Visual Navigation Through Large Directed Graphs and Hypergraphs*

Jason Eisner Michael Kornbluh Gordon Woodhull Raymond Buse Samuel Huang
Constantinos Michael George Shafer†

Johns Hopkins University

ABSTRACT

We describe Dynasty, a system for browsing large (possibly infinite) directed graphs and hypergraphs. Only a small subgraph is visible at any given time. We sketch how we lay out the visible subgraph, and how we update the layout smoothly and dynamically in an asynchronous environment. We also sketch our user interface for browsing and annotating such graphs—in particular, how we try to make keyboard navigation usable.

Keywords: Dynamic graph layout, graph browsing, hypergraph navigation, graph animation.

Index Terms: I.3.6 [Computer Graphics]: Methodology and Techniques—interaction techniques; H.5.2 [Information Interfaces and Presentation]: User Interfaces—graphical user interfaces; G.2.2 [Discrete Mathematics]: Graph Theory—hypergraphs

1 INTRODUCTION

Relational data can often be expressed as a directed graph or hypergraph. Unfortunately, real-world graphs are typically too large and tangled for static drawings of them to be comprehensible. We describe a system that allows *local browsing* of such structures.

Directed graphs have many natural applications. They can be used to visualize social or physical networks, taxonomies or ontologies, finite-state automata, and various relations in software design and development (subclass-of, calls, points-to).

Hypergraphs have received less attention. Our work applies to them as well. Some uses of directed *hypergraphs* include

- **digital circuits:** two or more inputs combine via a logic gate to produce one or more outputs
- **theorem proving:** two or more statements combine via a proof rule to derive a new statement
- **parse forests:** two or more adjacent phrases of a linguistic sentence combine to produce a larger phrase
- **genealogy:** two parents combine to produce zero or more children
- **chemical reactions:** two or more reactants (and perhaps catalysts) combine to produce several products

Each of these bullet points describes the form of a *single* directed hyperedge. (A directed hyperedge is a generalization of a directed edge: it connects a *set* of input vertices to a *set* of output vertices.) A database of many chemical reactions would need one hyperedge per reaction. Thus, the hyperedges connected to a given vertex (chemical substance) would show *all* reactions in which it participates.

Our original motivation was to build a visual debugger for Dyna [2], a declarative programming language that is designed for dynamic programming. A Dyna program does not prescribe an order of computation, but only specifies how some values are to be derived from others. Thus, Dyna debugging focuses not on procedural single-stepping but on the declarative relationships—hyperedges—among (millions of) derived values. The same would hold when debugging theorem provers, makefiles, or deductive databases.

Our open-source project, Dynasty, resides at `http://dyna.org/Dynasty`, where the reader may find screenshots and video.

2 DIRECTED HYPERGRAPH LAYOUT

Dynasty is willing to lay out any subgraph. Its layout engine, Dynagraph [5], chooses node positions and splined edge routes for the subgraph that is currently in view. It uses Sugiyama-style layout, closely following the `dot` program for static digraphs [3].

By contrast, TreePlus [4]—the only other browser we know of that focuses on large *directed* graphs—displays only tree-shaped subgraphs as the user browses. It therefore simplifies away cycles and reentrancies, which we choose to show (at some risk of tangles if the subgraph is made large).

We have improved Dynagraph in several ways:

Hyperedges To lay out a non-trivial hyperedge, we introduce an intermediate vertex, the "crux."[1] The source nodes are connected to the crux by ordinary directed edges, whose spline renderings "flow together" into the (otherwise invisible) crux with a common tangent vector. This tangent vector "flows apart" again as it leaves the crux, separating into edges to the target nodes.

