarchical layout algorithms have been invented. Dynamic hierarchical drawing should potentially have the same benefits, and thus be applicable in many interactive settings where static layout is now employed.

2 Layout server model

Dynadag uses a client-server model, familiar from other visualization tools [BLV95,BBD+99,ALMP95]. The client is usually a graphical user interface, and the server is a library or process that contains layout algorithms.[1] The client and server share an attributed graph that holds the geometric coordinates and other layout attributes of the managed graph. Client and server send changes to each other via *insert*, *modify*, and *delete subgraphs* of this *shared graph*. The client accumulates changes in these subgraphs and eventually calls the server's *Process* method to request a new layout.

To add nodes or edges to the layout, the client creates new graph objects, sets their attributes, then adds them to the *insert subgraph*. Similarly, to change an attribute, the client sets a new value and a flag marking which attribute changed, and adds the object to the *modify subgraph*. The *delete subgraph* works slightly differently: because the client must not destroy a node or edge from the shared graph before the server does its layout, it adds objects to the *delete subgraph* and only after the server has processed changes does it remove the objects from the shared graph. Because changes are cumulative, the server further appends to the contents of the *modify subgraph* as it computes the new layout. For example, adding a new node may involve moving others already in the layout. When the *Process* method has finished, the client updates its display to match all the changes made. (Note that the temporal sequence of updates can be important information; the underlying abstract graph library that we use preserves this by recording the order in which objects are inserted into subgraphs.)

Table 3 shows the attributes associated with graph objects within the server. We will now explain some key details. Position attributes are always optional in requests. If an insert or modify request does not have a valid position, the server may arbitrarily optimize the object's placement. On the other hand when the client specifies a position (as in interactive graph editing or to import a saved diagram), it is a strong recommendation to the server that the object be placed as close as possible to the request coordinate. Every node also has a flag that asks the server to pin the node to a fixed location. In this way, some graph objects may be positioned manually while others are being managed automatically. Coordinates are dimensionless but computed to a client-specified precision, such as the nearest pixel or millimeter. The client specifies the spacing of nodes

[1] In addition to hierarchical layout, our experimental implementation also supports force-directed and incremental orthogonal diagram servers with the same interface.

via the separation parameter, which determines the minimum distance between nodes horizontally and the minimum vertical span of edges.

The *insert-modify-delete subgraph* mechanism allows the client to make large or small changes to the graph using the same interface. For instance, a client may load a whole graph into the *insert subgraph* before invoking the server's *Process* method, resulting in a globally optimized layout. Or a large subgraph may be loaded and its layout re-computed. At the other extreme, an on-line editor supporting direct manipulation will call the *Process* method after every edit. A significant limitation is that our system does not exploit look-ahead, though doubtless it could potentially produce better layouts in off-line situations.

The change-subgraph interface between clients and servers makes almost no assumptions about how each behaves, and is well suited to a variety of layout algorithms. A server is assumed only to make a best effort to process requests and generate a new layout from the previous one. 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. Also, servers are not responsible for graphical effects such as in-betweening or fading. So animation techniques such as those of of Eades and Friedrich [EF00] are complementary to our proposal.

3 Dynadag Heuristic

Most hierarchical graph layout programs use variants of a batch heuristic due to Sugiyama, Tagawa and Toda [STT81]. We will refer to this algorithm as STT. STT draws directed graphs in five main phases. Each phase reduces the search space by solving a subproblem that optimizes properties, such as edge length or crossing number, that affect perceived layout quality. The optimizations involve constraints such as the requirements 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 descendents 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 (drawn as polylines or splines) short and avoiding bends

The precise sequence of phases in STT prioritizes the aesthetic properties corresponding to earlier phases over those of later phases. For example, STT computes the level assignment before determining edge crossings; this reflects a decision that emphasized flow is more important than the crossing avoidance. Adopting the aesthetic priorities of STT, our dynamic interpretation has the same phases as the original algorithm. STT lends itself to this modification because each phase depends only on limited

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

Table 1. shared graph objects and their attributes

information computed by previous phases, and because a server can maintain its framework of topological and geometric constraints incrementally as we will describe.

In addition to these aesthetics, the Dynadag algorithm must reflect some decision about what it means for on-line layouts to be stable. Graph layout metrics and their importance to retaining a stable "mental map" of a dynamic diagram are the subject of ongoing research. Without an extensive foundation of experimental studies to rely on, we will simply assume that first-order 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 if its edges are short and don't have many crossings. These goals are often in conflict with the goals 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 in different arts of the algorithm, it is not necessary to combine them into a unified quality or stability metric.)

