graphael: A System for Generalized Force-Directed Layouts*

D. Forrester, S. Kobourov, A. Navabi, K. Wampler, and G. Yee

Department of Computer Science
University of Arizona
{forrestd,kobourov,navabia,wamplerk,gyee}@cs.arizona.edu

Abstract. The graphael system implements several classic force-directed layout methods, as well as several novel layout methods for non-Euclidean geometries, including hyperbolic and spherical. The system can handle large graphs, using multi-scale variations of the force-directed methods. Finally, the system can layout and visualize graphs that evolve though time, using static views, animation, and morphing. The system is written in Java and is available as a downloadable program or as an applet at http://graphael.cs.arizona.edu.

1 Introduction

As researchers in the graph drawing community develop new algorithms and visualization techniques, it is natural for the creation of new graph drawing tools to follow. It is often the case, however, that the implementation of an algorithm is accompanied by time consuming tasks that have little to do with the algorithm itself. Researchers who would like to test a new layout algorithm should only have to concern themselves with the details of the algorithm itself and not other things such as graphics packages, file parsers, or user interface design.

In this paper we present graphael: yet another graph drawing system designed to provide the necessary structure and flexibility for force-directed graph drawing research. Our system is built with the following design considerations: (1) *Plug-and-Play:* it should be easy to integrate new algorithms and visualization methods; (2) *User Friendliness:* the user interface should be easy to use, but also powerful and versatile; (3) *Portability:* the system should run on any computational platform.

The graphael system attempts to meet these goals by providing a set of core algorithms and visualization routines, as well as a powerful interface that allows the user to combine different algorithms and visualization methods, and to easily add new ones. In addition, the system contains several novel algorithms and visualization techniques, such as force-directed methods in hyperbolic and spherical spaces, and techniques for dealing with graphs that evolve through time. The system is written in Java and is available as a downloadable program or as an applet at http://graphael.cs.arizona.edu.

1.1 Related Work

A number of automated graph drawing systems have been developed over the last few years; see [8] for a survey. The systems mentioned below is by no means a complete list, but intends

* This work is supported in part by the NSF under grant ACR-0222920 and ITCDI inder grant 003297.

to include the ones most relevant to our system. The GraphServer [13] is an online service that allows users to draw graphs and translate graph descriptions between multiple formats. Tulip is a framework built to facilitate large graph drawing research [1]. WilmaScope [3] is a Java application designed specifically for 3D visualization. yFiles [14] is a commercial library of Java classes developed to provide building blocks for graph drawing applications. Pajek [2] is a Windows program designed to handle large graphs for social networks analysis. `TGRIP` [5] is an extension on the `GRIP` system [7] and efficiently draws large temporal graphs using intelligent placement. The `GraphAEL` [4] system extracts three types of evolving graphs from a custom-built graph drawing literature database and creates 2D and 3D animations of the evolutions.

1.2 Our Contributions

In addition to sharing all the letters with the `GraphAEL` [4] , the `graphael` system was inspired by it. We wanted to provide a graph visualization framework that can easily be coupled with the bibliographic database to provide visualizations of the co-citation, collaboration, and topic graphs, produced from the database. This led to the development of the current system, which is equipped with a core package of force-directed algorithms and visualization tools. In addition to putting together well-known algorithms and visualization methods, `graphael` contains several novel features. Some of these features include support for temporal graphs, interactive graph visualization, multi-scale layout algorithms for large graphs, and embedding graphs in non-Euclidean spaces, such as hyperbolic space and spherical space.

2 System Overview

In this section we summarize the layouts, visualization methods, and other tools packaged in `graphael`. In-depth descriptions of novel features are also provided.

2.1 Force-Directed Layouts

Force-directed layout algorithms are a powerful and practical graph drawing heuristic. They rely on an objective function that maps a particular graph layout to an energy value. Typically such algorithms start with a random drawing of the graph and utilize standard optimization methods to minimize the energy function. The algorithms define functions in which low energies are associated with layouts where adjacent nodes are near some preferred distance from each other, and non-adjacent nodes are well-spaced. The main difference between force-directed algorithms is the choice of energy function and the methods for its minimization.

We have implemented two force-directed algorithms in `graphael`. The Fruchterman-Reingold [6] algorithm defines an attractive force function for adjacent nodes and a repulsive force function for non-adjacent nodes. For a vertex v, $F_{FR}(v) = F_{a,FR} + F_{r,FR}$, where the attractive force, $F_{a,FR}$, and repulsive force, $F_{r,FR}$, are given by:

$$F_{a,FR} = \sum_{u \in Adj(v)} \frac{\mathrm{dist}_{R^n}(u,v)^2}{\mathrm{edgeLength}^2}(\mathrm{pos}[u] - \mathrm{pos}[v]),$$

$$F_{r,FR} = \sum_{u \in Adj(v)} s \cdot \frac{\mathrm{edgeLength}^2}{\mathrm{dist}_{R^n}(u,v)^2} \cdot (\mathrm{pos}[u] - \mathrm{pos}[v]).$$

Another force-directed method implemented in `graphael` is the Kamada-Kawai [9] layout algorithm. In this method each pair of nodes connected by a path has forces proportional to the length of the path. The displacement of a vertex v of G is calculated by:

$$F_{KK}(v) = \sum_{u \in N_i(v)} \left(\frac{\text{dist}_{R^n}(u,v)^2}{\text{dist}_G(u,v) \cdot \text{edgeLength}^2} - 1 \right) (\text{pos}[u] - \text{pos}[v]).$$

In the above equations, $dist_{R^n}(u,v)$ is the Euclidean distance between pos$[u]$ and pos$[v]$, $dist_G(u,v)$ is the graph distance between u and v along a shortest path, edgeLength is the unit edge length, Adj(v) is the set of vertices adjacent to v, and s is a small scaling factor.

