Incremental Layout in DynaDAG

Stephen C. North
north@research.att.com

Software and Systems Research Center
AT&T Bell Laboratories
Murray Hill, N.J. 07974 U.S.A.

Abstract. Generating incrementally stable layouts is important for visualizing dynamic graphs in many applications. This paper describes *DynaDAG*, a new heuristic for incremental layout of directed acyclic graphs drawn as hierarchies, and its application in the *DynaGraph* system.

1 Introduction

Effective techniques have been developed for some important families of graph layouts, such as hierarchies, planar embeddings, orthogonal grids and forced-directed (spring) models [1]. These techniques have been incorporated in practical user interfaces that display static diagrams of relationships between objects [19, 18, 17].

Static diagrams are not completely satisfactory because in many situations, the displayed graphs can change. Three common scenarios are:

Manual editing. Most interactive graph drawing systems allow users to manually insert and delete nodes and edges. Layouts must be updated dynamically to reflect such changes.

Browsing large graphs. When only static layout is available, browsing large graphs usually means drawing the entire graph and then viewing portions in a window with pan and zoom controls, fisheye lenses, etc. The problem is that the section in the current window may not be very informative. For example it may contain edge segments whose endpoint nodes are outside the window, or nodes whose placement can be rationalized globally but not locally. Incremental layout offers the alternative of directly adjusting the set of displayed objects to make informative displays.

Visualizing dynamic graphs. Often, data being visualized is subject to change. In our experience with the `dotty` system, we found many applications for graph animations:

- *CIAO* is a program database that displays dependencies between the types, data, functions and files in a C or *C++* program [6]. Programs change throughout their life-cycle as they are debugged, maintained, and improved, so graph views should reflect such changes.
- *Improvise* is a multimedia viewer for software process models [11]. These models are incrementally corrected and refined. Users report that stable incremental layout and manual editing of diagrams are essential.
- *LDBX* is a prototype graphical debugger that runs on an unmodified `dbx` text-based debugger [17]. It displays data structure graphs. Records are drawn as nested boxes containing primitive data or pointer fields. Pointers may be traced

interactively, or automatically by the system, yielding incremental graph updates.
- *VPM* displays distributed programs as graphs [4]. Processes and resources are drawn as nodes. Edges represent dependence and communication. Subgraphs (that is, *zones* [10] or *clusters* [16]) show distribution across hosts. *VPM* would benefit greatly from stable incremental graph layout. Furthermore, graph updates are issued at an exceptionally high rate (the rate of system call issue) so efficiency is critical.

Most graph drawing algorithms to date are not incrementally stable. They usually apply batch techniques to optimize objectives such as reducing total edge crossings or edge length. A small change in the input set, even just its ordering, may yield unpredictable, instable changes between successive layouts. This may occur even if a previous layout is taken as a starting configuration. The results can be confusing when viewing a sequence of layouts. An example is shown in fig. 1 made by an extension of the algorithm of Sugiyama et al [20]. Graphs (a) and (b) differ by only one edge. Although the drawing could be updated by moving a subgraph downward, as shown in (c), the layout system makes more drastic, unnecessary changes.

Most of our applications involve software engineering diagrams drawn hierarchically, so our immediate goal is to "incrementalize" our variant of Sugiyama's hierarchical drawing algorithm. Some similar issues, though, are encountered in making other kinds of layouts incrementally.

2 Previous Work

Significant progress has been made in drawing dynamic trees [15] (using subtree contours), planar graphs [2] and series-parallel graphs [7]. Although these are useful techniques for these restricted classes of graphs, they are not directly applicable to general graph drawing.

Hornick, Miriyala and Tamassia describe a practical incremental edge router for orthogonal drawings, such as entity-relation diagrams [14]. Nodes are placed externally (typically, by the user); the system routes edges incrementally by a shortest-path technique that accounts for edge length, crossings, and number of bends. The technique would have to be extended to handle automatic node placement and adjusting layouts to make space for new objects.