3.1 Main algorithm

Dynadag maintains an internal *model graph* which satisfies the one-level edge constraint of phase three and holds internal information such as the integer rank assignments of nodes. It also stores the nodes of the model graph in a two-dimensional array called the *configuration* for efficient access. Dynadag's *Process* or main work procedure, listed in algorithm 5, applies the main phases of the STT algorithm incrementally. Each phase examines the insert, modify, and delete subgraphs 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 $Process$ (lines 4-6) act on the request subgraphs. We next describe each phase of Dynadag 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. In addition, deleting a node implies deleting its incident edges. The Preprocess method handles these cases as they arise, so the input to the rest of Dynadag is consistent and not redundant.

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

Table 2. objectives and constraints

3.2 Rerank nodes

This first 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; the client can easily orient the hierarchy in other directions by pre- and post-processing the coordinates. The level (rank) assignment employs an integer network simplex solver previously developed for the *dot* static layout algorithm [GKNV93]. In applying this solver to update the level assignment, Dynadag maintains an auxiliary graph CG_y whose nodes are interpreted as variables, and edges as constraints as shown in tables 3 and 4.

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

Table 3. variables in CG_y

Dynadag allows client edges to be treated 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,

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

Table 4. constraints in CG_y

weak edges have a high cost c_{rev} associated with a non-downward orientation, but 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.

An issue is that the network simplex solver we employ does not consider stability in computing level assignments. Its process of constructing an initial feasible spanning tree and consecutively exchanging tight (minimum-length) tree edges has no inherent stability. To compensate, we add explicit variables and constraints to penalize level assignments by their variance from some given assignment (the previous layout or a client-suggested coordinate), as shown in figure fig:stabilizer. Adjusting the penalty edge weights determines the tradeoff between minimizing edge length and maintaining geometric stability.

There are some other 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 node's previous location is assumed to be important if it was not connected. 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.

Dynadag supports two ranking systems. The first, familiar from *dot*, is intended when nodes all have about the same height. In this case it aligns nodes on ranks and adjusts separations between ranks to fit all the nodes. The second system 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 scheme is more general than the first, but we retained the first for efficiency and ease of preventing node-edge crossings.

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_G) to reflect the new levels $\lambda(v)$. Recall that the configuration records the sequence within each level of its nodes and the edges that pass through it. To represent the position of edges that pass through a rank, we convert edges between nodes more than one level apart into chains of edges between nodes on consecutive levels. When Dynadag is using the multirank node system, nodes that span ranks are also converted to chains here. After this simplification, edges in L always connect nodes one level apart. We will denote the model node chain of $e = (u, v)$

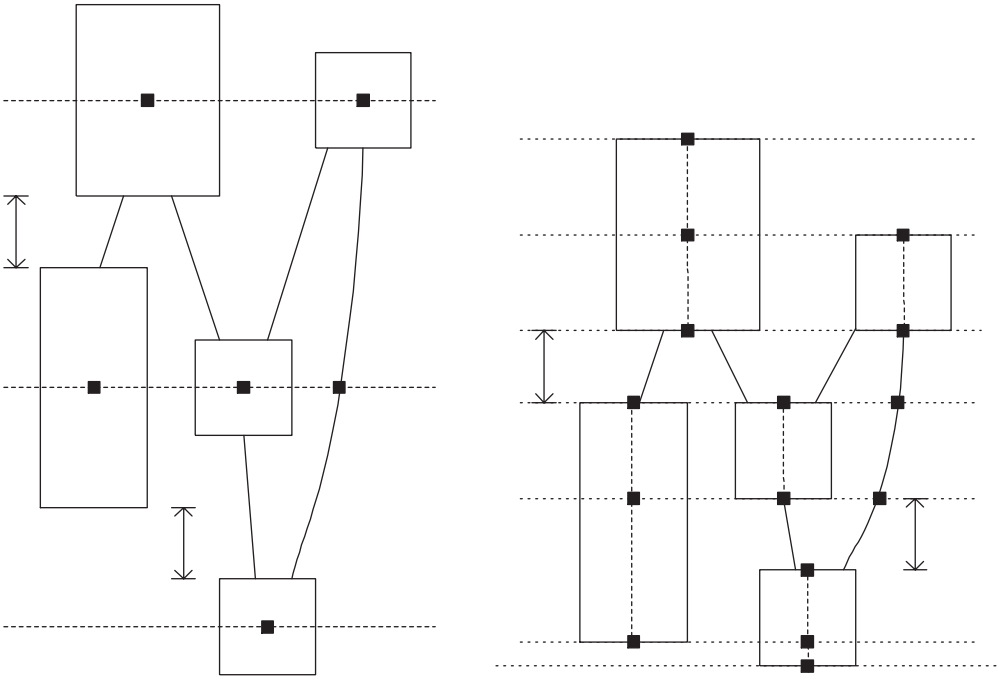


Fig. 1. multiheight ranking system in CG_y

as $u, \phi_0, \phi_1, \ldots, v$. (Self-arcs and flat edges within the same level are ignored in this phase.)