2.2 Multi-scale Graph Drawing

The effectiveness of force-directed methods rapidly decreases as the input graphs get larger. This is mainly due to the increased difficulty of getting out of local minima and to the runtime complexity, typically quadratic, or cubic in the size of the graph. Multi-scale graph drawing methods address both of these problems by filtering the graph into different levels, called filtration levels, each containing a subset of the initial graph. The filtration levels are laid out from least to most complex, which allows the structure of the graph to form well from the start. The multi-scale method relies on good filtrations, good initial placement of the vertices, and on local refinement on each level.

Filtrations: The effectiveness of the multi-scale method depends on each successive filtration level containing a constant fraction of the nodes from the previous level. Thus, good filtrations have $O(\lg n)$ depth and can be quickly computed. In `graphael` we currently provide three methods of filtering the graph: Maximal Independent Set (MIS) Filtration, Random Graph Filtration, and Cores Filtration:

1. *Maximal Independent Set Filtration:* A filtration $V = V_0 \supset V_1 \supset \ldots \supset V_k \supset \emptyset$ of the vertex set V of G is called a maximal independent set filtration if V_1 is a maximal independent set of G, and each V_i is a maximal subset of V_{i-1} so that the graph distance between any pair of its elements is at least $2^{i-1} + 1$. The graph distance between a pair of vertices is defined as the length of the shortest path between them in the graph. MIS filtrations have depth $O(\lg n)$ and can be computed in near-linear time [7].
2. *Random Graph Filtration:* Random filtrations are created by repeatedly removing half of the vertices, chosen at random, starting with the original vertex set V of G. The depth of this filtration is also $O(\lg n)$ and the computation time required is linear in the size of V. Although simple, this method produces reasonable layouts for large graphs.
3. *Cores Graph Filtration:* Graph cores are described in [12]. Given a graph $G = (V, E)$, a subgraph $H_k = (W; E|W)$ induced by the set W is a k-core, or a core of order k if $\forall v \in W : deg_H(v) \geq k$, and H_k is the maximum subgraph with this property. The core of maximum order is also called the main core. The core number of vertex v is the highest order of a core that contains this vertex. Graph cores can be computed in linear time [2]. If the number of cores is a small constant compared to the size of the graph, we augment the filtration induced by the cores to depth $O(\lg n)$ using the peeling process inherent in the core computation.

Initial Placement and Refinement: The main idea of good initial placement is to add vertices to the current drawing one at a time at a carefully computed position, rather than a random one. For simplicity we describe the process in 2D, but in practice this is done in arbitrary Euclidean, and even some non-Euclidean, spaces. Assume that the highest filtration level has exactly 3 vertices. These vertices are placed at the endpoints of a triangle with sides proportional to the graph distances between the points in the original graph. Vertices in subsequent filtration levels are placed based on their graph distances from already placed vertices from previous filtration levels. The intuition is that if we place the vertices close to their optimal positions initially, the refinement phase will only need a few iterations of a local force-directed calculations to reach a minimal energy state. In `graphael`'s implementation, we use the "3-closest-vertices" strategy. Using this method we place the vertex t at the barycenter of u, v, and w, the three vertices closest to t from the previous filtration level. Once all the vertices at the current filtration level have been placed, we apply a local force-directed refinement. The refinement stage is local as for a given node v in the current filtration, only a small neighborhood of vertices $N_i(v)$ is considered in the force computation.

2.3 Graphs that Evolve Through Time

We have implemented algorithms in `graphael` for visualization of graphs that evolve through time based on techniques described in [4, 5]. The algorithms are modifications of the standard Fruchterman-Reingold and Kamada-Kawai algorithms that allow us to deal with vertex-weighted and edge-weighted graphs. Graphs that evolve through time are converted to node-weighted and edge-weighted graphs, by treating each instance of the graphs as a timeslice, and connecting neighboring timeslices. The edges connecting different timeslices are called inter-timeslice edges. By changing the weights of these edges, we are able to balance the individual graph readability with the overall mental map preservation between consecutive graphs. Making the inter-timeslice edges heavy, results in fixing the vertex positions in each graph instance. Alternatively, making the inter-timeslice edges light, results in nearly independent layouts of each graph instance.

Weighted Graphs: We modify the force-directed equations for calculating the force vectors to include edge weights and vertex weights so as to place heavy nodes well away from each other and to place vertices connected by heavy edges closer to each other. The unit edge length is modified for weighted graphs to $\sqrt{w_u \cdot w_v}/w_e$, for an edge of weight w_e, connecting vertices u, v of weight w_u, w_v, respectively.

The Kamada-Kawai method relies on the notion of graph distance between pairs of vertices. It is easy to generalize this notion to weighted graphs, but because of the computational and space overhead associated with calculating the shortest path between all pairs of vertices in the graph, we use an approximation. Let $p_1, p_2, \ldots, p_n$ be the sequence of vertices in the shortest unweighted path in G connecting two vertices, u and v. The modified Kamada-Kawai force vector is given by:

$$F_{KK}(v) = \sum_{u \in N_i(v)} \left(\frac{2 \cdot \text{dist}_{R^n}(u,v)^2}{\text{optDist}_G(u,v)^2 \cdot \text{edgeLength}^2 + \text{dist}_{R^n}(u,v)^2} - 1 \right) (\text{pos}[u] - \text{pos}[v]),$$

where $\text{optDist}_G(u, v)$ is defined by:

$$\text{optDist}_G(u,v) = \sum_{i=2}^{n} \frac{\sqrt{w_{p_i} \cdot w_{p_i-1}}}{w_{e_{p_i p_{i-1}}}}.$$