S-Shaped Backedges Sugiyama-style layout already emphasizes the directional flow of a graph from top to bottom. It tries to set nodes' y coordinates ("ranks") to avoid upward edges. When cycles necessitate such backedges, we preserve the flow by making each backedge "S-shaped": it exits its source node from the bottom (like all out-edges), but doubles back up to reach its target node, which it enters from the top (like all in-edges). Thus, edge direction and cycles are easy to see without hunting for arrowheads.[2]

By insisting on top-to-bottom (or left-to-right) flow, we can support traditional intuitive layouts of trees, genealogies, proof forests, and so on. This is not possible with graph browsers that do not emphasize directional flow, such as TreePlus [4] and those based on radial or force-directed layout (but see [1]).

Ordered Edges We can constrain the ordering of a node's (or crux's) out-edges as they fan out from the node (or crux).[3] Sometimes this has semantic content: e.g., in an arithmetic expression's parse tree, the order of arguments to subtraction is significant.

Stubs As only a small subgraph is shown at any time, it is important to advise the user about what lies beyond. Each visible node or crux advertises its degree by showing all its incident edges. Edges to currently invisible neighbors are displayed as short stubs.

If two visible nodes are endpoints of the same hyperedge, we insist on showing this relation, by ensuring that the crux is visible.

*This work has been supported in part by NSF ITR grant IIS-0313193 to the first author and by Joseph C. Pistritto Research Fellowships to the second and fifth authors. The views expressed are the authors' only.

†e-mails of all authors: jason@cs.jhu.edu, kornbluh@gmail.com, gordon@woodhull.com, data@jhu.edu, srhuang@jhu.edu, stvorph@gmail.com, playswithfire@gmail.com

[1] For a plain 1-to-1 edge, the crux is logically present but not laid out.

[2] We have also experimented with a special color for backedges.

[3] For some graphs, they may be forced to cross later along the length of the edge, although the layout engine tries to avoid this.

3 Topological Navigation with the Keyboard

It should be easy to locate (and visit) a node's parents, children, and siblings in the graph. The natural approach is keyboard navigation. But navigation in graphs is confusing. Where should the "up" and "down" keys go if the cursor node has *multiple* parents or children? Should "left" and "right" visit one's co-children (i.e., half-siblings) or one's co-parents (i.e., mates)? And which ones?

Our solution involves a visible **trail** that connects the cursor node to its just-visited parent or child, called the **pivot** node. The left and right keys step through the co-children of a pivot parent, or the co-parents of a pivot child. (To change the pivot, one can navigate while holding down the Ctrl key. We also have some heuristic selection of new pivots when left and right fail.)

Like ants, the up/down keys remember the most recent upward and downward trails from each node, and prefer to follow those.[4] This makes navigation predictable, and ensures that up and down cancel each other out.

We extend this interface cleanly to hypergraphs. If the user holds down the Alt key, the arrow keys simply navigate the bipartite graph consisting of nodes and cruxes. Thus the cursor and pivot can rest on cruxes (that is, on hyperedges). Without the Alt modifier, the arrow keys always pass through cruxes to arrive at real nodes.

4 Updating the Layout

If the user moves the cursor into invisible territory, by keyboard or mouse, we reveal more of the underlying graph by adding nodes to the visible graph. The camera primarily follows the cursor. We control the size of the visible graph by pruning nodes far from the cursor (by path distance). The user can also manipulate the graph size directly, or "refocus" to show the neighborhood of the cursor.

When the visible graph changes, the layout engine computes a new layout for it, trying to balance layout quality and stability. It is most important to preserve stability near the cursor and pivot, and in particular to preserve the fanning order of the pivot node during left/right movement.[5] The display engine animates smoothly to the new layout, while fading out old nodes and fading in new ones.

The visible graph may change for reasons other than navigation. Relayout may be triggered if the large underlying graph changes (see section 6); if the user drags nodes or suppresses nodes (see section 5); or—in future—if the user adjusts a threshold slider to display only highly-weighted nodes and hyperedges.

5 Selection, Search and Annotation

The user can select multiple nodes and hyperedges, in addition to the one beneath the cursor. Dynasty supports selection through various keyboard, mouse, command, and search interfaces.