Lyons describes a way of incrementally improving layouts of undirected graphs such as those made by force directed modeling with unconstrained optimization [13]. The idea is to adjust regions where nodes are too close by computing Voronoi sets around each node and moving each node that has a conflict to a better place within its region. Because each node moves to a point that is closer to its old position than to any other node, faces are preserved. This technique is claimed to give better and more stable results than alternatives, such as re-solving a spring model with adjustments in springs intended both to force overlapping nodes apart, and to anchor nodes to their original positions. Lyons' algorithm is effective for this problem, but does not address how to maintain stability if the underlying graphs change. Also, virtual physical modeling is somewhat limited by the characteristic that all edges tend toward unit length.

Newbery and Bohringer show how to add stability constraints to graphs drawn by a batch technique in the *Edge* system using Sugiyama's directed graph drawing algorithm [20, 3]. After making an initial layout, intra-rank node ordering constraints are appended. If u, v are neighbors on rank r, they will appear in the same order in succeeding layouts as long as they stay on r. Adding such constraints is a good step toward making stable layouts of directed graphs, but preserving this ordering is not difficult if one assumes an on-line layout algorithm. The more fundamental problem is how to adjust layouts when groups of nodes must change ranks, and how to maintain geometric stability.

3 Incremental layout

3.1 Goals

The basic problem is, given a sequence of graphs

$$G_0, G_1, G_2, \ldots, G_n$$

interpreted as successive versions of G, find a "good" sequence of layouts

$$L_0, L_1, L_2, \ldots, L_n$$

where each L_i is a drawing of G_i. An update $G_i \rightarrow G_{i+1}$ may be written $U_i = (V+, V-, E+, E-)$ where these are sets of nodes and edges inserted and deleted.

There are important advantages if L_{i+1} resembles L_i:

- Users can retain a persistent "mental map" [8].
- Graphical updates reflect actual changes in the data.
- Large layouts can potentially be updated quickly.

The first two properties concern effective data visualization. In any context, visualization should help reveal meaningful patterns in data while avoiding irrelevancy and display artifacts. Efficiency is also desirable but not particularly important at this stage until the right problem has been identified.

While stability is important, it is not the only desirable characteristic for incremental layouts. We propose the following, in order of importance:

- consistency
- stability
- readability

Consistency or adherence to layout style rules is most important because the displayed diagram should always reveal properties of interest. Otherwise, visualization is pointless. For example, if the purpose of visualization is to demonstrate that a given graph is a tree, DAG, planar, or embeddedable in a grid, the diagram should always capture this property. If consistency is relaxed or abandoned, successive incremental layouts could quickly become obscure, ambiguous, or even incorrect. The second property, stability, refers to a principle of least change between successive

diagrams, subject to consistency. Readability refers to other properties to make diagrams pleasing and easy to read.

Assuming that consistency is more important than stability, then edits that cause fundamental changes in graph structure may be expected to cause large changes in layouts in order to maintain consistency. As a simple example, if a display shows two trees in a conventional downward drawing, and an edge is inserted that makes one tree become a subtree of the other, consistency requires moving the entire subtree, no matter how large. Thus, stability implies weak constraints on node and edge placement. This is a useful view because weak constraints may also reflect user-specified object placement requests.

3.2 Graph updates

An important question is what updates U_i to allow. Some possibilities include: arbitrary updates, single node or edge operations, append-only updates, homeomorphic expansion and subgraph abstraction (collapsing a subgraph into a node or restoring the subgraph [5, 18]).

3.3 Look-ahead

Often the entire sequence $G_0 \ldots G_i$ is known in advance. This should be important information and it is available whenever an animation is made off-line. In other situations, some look-ahead may be available, such as when updates are batched.

3.4 Stability

A key question is how to characterize stability between layouts. The answer depends on how people perceive and remember the structure of diagrams. For now we assume that important factors include:

- position (geometry)
- order (topology)

Geometry and order can be considered absolute properties, or relative to a neighborhood, such as the set of logically or geometrically adjacent nodes. An interesting proposition is that node stability is more crucial than edge stability. The rationale is that nodes are sites that users learn and return to in a diagram, while edges are generally traced on the fly to discover connections, and consequently their routing is less important. (Some researchers have suggested that interactive displays of dense layouts can be improved by making only a small subset of edges visible at a time.) If this is true, it seems advantageous to adjust edge routes aggressively to improve layouts, but move nodes more conservatively.