Similarly, we modify the Fruchterman-Reingold forces as follows:

$$F_{a,FR} = \sum_{u \in Adj(v)} \frac{w_e \cdot \text{dist}_{R^n}(u,v)^2}{\text{edgeLength}^2}(\text{pos}[u] - \text{pos}[v]),$$

$$F_{r,FR} = \sum_{u \in Adj(v)} s \cdot \left(\frac{\text{edgeLength}^2 \cdot \sqrt{w_u \cdot w_v}}{\text{dist}_{R^n}(u,v)^2} \right) (\text{pos}[u] - \text{pos}[v]).$$

Timeslice Attribute: To visualize a series of graphs embodying the evolution of a set of relationships over time, we associate a timeslice attribute with each vertex. The timeslice of a vertex is just a label identifying which graph instance the vertex belongs to. We use the timeslice attribute to partition the vertices of a graph into groups by time. Additional modifications to the force-directed algorithms are needed to accommodate timeslice information. For the Kamada-Kawai layout method, repulsive forces for a given node u on node v should be equal to the repulsive force of w on v, where u and w are corresponding nodes from different timeslices. Thus, the function $\text{optDist}_G(u,v)$ is modified so that for two vertices u, v with timeslice indices of t_u and t_v, respectively, is given by :

$$\text{optDist}_G(u,v) = \sum_{i=2}^{n} \delta_{t_u t_v} \cdot \frac{\sqrt{w_{p_i} \cdot w_{p_{i-1}}}}{w_e},$$

where $p_1, p_2, \ldots, p_n$ is the shortest unweighted path in G connecting two vertices, u and v and $\delta_{t_u t_v}$ is 1 if $t_u = t_v$ and 0 otherwise.

The modifications needed for the Fruchterman-Reingold calculations are similar. Repulsive forces are simply eliminated between vertices in different timeslices, $F_{r,w,t,FR} = \delta \cdot F_{r,w,FR}$ while the attractive forces remain unchanged, $F_{a,w,t,FR} = F_{a,w,FR}$.

2.4 Visualizing Evolving Graphs

The timeslice information alone is not enough to nicely layout evolving graphs; we must also arrange edges between timeslices so that the layouts can be used for animation. The most straightforward method to animate is simply to use a series of "snapshots" of a graph taken at some interval over a period of time. When visualizing an evolving graph, we would ideally like the graphs of each timeslice to have high readability (i.e. have a pleasing layout) and for consecutive timeslices to be similar, that is, the mental map should be preserved. To meet these constraints the timeslices are combined into a single graph by connecting vertices with the same labels from adjacent timeslices.

Because of the modified optimal distance function, corresponding nodes in different timeslices have no repulsive force on each other, but they still have attractive forces due to the inter-timeslice edges. In `graphael` the balance between readability and mental map preservation can be controlled by changing the weights of the inter-timeslice edges.

Once the layout of the evolving graph has been computed, `graphael` offers different methods for visualizing the graphs. Each timeslice can be drawn in a restricted 2D view, or the graphs can be drawn in 3D with individual graphs arranged on top of each other (called

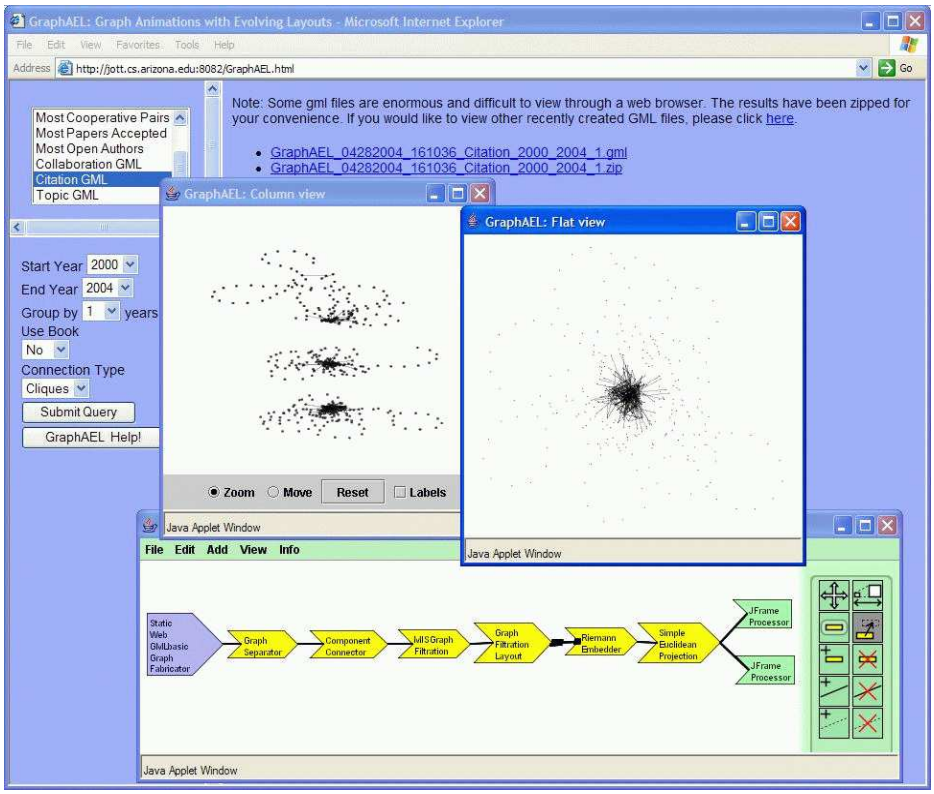

Fig. 1. `graphael` in cooperation with `GraphAEL`. The graphs shown are the column view and flat 2D view of a citation graph from 2000 to 2003 by 1 year increments.

column-view). Smoothly stepping through the evolving graphs using linear interpolation can also produce visually pleasing animations of the evolving graphs.