Dynadag first moves the pre-existing nodes or node chains to match the new ranks assigned by the last 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 described in [CMM99] to avoid re-counting crossings each scan.

On every scan, either the median sort or the transposition sort is allowed to 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 changes the number of crossings.

It is important to ignore node crossings on the first scan, because nodes must be temporarily moved 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 which 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 stored in an auxiliary graph CG_x.

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

Table 5. CG_x variables

$\forall i : \chi(L_{i,0}) \geq \lambda_{\text{left}}$	maintain left boundary
$\forall i,j : \chi(L_{i+1,j}) \geq \chi(L_{i,j})) + \Delta(G)_x + \frac{S(L_{i,j})+S(L_{i+1,j})}{2}$	separate adjacent nodes
$\forall v \in G : \tau(v) \geq \lambda_{\text{left}}$	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:

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

Notice the objective has one term for the total weighted edge length (measured by the L_1 norm) and another for the total distance that nodes move from certain fixed positions (their previous placements or client-requested positions). Their ratio c sets the tradeoff between stability and edge length minimization.

Although a new constraint graph may be computed each time Dynadag runs, it is easy to update the previous one in place.

After node position assignment, Dynadag recomputes edge spline routes $C(e)$ as required. A spline is recomputed if either an endpoint moved, or its edge contains a model node whose distance has become less than $\Delta(G)_x$ from a neighbor in the same level. Edge splines are computed by a separate 2-D spline fitter described elsewhere [DGKN98]. Its input is a simple path and a list of barrier segments. It returns an aesthetically pleasing 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 first finds the model node path of the edge being drawn, and computes a constraint polygon that contains the path nodes extended horizontally to $\Delta(G)_x$ from neighboring

model nodes on the same ranks, but ignoring neighbors that are model nodes of edges that cross the one being routed. (In this way, spline crossings do not appear artificially "forced" to a certain point.) Dynadag then computes the shortest path within the constraint polygon, and provides 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 of an edge e to reflect the actual intersections of $C(e)$ with the centerlines of the levels that it crosses.

The $UpdateGeometry$ algorithm follows from these details, so its listing is omitted for brevity.

3.6 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.

Timing measurements were obtained from an experimental implementation of the proposed heuristic. The example graphs are Forrester's World Dynamics graph and the Unix family tree circa 1988, available as `world.dot` and `unix.dot` in the standard graphviz package. To treat these dynamically, we ran experiments in which we incrementally inserted the nodes (and their pertinent edges) one at a time until the whole graph was built. The node ordering was depth-first, breadth-first. We also tried a simply random graph generation method. In the experimental runs we recorded measurements of static layout quality (crossings, edge length, area), stability (node movement, fraction of objects moved) and computation time. The results are in figures 6–6. We noticed that layouts remain readable throughout long editing sequences. This is fortunate, as we had suspected that they might deteriorate so much as to require frequent instable global reoptimization.

In practice, our implementation provides adequate interactive response for graphs of at least several dozen nodes and edges. With much larger graphs, especially ones having many long edges and therefore many model nodes, the proposed heuristic slows down very noticeably. Devising an output-sensitive algorithm whose running time is proportional to the number of objects updated is an important, but so far, elusive goal. Instead, we augmented the heuristic with special-case code for frequently-occurring trivial updates such as leaf node insertion.

4 Related Work

Newbery-Paulisch and Bolinger proposed augmenting a batch hierarchical layout algorithm with constraints that preserve the order of the nodes staying within the same hierarchical level between successive layouts [BP90]. 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 [ELMS95].

In the other main layout families, Tamassia et al and Biedl and Kauffman propose sophisticated incremental algorithms for orthogonal layout [BFG+98,BK97,PST97]. In contrast, force-directed layout algorithms often rely on incremental local search algorithms that can easily drive animated displays [Coh97,BETT99,ECH98]. Thus there is a straightforward way of defining additional forces to anchor nodes near intended stable positions, as reported in experiments by Eades and Huang [HE98].