Locality (spatial and temporal) is often relevant in designing user interfaces. For graph layout, if a node is in the geometric or logical neighborhood of another that was recently updated, it may be a better candidate for update than a node that is not in any such neighborhood. Likewise, a node that was recently moved may be a better candidate for another update than one that was not recently adjusted. This suggests using “age” or ”memory” to control stability over time.

3.5 Display update

Smooth animation is often easier to understand than instantaneously switching images in a display. This means extending

$$L_i \rightarrow L_{i_0}, L_{i_1}, \ldots, L_{i_k}$$

to perform in-betweening. For smooth animation, some L_{i_j} may be inconsistent with the layout rules. Some nodes may overlap other nodes or edges as they are moving. Because it is cumbersome to support this directly in a layout system that assumes consistent diagram structure, we propose to separate the logical layout and physical display. The physical update layer is also an appropriate place to implement cues that emphasize updates, such as blinking or changing color.

4 *DynaDAG* Heuristic

4.1 Overview

DynaDAG is an incremental heuristic for drawing ranked digraphs, based on previous refinements to Sugiyama's heuristic [21, 9]. In the following discussion, we assume graphs are drawn in levels numbered from top to bottom, and that long edges are broken into chains of virtual nodes on adjacent ranks. *DynaDAG* preserves stability geometrically (exactly) and topologically (heuristically). It does not yet incorporate temporal information nor a separate display update module, but it is a suitable testbed for such future experiments.

As a simplifying assumption, *DynaDAG* supports only these operations:

$$\{ \text{ insert} \mid \text{optimize} \mid \text{delete} \} \times \{\text{node} \mid \text{edge}\}$$

More complex updates must be decomposed into these primitives. Our underlying hypothesis was that this decomposition yields good incremental layouts. This turned out to be partially correct, but an unnecessary restriction. The internal primitives of the heuristic do operate on only one node or edge at a time. On the other hand, applications often require performing a number of updates at once. In retrospect, it would make more sense to collect the updates and perform them together. This would not involve many changes to the heuristic.

The procedure *insert_node* (with an optional edge set) is most interesting because it may potentially involves moving many pre-existing nodes and edges. There are several phases. A rank assignment is determined for the incoming node. Pre-existing nodes and edges are adjusted to be consistent with this assignment. As a simplifying assumption, we compute new rank assignments by DFS and only move nodes downward. A reasonable enhancement would be to re-solve the global rank assignment problem and allow moving some nodes upward, symmetric to the downward case. Finally, the new node is installed with local optimization of its position and that of adjacent edges.

make_feasible moves an individual node to a different rank. First, the node is moved to the same X coordinate in the adjacent rank. Second, it is shifted right or left so that its label is locally consistent with the median sort order of nodes in the

new rank. This process is iterated until the node reaches its destination. Finally, the node's adjacent edges are updated by adjusting (shrinking, moving, or stretching) the virtual node chain. Virtual nodes are moved by a similar heuristic. Informally, when a node moves upward or downward, it follows a "valley" in the median function.

DynaDAG contains two heuristics that find edge routes. The first applies a variant of the median sort heuristic to any nodes and edges of the graph marked as movable with the rest of the configuration held fixed. The second heuristic routes individual edges by exhaustive search using limited backtracking [12].