The column view lays out each timeslice on a separate plane, allowing the user to view the changes in the graph over time; see Fig. 1. The inter-timeslice edges can be hidden or displayed[1]. Simple navigation controls such as zoom and move are also provided in this view to allow the user to better view the graph. It might be difficult for the user to see all of the changes between two times just by looking at two separate graphs though, so we provide an animation view, which animates the changes between different timeslices in a graph. The simplest way to animate a time-evolving graph is to display one timeslice after the next, but the user might not see that some nodes have moved. Therefore, we also allow the user to play the animation so that a smooth transition is made from one time to the next, linearly interpolating node positions, colors, and weights. Nodes and edges also fade in and out from one timeslice to the next.

2.5 Graph Drawing in Non-Euclidean Spaces

A novel feature in `graphael` is the option to layout graphs in non-Euclidean spaces, in particular, in hyperbolic space and in spherical space. Existing force-directed algorithms

[1] If the inter-timeslice edges are absent in the input file `graphael` adds them based on node labels.

are restricted to calculating a graph layout in Euclidean geometry. Euclidean space has a very convenient structure for force-directed methods. It is easy to define distances and angles, and the relationship between the vector representing the net force on an object and the appropriate motion of that object are quite straightforward. There are, however, cases where Euclidean geometry may not be the best option. Certain graphs may be known to have a structure that would be best realized in a different geometry, such as on a sphere or on a torus. Furthermore, it has also been noted that certain non-Euclidean geometries, specifically hyperbolic geometry, have properties which are particularly well suited to the layout and visualization of large classes of graphs [11].

Riemannian Geometries: In `graphael` we have implemented a generalization of force-directed methods to non-Euclidean geometries [10]. These methods use mappings between non-Euclidean geometries and corresponding tangent spaces. While a non-Euclidean geometry does not afford all of the conveniences of Euclidean geometry, there is a straightforward way to define distances and angles, provided we restrict ourselves to geometries which are smooth. Such geometries are known as Riemannian geometries, and while they have less convenient structure than Euclidean geometry, they retain many of the characteristics which are useful for force-directed graph layouts. A Riemannian manifold M has the property that for every point $x \in M$, the tangent space T_xM is an inner product space. This means that for every point on the manifold, it is possible to define local notions of length and angle.

Using the local notions of length we can define the length of a continuous curve $\gamma : [a, b] \to M$ by $\text{length}(\gamma) = \int_a^b \|\gamma'\| dt$. This leads to a natural generalization of the concept of a straight line to that of a *geodesic*, where the geodesic between two points, $u, v \in M$ is defined as a continuously differentiable curve of minimal length between them. These geodesics in Euclidean geometry are straight lines, and in spherical geometry they are arcs of great circles. We can then define the distance between two points, $d(x, y)$ as the length of the geodesic between them.

Hyperbolic Geometry: Hyperbolic geometry is particularly well suited to graph layout because it has "more space" than Euclidean geometry – in the same sense that spherical geometry has "less space". Unlike in Euclidean geometry, where the relationship between the radius and circumference of a circle in two-dimensional geometry is linear with a factor of 2π, and constant in a spherical geometry, in hyperbolic geometry the circumference of a circle increases exponentially with its radius. The applicability of this geometric property to graph layout is well-illustrated with the example of a tree. In hyperbolic space, it is possible to layout a tree structure with a uniform distribution of the nodes and with uniform edge lengths despite the fact that the number of vertices at a certain depth in the tree increases exponentially with the depth.

In order to visualize a layout in hyperbolic geometry it is necessary to map the layout into the (2D) Euclidean geometry of a computer monitor. There are various ways to do this. The two most common methods are Poincare disk and Beltrami-Klein projections. In both of these cases the hyperbolic space is mapped onto the open unit disk. To obtain such projections it is necessary to distort space, which in these cases takes the form of compressing the space near the boundary of the unit disk, giving the impression of a fish-eye view. This naturally provides a useful focus+context technique for visualizing the layouts of graphs. The method used in `graphael` is the Poincare disk. This model preserves angles, but distorts lines. A line in hyperbolic space is mapped to a circular arc which intersects the unit circle at

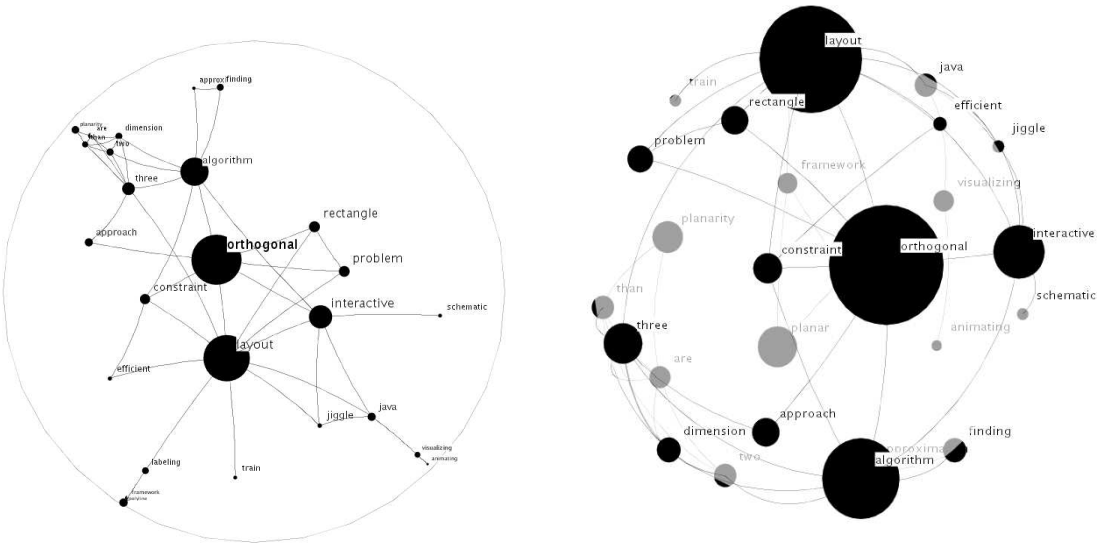

Fig. 2. Layouts of a title-word graph, obtained in Hyperbolic space and in Spherical space. The graph has 27 nodes and 50 edges and the nodes correspond to title-words from papers in the 1999 Graph Drawing conference. The size of a node is determined by its frequency and edges are placed between two nodes if they co-occur in at least one paper.

right angles.The Poincare disk progressively distorts the graph view as we move away from the center of projection. In Fig. 2(a) we show an drawing of a graph layed out in hyperbolic geometry and displayed in two-dimensional Euclidean space.