5 Conclusions

The heuristic has been implemented in an experimental testbed for studying dynamic layout algorithms and applications [EN96,WN98]. It produced the sequence shown in figure 4. A demonstration video can be seen at `http://www.research.att.com/areas/visualization/videos/...`

There are several ways the heuristic could be improved. Implementing edge labels as model nodes is a straightforward extension. Considering flat edges in counting crossings also yields better layouts. Also, as with the original STT heuristic, the $UpdateOrdering$ heuristic is an especially promising area for exploration and improvement. For example it would be interesting to experiment with the sensitivity of the heuristic to the ordering of nodes in $moveOldNodes$.

Another desired extension is to support nested diagrams with provision for routing external edges connecting nodes in different diagrams. We envision a design with a layout manager per nested diagram, and a global manager to transmit updates between them.

Off-line layout with look-ahead is a very promising idea; knowledge about the future ought to be quite valuable in creating good animations.

In a broader context, many basic questions about dynamic graph visualization deserve further investigation. The importance of animation to information visualization is an active research topic. Even in the domain of graph layout, what geometric and topological properties contribute most toward a user's retaining context in dynamic layouts? What tradeoffs between readability and stability are best in practice? How can these properties be provided by efficient algorithms?

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

6 Acknowledgments

Emden Gansner and John Ellson had many interesting conversations with us this work, and made key contributions to our implementations and user interfaces. John Mocenigo wrote Grappa, a Java graph interface that we also use for experiments.

References

[ALMP95] N. Amenta, S Levy, T. Munzner, and M Phillips. Geomview: a system for geometric visualization, 1995.

[BBD+99] Gill Barequet, Stina S. Bridgeman, Christian A. Duncan, Michael T. Goodrich, and Roberto Tamassia. Geomnet: Geometric computing over the internet. Technical Report http://www.cs.jhu.edu/ barequet/gmn/gmn-ic.html, Center for Geometric Computing, 1999.

[BETT99] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: algorithms for the visualization of graphs*. Prentice-Hall, 1999.

[BFG+98] 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.

[BK97] 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.

[BLV95] G. Di Battista, G. Liotta, and F. Vargiu. Diagram server. *Journal of Visual Languages and Computing*, 6(3):275–298, September 1995.

[BP90] 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.

[CMM99] R. Schonfeld C. Matuszewski and P. Molitor. Using sifting for k-layer crossing minimization. In *Graph Drawing*, pages 217–224, 1999.

[Coh97] J. Cohen. Drawing graphs to convey proximity: an incremental arrangement method. *ACM Transactions on Computer-Human Interfaces*, 4(11):197–229, 1997.

[DGKN98] 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.

[ECH98] 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.

[EF00] P. Eades and C. Friedrich. The marey graph animation tool demo. In *Graph Drawing*, pages 396–406, 2000.

[ELMS95] 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.

[EN96] J. Ellson and S. North. TclDG - a Tcl extension for dynamic graphs. In *Proc. 4th USENIX Tcl/Tk Workshop*, pages 37–48, 1996.

[GKNV93] 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.

[HE98] M. Huang and P. Eades. A fully animated interactive system for clustering and navigating huge graphs. In *Graph Drawing*, pages 374–383, 1998.

[Mun00] Tamara Munzner. *Interactive visualization of large graphs and networks*. PhD thesis, Stanford University, 2000.

[PST97] 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.

[STT81] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical systems. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-11(2):109–125, 1981.

[WN98] 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 ←*Preprocess*(inG)
2. outG ←*RerankNodes*($outG$)
3. outG ←*UpdateOrdering*($outG$)
4. outG ←*UpdateGeometry*($outG$)
5. **return** outG

Algorithm *RerankNodes*(inG)
($*$ top level of phase 1- compute new levels $\lambda(v)$. see table XYZ $*$)
1. ($*$ deletions $*$) **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. ($*$ move old nodes $*$) **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 arc $aux_0 = \tau(v), tail(e)$ with $\omega(aux_0) = \mathrm{BackwardEdgeCost}$
13. create arc $aux_1 = \tau(v), head(e)$ with $\omega(aux_1) = \mathrm{ForwardEdgeCost}$
14. ($*$ stabilize $\lambda(v)$ $*$)
15.

Algorithm *OptimizeCrossings*(M, S)
($*$ reduce crossings on edges incident to nodes in S $*$)
1. pass ←0
2. best ←crossings(M)
3. **while** pass $<$ NPASSES and best $>$ 0
4. tired ←0
5. **while** pass $<$ NPASSES and tired $<$ PATIENCE
6. leftward ←pass mod 2 == 0
7. downward ←pass mod 4 $<$ 2
8. equalPass ←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 ←crossings(M)
13. **if** $current < best$ **then**
14. save configuration
15. best ←current
16. tired ←0

```
17.                 else
18.                         tired ←tired +1
19.         if current > best then
20.                 restore configuration
21.
```