update_geometry employs a form of linear programming for node coordinate assignment. This technique was previously introduced in `dot`. As illustrated in fig. 3, given an initial ranking, there is a way of adding a variables and two constraint edges to impose a linear penalty for moving a node from its old assignment. The construction involves creating an additional node as an anchor or reference point for the layout. The minimum lengths of the edges marked λ reflect the stable coordinate. The stable coordinate is 2.0 for u and 4.0 for v and w. In this figure, an additional constraint has been introduced between v and w that forces at least one of them to move away from its old position. The cost of this adjustment is set by the weights of the auxiliary edges. (A coarse approximation of a non-linear penalty could thus be simulated by summing linear terms.)

```
procedure insert_node(view, user_node, edge_set, hint_coord)
{
    //  map node to its layout representative
    v = layout_node(view, user_node);

    range = feasible_ranks(v, edge_set);
    if feasible(range)
        rank = choose_rank(v,hint_coord.y);
    else
        { rank = low(range); make_feasible(v,rank); }

    if is_valid_point(hint_coord) pos = hint_coord.x;
    else pos = mean_x(adjacent(v,edge_set));

    install_node(view,v,pos);
    install_edges(edge_set);
    opt_neighborhood(v,is_valid_point(hint_coord));
    update_geometry(view);
}

procedure insert_edge(view, orig_edge)
{
    u = layout_node(view, tail(orig_edge));
    v = layout_node(view, head(orig_edge));

    if path_exists(v,u) {temp = u; u = v; v = temp;}
    e = layout_edge(view, u, v);

    if rankof(u) + minlength(e) > rankof(v)
```

```
        make_feasible(v, rankof(u) + minlength(e));

    route_edge(e);
    update_geometry(view);
}

procedure make_feasible(v, v_rank)
{
    by DFS, find new ranks of nodes in G w.r.t. v on v_rank
    MS = { u in G : newrank(u) != oldrank(u) }
    for u in MS
        move_node_down(u,newrank(u))
    for u in MS
        for e in adjacent_edges(u)
            adjust_edge(e)
}

procedure move_node_down(v,newrank)
{
    x = position(v).x;
    for i = oldrank(v) + 1 to newrank(v) {
        set_medians(G,i);
        //  place_and_reopt makes a new leaf under v at
        //  the same x coordinate, moves it to a locally
        //  optimal position, and replaces the leaf with v.
        x = place_and_reopt(v,i);
    }
}

procedure adjust_edge(e)
{
    compute shrink, same, stretch segment sizes of e
    r = oldrank(tail(e)) + 1;
    for i = 1 to shrink
        { delete_vnode(e,r); r = r + 1; }
    for i = 1 to same
        { move_node_down(vnode(e,r),r+1); r = r + 1; }
    for i = 1 to stretch
        { copy_node_down(vnode(e,r),r+1); r = r + 1; }
}

procedure opt_neighborhood(node, node_is_movable)
{
    if node_is_movable
        set_movable(node,TRUE);
    for e in edges(node)
        set_movable(e,TRUE);        //  vnode chain

    range = movable_region(node);
    for iter = 1 to MAXITER {
        for r = range.low to range.high
```

```
            optimize_rank(r,r-1);   //  use in-edges
        for r = range.high downto range.low
            optimize_rank(r,r+1);   //  use out-edges
    }
    set_movable(n,FALSE);
    for e in edges(node)
        set_movable(e,FALSE);
}

procedure update_geometry(view)
{
    //  update the auxiliary constraint graph
    r = low_rank(view);
    while (r <= high_rank(view)) {
        v_left = leftmost_node(view, r);
        while (v_left) {
            constrain_prevposition(v_left);
            constrain_outedge_cost(v_left);
            v_right = right_neighbor(v_left);
            if v_right
              constrain_separation(v_left,v_right);
            v_right = v_left;
            v_left = right_neighbor(v_left);
        }
        r = r + 1;
    }
    //  invoke network simplex coordinate solver
    ns_solve(view);
    //  node and edge callbacks can be done here
}
```

This heuristic has some useful properties. Its internal primitives are not difficult to program. Some can be applied individually, opening the way to explore higher-level incremental update strategies, for example, allowing a heuristic or the user to identify specific nodes and edges to be re-optimized. Further, using linear constraints and weights to solve coordinates gives precise control of tradeoffs between consistency, stability, and readability and yields predictable results.

Figure 4 shows frames from an animation created by incrementally inserting all the nodes and edges of an example graph. Most updates are apparently stable; an exception can be found between frames 26 and 27, where there is a larger adjustment, but other parts of the layout still closely resemble previous frames.

4.2 Incremental Layout Systems

We implemented *DynaDAG* and several other algorithms to provide that provide an incremental layout service through a library interface (*DynaGraph*). The other algorithms include an implementation of the Hornick-Miriyala-Tamassia orthogonal embedder, and a spring embedder with a constraint enforcement heuristic (following Lyons [13]). In *DynaGraph*'s model, an abstract graph may have one or more inde-

pendent views, each containing a subset of the base graph. *DynaGraph* deals only with graphs and coordinates, and is window-system independent.