Spherical Geometry: Using the same ideas, we can generalize force-directed methods to spherical space. Spherical geometry, like hyperbolic geometry, has a constant curvature and the equations for mapping to and from the tangent space can be calculated analytically. Each point on a sphere is given a longitude and latitude. The sphere can then be embedded in three-dimensional Euclidean space by a simple parameterization. In Fig. 2(b) we show a drawing of a graph layed out in spherical geometry and displayed in three-dimensional Euclidean space.

Multi-Scale Graph Drawing in Non-Euclidean Space: Since we are able to utilize mappings between Riemannian points and its tangent space to use existing force-directed methods for graph layouts, we can also generalize the multi-scale method for drawing large graphs to non-Euclidean spaces. The multi-scale method has three main stages. The first stage is the filtration stage. Calculating a desirable filtration is completely dependent on the graph itself and no modifications are necessary. The next two stages are the initial placement and the refinement stages. Since the refinement stage simply uses force-directed force functions, and we are now able to use the same force functions in Riemannian space, the only stage for which we need further consideration is the initial placement stage.

In the initial placement stage we place each vertex one at a time in the barycenter of its neighbors. In Euclidean space we simply take the average of each dimension and place the vertex at that point. For non-Euclidean points we use the mapping to and from a tangent space. Specifically, we map the non-Euclidean points that correspond to the location of the neighbors to a tangent space. From there we are able to calculate the barycenter. We map that back into the non-Euclidean manifold and place the node at that location.

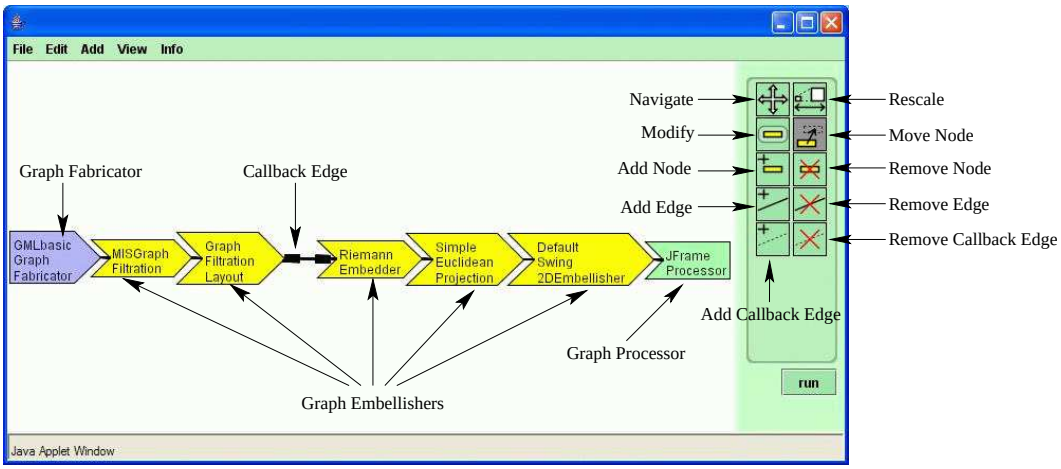

Fig. 3. A screenshot of the `graphael` CF-graph.

2.6 Graph Editor

There are several ways to experiment with the `graphael` system. Loading one of the sample graph files, loading a new file, or creating a new graph. When a user wants to create a graph manually, they have the option of using a basic graph editor that can be accessed from within `graphael`. The editor is simple, but useful for users that do not have data to generate a graph from. The graph editor is especially helpful in the testing of new components, since simple cases can be modeled easily in the editor. With a point/click/drag user interface, nodes and edges can be added, deleted, or moved. While editing a graph, node positions are stored so that the user can visualize it, but when the graph is input to `graphael`, the node positions are stripped off as the layout algorithms ignore initial positions.

3 Control Flow Graph

Here we describe another novel features of `graphael`, the Control Flow (CF) graph[2]. The CF-graph allows the user to put together different combinations of layout algorithms, projections and visualizations, while offering a visual representation of the derived graph's production process; see Fig. 3. CF-graphs contain CF-nodes (such as layout algorithms) that act to generate and refine a derived graph, and CF-edges that represent the channels of input and output, internally passed between the CF-nodes.

Users create graphs by adding new CF-nodes and connecting them with edges. Once a complete chain of appropriate CF-nodes has been completed, the user can push the "run" button to activate all the graph fabricators (described below). Modifying how an existing CF-graph produces derived graphs can be done in one of two methods. The first is to manipulate the composition of the CF-graph. Users can then easily modify a final, derived graph by pulling out certain CF-nodes and replacing them with others that will act upon the derived graph differently. The second method is to change the internal properties of the production units that the CF-node represents.

[2] Note: For clarity we shall call a Control Flow graph a CF-graph and a graph produced by a CF-graph a "derived" or "production" graph.

3.1 CF-Nodes

There are three different types of production units: fabricators, embellishers, and processors. Each of them is briefly described below.