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 ∈ S or v ∈ S) then break
6.                if not comparable(v) then continue
7.                if compare(u, v) then
8.                        put u after v according to leftward
9.
```

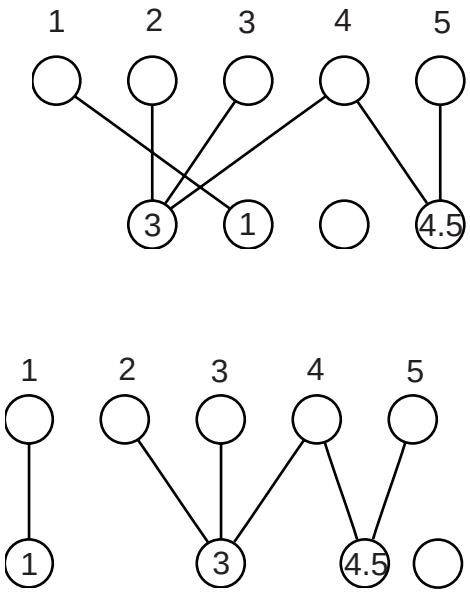


Fig. 2. median sort example. The top rank is fixed.

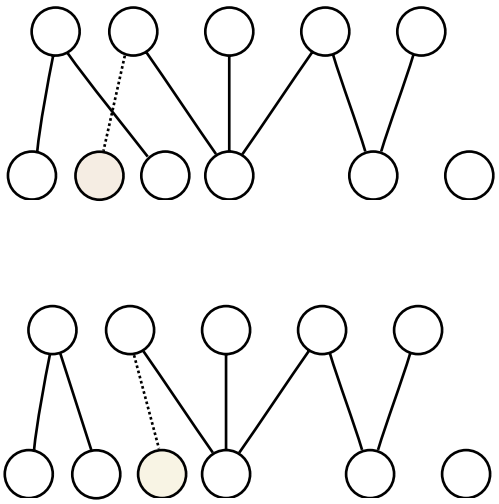