We created several compatible graph viewers on top of this interface. *DynaGraph* is an OLE-compliant Microsoft Windows graph viewer.[1] It can act as a server, managing active diagrams embedded in other documents, and as a client allowing external objects to be embedded as nodes. This greatly simplifies integration of hypertext documents or multimedia clips in graph diagrams.

dged is a programmable front end implemented in the Unix TCL/tk environment.[2] Fig. 2 is a sample of some output frames in a sequence that was created by inserting new nodes and edges. (In this execution of *DynaDAG*, stability of absolute coordinates was intentionally disabled on nodes that change ranks to see how this affects displays.) *dged* supports multiple views maintained in synchrony by different layout engines. Its intended use is to construct prototype network management utilities.

5 Conclusion

DynaDAG is a heuristic for hierarchical layout of directed graphs that incorporates geometric and topological stability. It incorporates a heuristic to move nodes between adjacent ranks, based on median sort. The heuristic is effective for viewing incremental layouts of graphs of at least several dozen nodes, though further tuning is needed. Experience with a working implementation in real applications offers invaluable guidance. The heuristic does not use look-ahead, temporal information, or adaptive update strategies; there is a good opportunity for further work here.

There are many factors that may affect how users perceive stability in graph drawings. More work is needed to understand what properties are most important, and to find efficient incremental layout algorithms for general graphs.

References

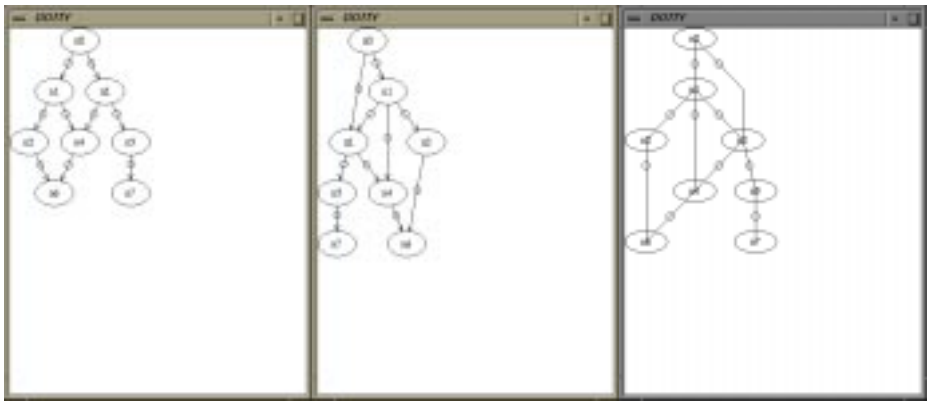
1. G. Di Battista, P. Eades, R. Tamassia, and I.G. Tollis. Algorithms for drawing graphs: An annotated bibliography. *Computation Geometry: Theory and Applications*, 4(5):235–282, 1994. Available at ftp.cs.brown.edu in /pub/compgeo/gdbiblio.tex.Z.
2. G. Di Battista and R. Tamassia. Incremental planarity testing. In *Proc. 30th IEEE Symp. on Foundations of Computer Science*, pages 436–441, 1989.
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. Yih-Farn Chen, Glenn S. Fowler, David G. Korn, Eleftherios Koutsofios, Stephen C. North, David S. Rosenblum, and Kiem-Phong Vo. Intertool connections. In B. Krishnamurthy, editor, *Practical Reusable UNIX Software*, chapter 11. Wiley, 1995. To appear January 1995.
5. Yih-Farn Chen, Leftheris Koutsofios, and David Rosenblum. Intertool connnections. In Balachander Krishnamurthy, editor, *Practical Reusable UNIX Software*, chapter 11. John Wiley & Sons, 1995.

[1] Written by Giampiero Sierra, Princeton University.

[2] Written by John Ellson, AT&T Bell Laboratories.