1. *Graph Fabricators:* These are graph production units that take no input from other CF-graph entities. They act as the starting point for the generation of derived graphs since are not dependent on a derived graph as input. Many of the current graph fabricators available in `graphael` create the most raw form of a derived graph (i.e., vertex and edge declarations) by reading input files.
2. *Graph Embellishers:* These are methods that require a single, derived graph element as input from within the CF-graph and output a newly augmented, derived graph. In many cases, graph embellishers are used to add or modify properties of the derived graph they receive as input. For example, an embellisher could take a weighted graph as input and produce a weighted graph in which the color property of the heaviest nodes makes them stand out.
3. *Graph Processors:* These methods are the final stage in any CF-graph, since they do not pass a derived graph to other production units. Typically, graph processors output the final derived graph in the form of a picture, or an output file.

3.2 Callback Edges

In a CF-graph with normal edges, once a production unit is finished with its input, the graph is passed to the next unit until it reaches the end of the CF-graph. However, there are cases when this is not desirable. If we wish, for example, to show a graph layout in a series of iterations (as opposed to just the final product), we would require the use of `graphael`'s callback edges. These edges allow the source CF-node to suspend its execution and pass the graph to the remainder of the CF-graph, starting at the target of the callback edge. Once this finishes, execution would resume where the source CF-node left off. Callback edges can be identified as thick, dotted lines.

Using callbacks, we were able to implement features such as animation. Specifically, a layout that needs to iterate over the graph multiple times can suspend itself to let the resulting graph from each iteration reach the processor and be displayed on the screen. After the processor finishes, the layout runs the next iteration.

3.3 CF-Node Property Management

Whereas the panel in Fig. 3 offers different ways to manipulate the CF-graph, the individual CF-nodes can be manipulated as well. Recall that one of the two ways to modify how a derived graph is produced is to change CF-node properties. While we do allow for customized property managers, we have implemented an automatic GUI generator to minimize the amount of work required to make additions to the `graphael` library. The GUI generator is implemented using Java's reflection capabilities. This allows `graphael` to dynamically examine methods and data members of Java classes that have been added to its library. The system detects which properties can be modified by looking for a pair of *getter* and *setter* methods that meet the following conditions: (1) the names of the getter and the setter methods must be prefixed by 'get' and 'set', respectively; (2) the names of the getter and the setter methods must be identical, with the exception of the 'get' and 'set' prefix; (3) the return value of the getter method and the first parameter of the setter methods must be of the same type. Fig. 4 shows the GUI for one of `graphael`'s CF-nodes.

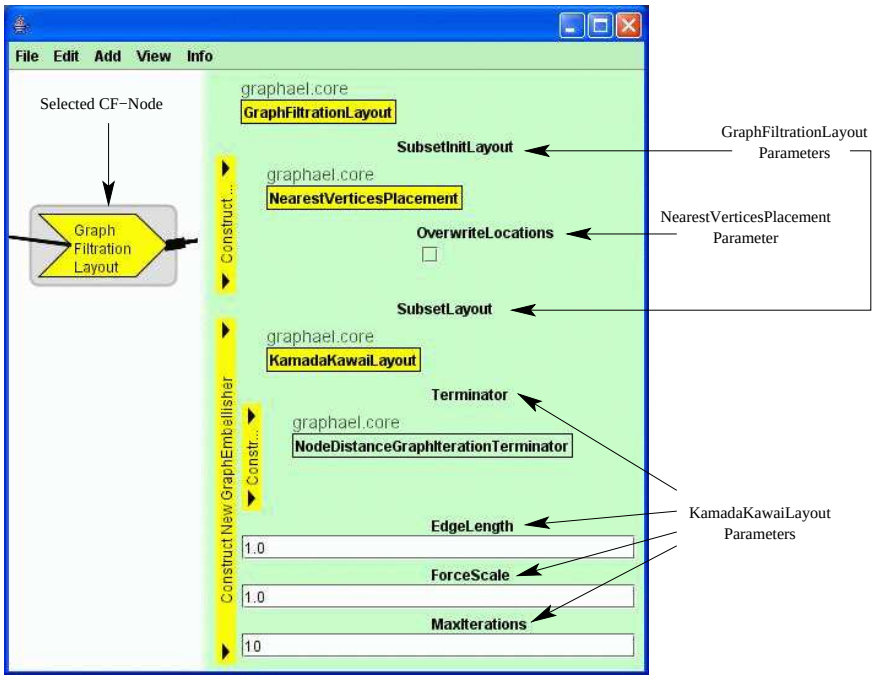

Fig. 4. Screenshot after selecting the GraphFiltrationLayout embellisher using the manage properties tool. Property manager GUIs such as the one shown here are automatically generated in `graphael`.

4 Implementation

The `graphael` system has been implemented using Java 1.4.2. This choice was made largely because the object oriented paradigm is well-suited for our plug-and-play and extensibility design requirements. Java's library also provides a mature code-base from which we derived many application components. For example, most of the `graphael` view is based on Java's Swing package. Using Java also made the conversion of `graphael` to an applet a straightforward task, helping achieve our design goal of portability.

Our system can be used as visualization platform for problems that generate graphs as output. For example, a database that produces graphs, such as the one described in [4] can be coupled with `graphael` to provide visual interaction with the graphs. Conceptually, the coupling is summarized in Fig. 5.

The `graphael` system currently supports the standard GML (graph markup language) file format. Simple modifications to the standard format accommodate node-weights and edge-weights, as well as timeslice information. More images from the system are shown in Fig. 6-8.

References

1. D. Auber. Tulip - a huge graph visualization framework. In M. Jünger and P. Mutzel, editors, *Graph Drawing Software*, pages 105–126. Springer-Verlag, 2003.