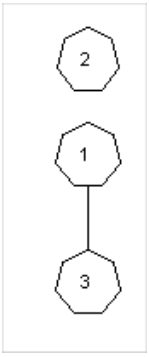
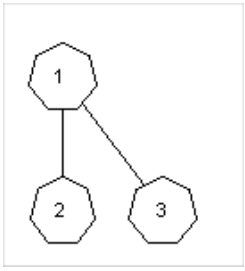


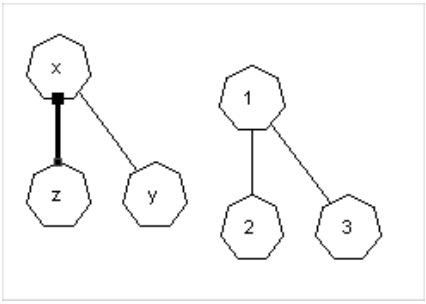
Fig. 3. place_and_reopt example. A node from the top rank is copied to the bottom one, then re-sorted.



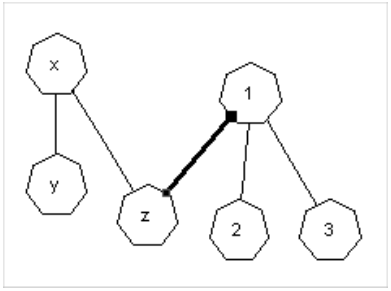
(a) Initial graph



(b) After inserting $1 \rightarrow 2$