6. Yih-Farn Chen, Michael Nishimoto, and C. V. Ramamoorthy. The C Information Abstraction System. *IEEE Transactions on Software Engineering*, 16(3):325–334, March 1990.
7. Robert F. Cohen, Giuseppe Di Battista, Roberto Tamassia, and Ionnis G. Tollis. Dynamic graph drawings: Trees, series-parallel digraphs, and planar st-digraphs. In *Proc. Symposium on Computational Geometry*, pages 261–270, 1992. to appear in SIAM J. Computing.
8. P. Eades, W. Lai, K. Misue, and K. Sugiyama. Preserving the mental map of a diagram. In *Proceedings of Compugraphics 91*, pages 24–33, 1991.
9. E.R.Gansner, E. Koutsofios, S.C. North, and K.-P. Vo. A technique for drawing directed graphs. *IEEE Trans. on Soft. Eng.*, 19(3):214–230, 1993.
10. C. Kosak, J. Marks, and S. Shieber. Automatic the layout of network diagrams with specific visual organization. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-24(3):440–454, 1994.
11. B. Krishnamurthy and N. Barghouti. Provence: A Process Visualization and Enactment Environment. In *Proc. of the Fourth European Conference on Software Engineering*, pages 151–160, Garmisch-Partenkirchen, Germany, September 1993. Springer-Verlag. Published as *Lecture Notes in Computer Science* no. 717.
12. Panagiotis Linos, Vaclav Rajlich, and Bogdan Korel. Layout heuristics for graphical representations of programs. In *Proc. IEEE Conf. on Systems, Man and Cybernetics*, pages 1127–1131, 1991.
13. K. Lyons. Cluster busting in anchored graph drawing. In *Proceedings of the 1992 CAS Conference*, pages 7–16, 1992.
14. Kanth Miriyala, Scot W. Hornick, and Roberto Tamassia. An incremental approach to aesthetic graph layout. In *Proc. Sixth International Workshop on Computer-Aided Software Engineering*, pages 297–308. IEEE Computer Society, July 1993.
15. S. Moen. Drawing dynamic trees. *IEEE Software*, 7:21–8, 1990.
16. Stephen C. North. Drawing ranked digraphs with recursive clusters. In *Proc. ALCOM Workshop on Graph Drawing '93*, September 1993. submitted.
17. Stephen C. North and Eleftherios Koutsofios. Applications of Graph Visualization. In *Graphics Interface '94*, pages 235–245, 1994.
18. F. Newbery Paulish and W.F. Tichy. Edge: An extendible graph editor. *Software – Practice and Experience*, 20(S1):1/63–S1/88, 1990. also as Technical Report 8/88, Fakultat fur Informatik, Univ. of Karlsruhe, 1988.
19. L.A. Rowe, M. Davis, E. Messinger, C. Meyer, C. Spirakis, and A. Tuan. A browser for directed graphs. *Software – Practice and Experience*, 17(1):61–76, 1987.
20. 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.
21. 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.

This article was processed using the LaTeX macro package with LLNCS style



(a) (b) (c)