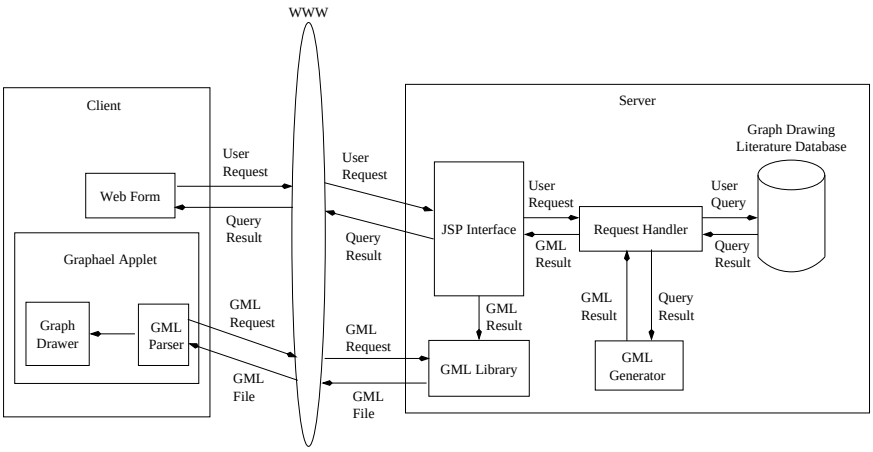

Fig. 5. The architecture of `graphael` in conjunction with `GraphAEL`.

2. V. Batagelj and A. Mrvar. Pajek - analysis and visualization of large networks. In M. Jünger and P. Mutzel, editors, *Graph Drawing Software*, pages 77–103. Springer-Verlag, 2003.
3. T. Dwyer and P. Eckersley. Wilmascope - a 3d graph visualization system. In M. Jünger and P. Mutzel, editors, *Graph Drawing Software*, pages 55–75. Springer-Verlag, 2003.
4. C. Erten, P. J. Harding, S. G. Kobourov, K. Wampler, and G. Yee. Graphael: Graph animations with evolving layouts. In *11th Symposium on Graph Drawing*, pages 98–110, 2003.
5. C. Erten, P. J. Harding, S. G. Kobourov, K. Wampler, and G. Yee. Exploring the computing literature using temporal graph visualization. In *Visualization and Data Analysis. Proceedings of SPIE 2004*, To appear in 2004.
6. T. M. J. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Softw. Pract. Exper.*, 21(11):1129–1164, 1991.
7. P. Gajer and S. G. Kobourov. Grip: Graph drawing with intelligent placement. *Journal of Graph Algorithms and Applications*, 6(3):203–224, 2002.
8. M. Jünger and P. Mutzel, editors. *Graph Drawing Software*. Springer-Verlag, 2003.
9. T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Inf. Process. Lett.*, 31(1):7–15, 1989.
10. S. G. Kobourov and K. Wampler. Non-euclidean spring embedders. Technical Report TR2004-012, University of Arizona, 2004.
11. J. Lamping, R. Rao, and P. Pirolli. A focus+context technique based on hyperbolic geometry for visualizing large hierarchies. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 401–408. ACM Press/Addison-Wesley Publishing Co., 1995.
12. S. B. Seidman. Network structure and minimum degree. *Social Networks*, 5:269–287, 1983.
13. R. T. Stina S. Bridgeman, Ashim Garg. A graph drawing and translation service on the www. *International Journal on Computational Geometry and Application*, 9(4–5):419–446, 1999.
14. R. Wiese, M. Eiglsperger, and M. Kauffmann. yfiles - visualization and automatic layout of graphs. In M. Jünger and P. Mutzel, editors, *Graph Drawing Software*, pages 173–192. Springer-Verlag, 2003.