(c) After inserting new subgraph on left



(d) After inserting $1 \rightarrow z$

Fig. 4. example animation sequence. The multirank node option is enabled.

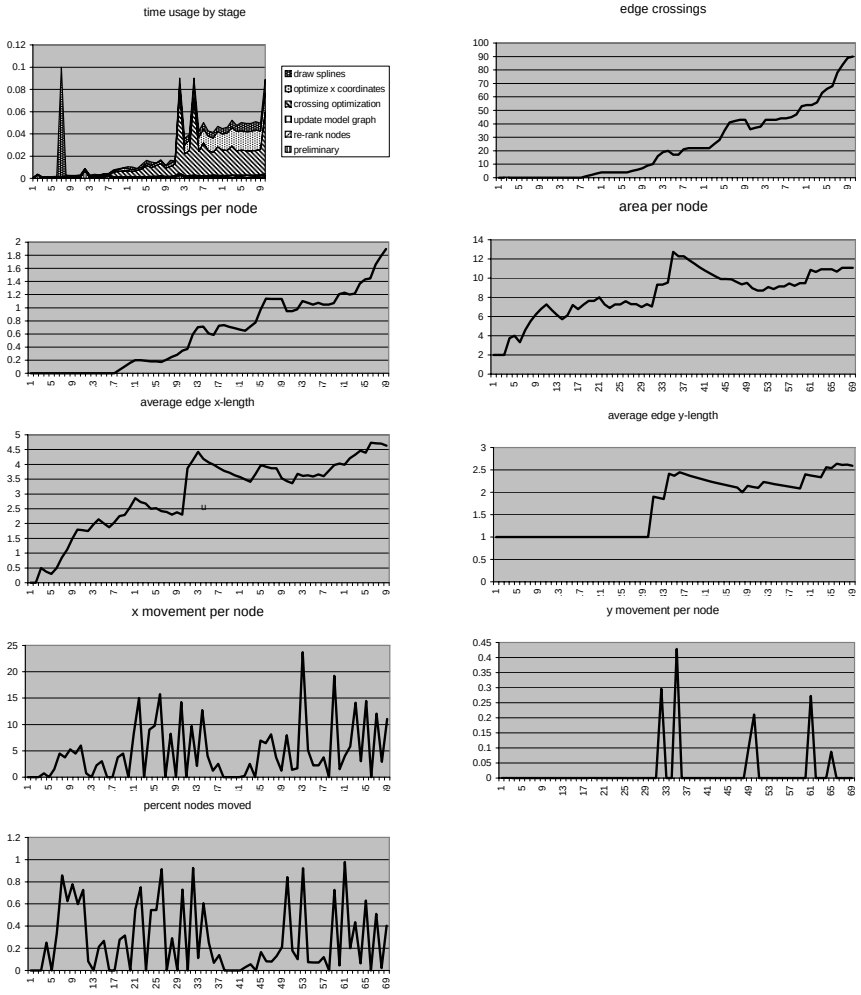


Fig. 5. world.dot – bfs

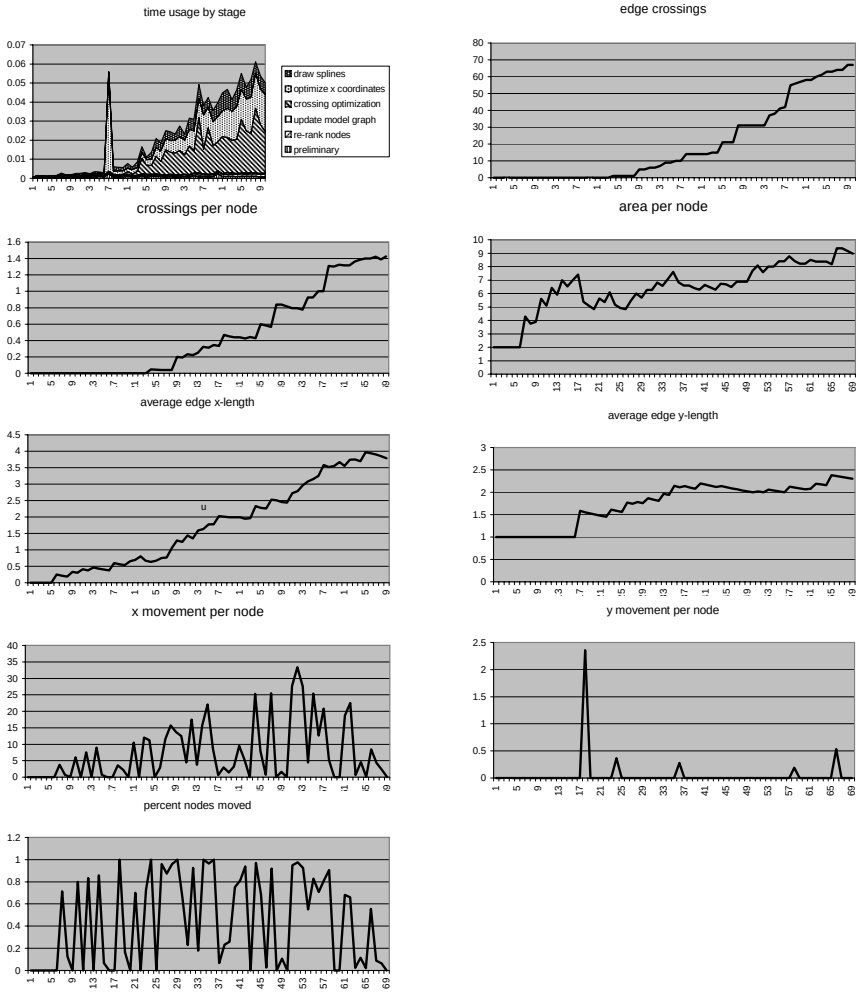


Fig. 6. world.dot – dfs

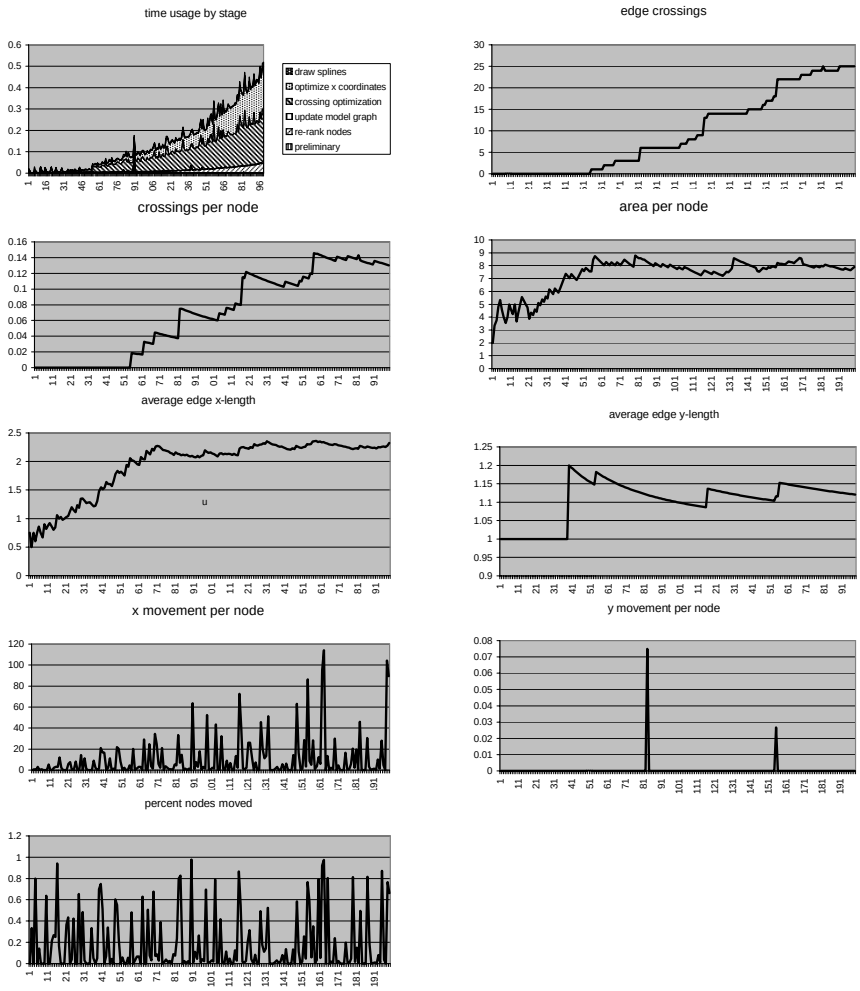