Fig. 1. stable and instable updates

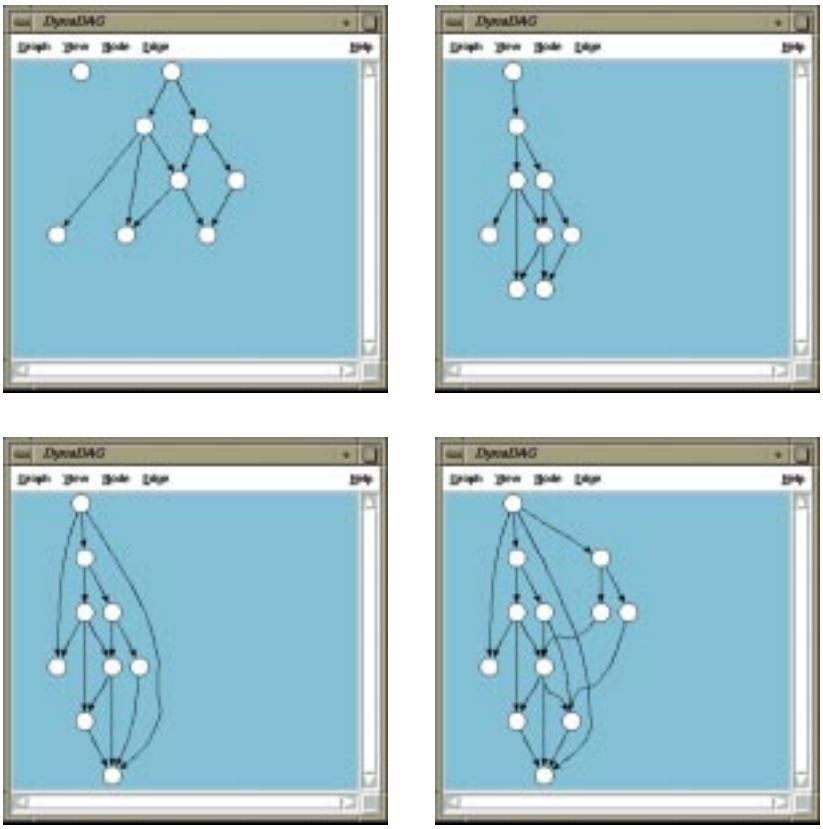


Fig. 2. *DynaDAG* running in *dged*

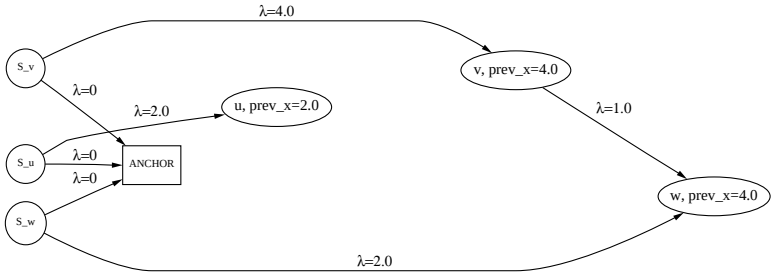


Fig. 3. measuring node stability

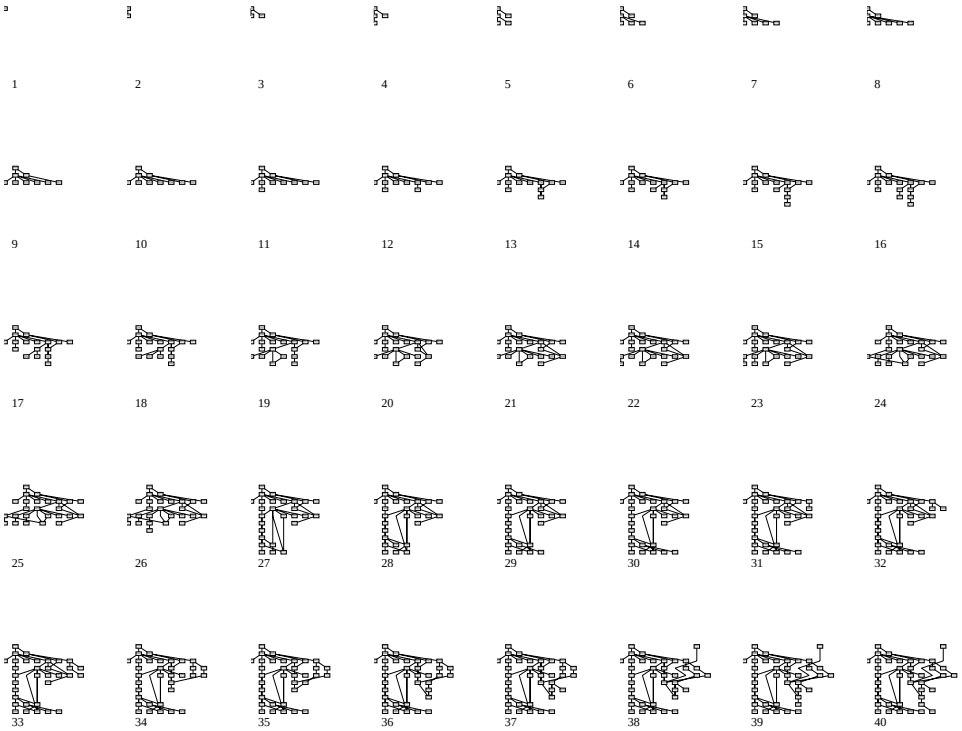


Fig. 4. *DynaDAG* animation