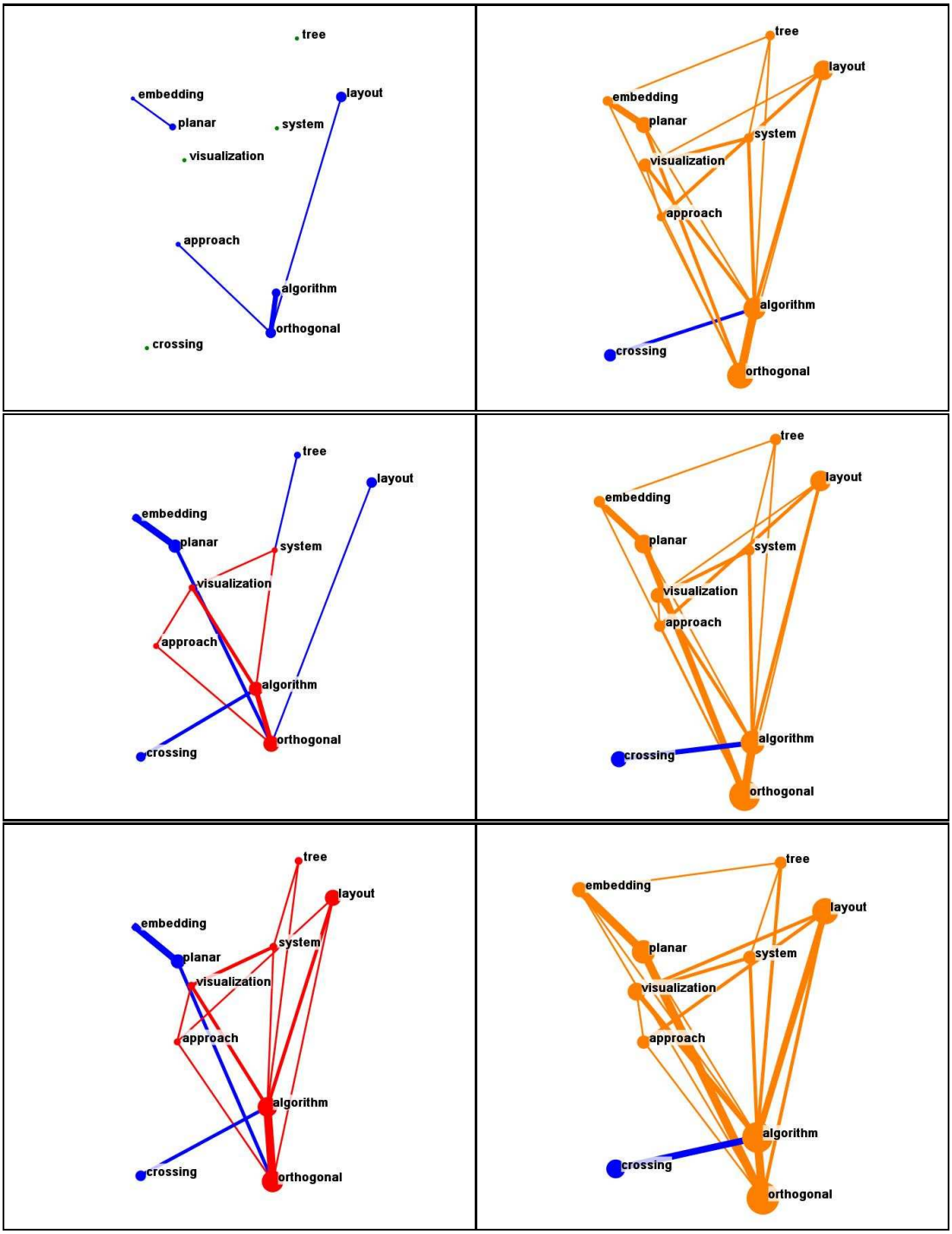

Fig. 6. Cumulative title-word graph from the Graph Drawing Symposium from 1998 to 2003; the sequence goes from top to bottom, left column first.

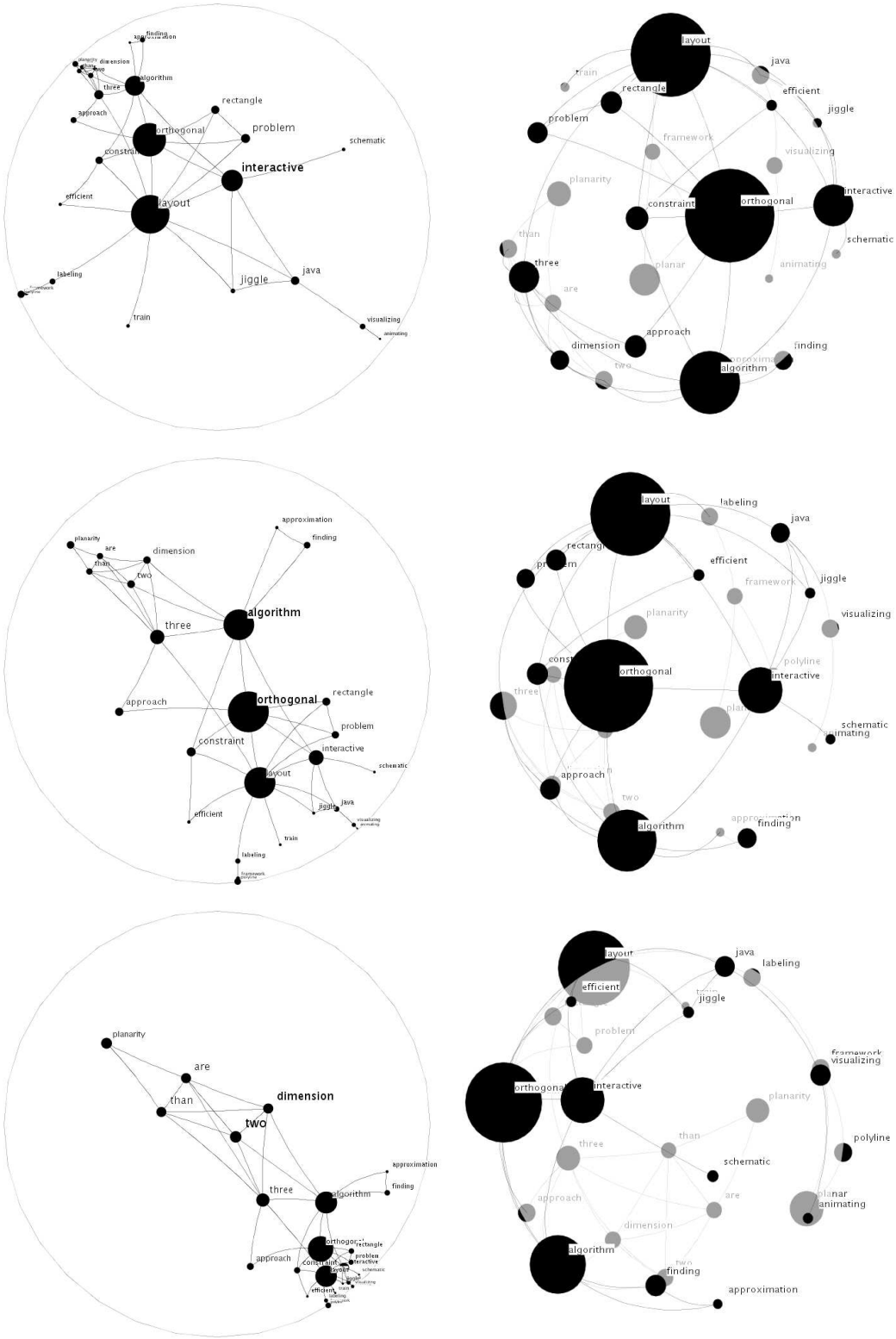

Fig. 7. Layouts of the title-word graph with different centers of attention in hyperbolic space (left) and spherical space (right).

Fig. 8. Core views of the graph drawing literature collaboration graph: core 13, core 11, and core 7.