Fig. 7. random graph generated by incrementally connecting a new leaf node with $p = .95$ or adding an edge between two nodes selected at random with $p = .05$.

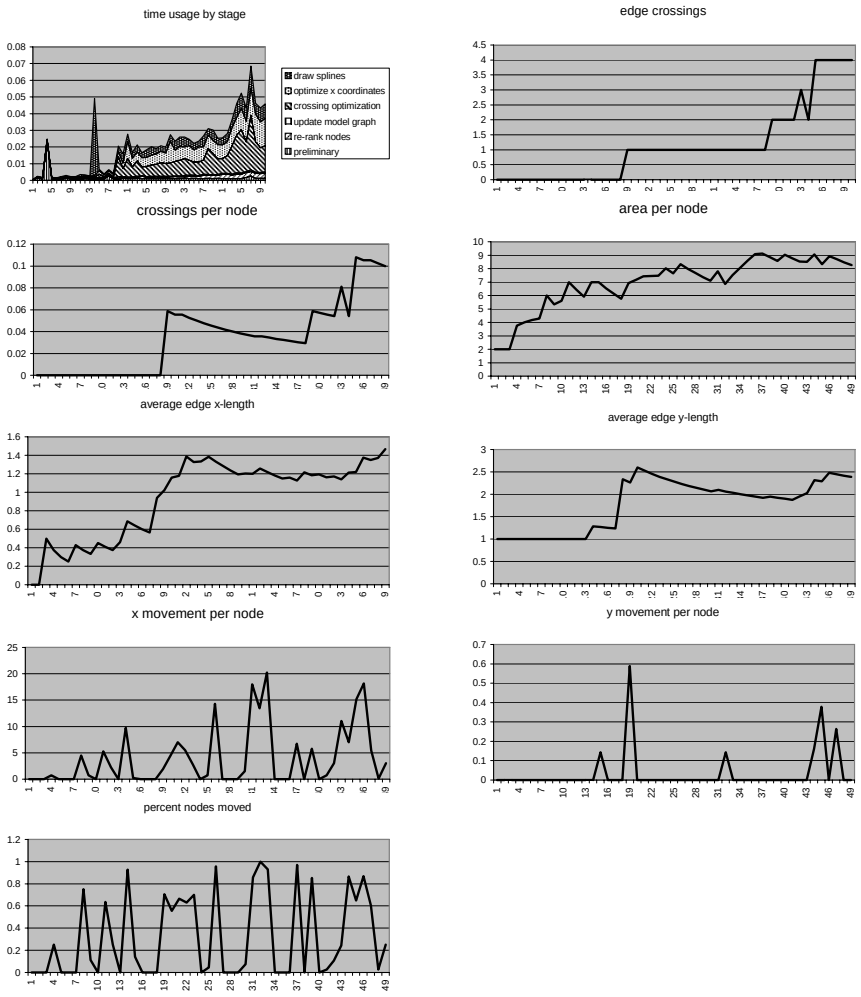


Fig. 8. unix – dfs

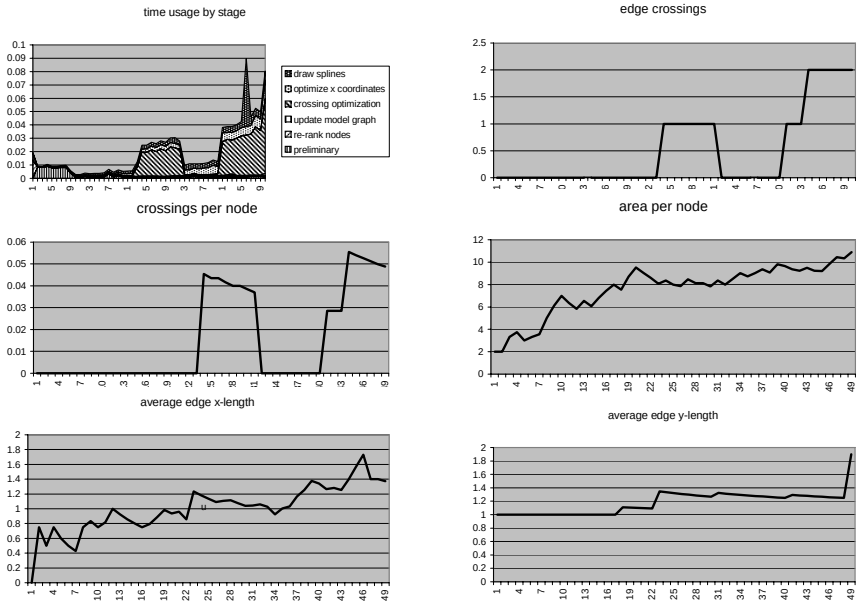


Fig. 9. unix.dot – random insertion, batch layout

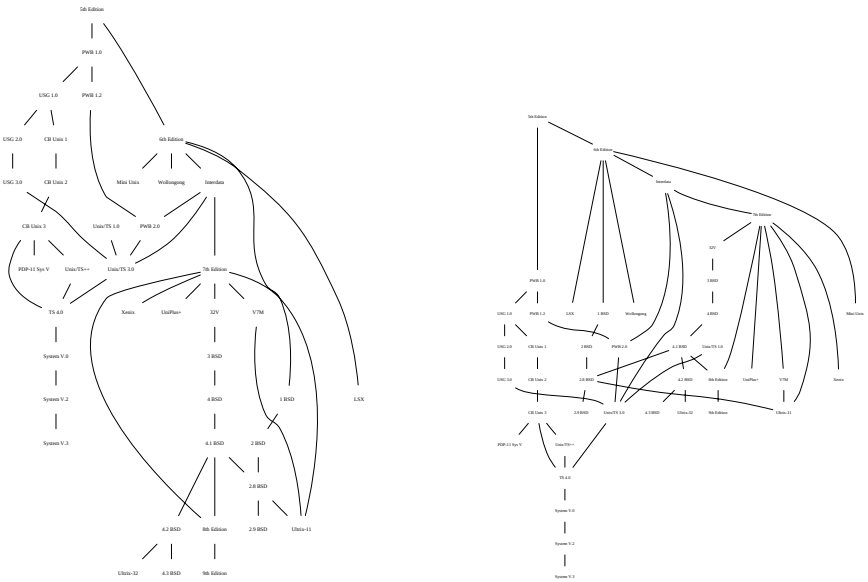


Fig. 10. unix graph, final dfs and bfs drawings