Selected objects are highlighted if they are in the currently visible subgraph. However, some of them may be outside that subgraph (just as a selection in a word processor may extend outside the visible window). This is particularly powerful when selecting a potentially infinite set of nodes with a search query.

Once nodes and/or hyperedges are selected, it is easy to cycle through visiting them with the cursor, or to apply an action to them:

- refocus the visible graph on the selection
- change color, shape, or font properties
- delete the selection
- contract the graph across the selection

[4] Where there is no remembered trail, they use geometric heuristics.

[5] For this reason, we take some care choosing the fanning order of stubs with respect to full edges. Placing a stub randomly would tend to create conflicts between layout quality and stability when the stub was later expanded. Thus, we "plan ahead" as to where the invisible neighbor might go.

This makes it possible to colorize important nodes or suppress nodes that represent unimportant intermediate steps. Such annotations affect the entire selection, beyond the visible subgraph. Ordinarily, they do not modify the underlying graph but rather reside in an "annotated graph" layer that wraps around the underlying graph.

In future, we would like to support more forms of graph analysis. A simple example is to highlight the paths or hyperpaths that relate selected nodes. Another example is cluster discovery.

6 Asynchronous Changes

Dynasty is designed to operate correctly in a dynamic environment where, simultaneously,

A. An agent is updating the underlying graph to reflect new info.
B. User navigation is changing the visible subgraph.
C. The layout engine (a concurrent process that assigns positions) is catching up with changes to the visible subgraph.
D. The display engine is animating to the latest target layout.

underlying graph → visible subgraph → layout → animated display

While we do not have space to describe the details of our message-passing layered architecture, we note the main issues:

A. The visible subgraph periodically polls the relevant part of the underlying graph to detect any changes. This is much faster than informing the visible subgraph about all changes everywhere in the large (or infinite) underlying graph.

B. Changes to the visible subgraph are propagated to the layout engine, which may still be working on the old layout. We have reorganized the layout engine, Dynagraph [5], so that it can be interrupted by such new requests. It tries to salvage any relevant work it has done on the old layout.

C. The layout engine may take some time to choose new positions and spline routes and send them to the display engine. Our architecture can handle arbitrary layout delays. However, for responsiveness, we have also improved the layout engine to send quick-and-dirty positions at first and later revise them, as it works to untangle edge crossings.

D. The layout engine tells the display engine only how to update the *target* layout. The display engine must compensate if it was still animating toward the *old* target layout. Ongoing trajectories must swerve toward their revised target positions and possibly change speed. New nodes should not appear at their layout targets if their neighbors have not yet reached their own targets; they should fade in along similar trajectories. Conversely, deleted nodes that are still fading out should be dragged along if their neighbors move or swerve, even though the layout engine is no longer responsible for them.

References

[1] L. Carmel, D. Harel, and Y. Koren. Drawing directed graphs using one-dimensional optimization. In *Proceedings of the 10th International Symposium on Graph Drawing (GD'02)*, volume 2528 of *Lecture Notes in Computer Science*, 2002.

[2] J. Eisner, E. Goldlust, and N. A. Smith. Compiling comp ling: Weighted dynamic programming and the Dyna language. In *Proc. of the Conf. on Empirical Methods in Natural Language Processing*, 2005.

[3] E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo. A technique for drawing directed graphs. *Software Engineering*, 19(3), 1993.

[4] B. Lee, C. S. Parr, C. Plaisant, B. B. Bederson, V. Veksler, W. Gray, and C. Kotfila. TreePlus: Interactive exploration of networks with enhanced tree layouts. *IEEE Transactions on Visualization and Computer Graphics*, to appear 2006.

[5] S. C. North and G. Woodhull. Online hierarchical graph drawing. In *Revised Papers from the 9th International Symposium on Graph Drawing (GD'01)*, volume 2265 of *Lecture Notes in Computer Science*, 2001.