An Interactive Small World Graph Visualization

Stephen F. Ingram*
University of British Columbia

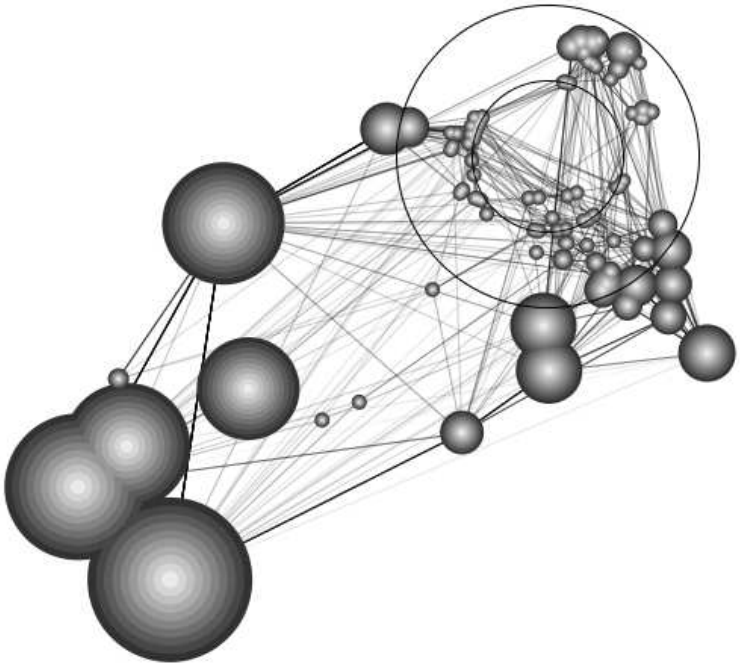

Figure 1: Generic Screenshot of graph exhibiting small world structure using the small world prefuse components.

ABSTRACT

Small world graphs are a common class of graphs whose topology exhibits structure characteristic of both order and randomness. This report summarizes the implementation of a 3D visualization technique designed to exploit the topological features of small world graphs using the prefuse toolkit. We successfully transition the algorithm from 3 to 2 dimensions. We then explain and resolve several key ambiguities in the algorithm. We compare the performance on a variety of small world graphs. Finally, we analyze the utility of the visualization through several scenarios of use.

Keywords: Small World Graphs, Graph Visualization, Graph Drawing, Prefuse Toolkit, Clustering

1 INTRODUCTION AND PREVIOUS WORK

Imagine the task of classifying all node and link graphs according to their topology on the spectrum between complete regularity and complete randomness. How would one definitively classify an arbitrary graph? One strategy is to note the characteristics of extreme graphs. Regular graphs, such as grids, exhibit a cliquishness. That is, a given node's neighbors share many of the same neighbors. This generally has the consequence of long path lengths between arbitrary nodes, where you must travel through a long chain of cliques to find the shortest path between these nodes. On the other hand, completely random graphs exhibit the opposite behavior. Their neighbors do not exhibit the same cliquishness, forming random relationships across the network. This has the consequence of making the average path length between arbitrary nodes very small. These two respective measures are technically called Clustering Coefficient and the Characteristic Path Length and we can use these to form a two-dimensional spectrum of graph topologies.

In the part of the spectrum where path length is small and clustering his high, in other words, where graphs exhibit characteristics of both random and regular graphs, is the class of graphs with "small world" structure [11]. These small world graphs have recently emerged as interesting because they model systems, such as neural networks or social networks, that are hard to analyze using conventional means. This resistance to conventional analysis make small world graphs a good target for visualization. It is well known that the human visual system can discern patterns that are exceedingly difficult for machines to perceive.

However, like conventional analysis, graph layout has suffered from a deficit of techniques to exploit the unique topology of these graphs. Standard physical layout seeks to minimize the variance in edge lengths, leading to a uniform graph layout. But small world graphs should contain a variety of edge lengths, with shorter lengths for edges in tight clusters and longer lengths for random edges between these clusters. Recently Noack [9, 10] has devised a physical algorithm to calculate such an embedding. Dubbed LinLog, the layout's edge lengths are proportional to their coupling.

Armed with such a layout, van Ham and van Wijk [2] introduced a series of interactive techniques specifically designed for small

*e-mail: sfingram@cs.ubc.ca

world graph visualization. They use the geometry computed by the Noack layout to define a dendrogram of nodes where the nodes in the dendrogram represent graph nodes, clusters of graph nodes, or clusters of these clusters. Next, they define a smooth method of interpolating between nodes in a given subtree called the degree of abstraction. Using this semantic/geometric interpolation, they create a visual analogue of the cluster dendrogram and use fisheye focus-plus-context techniques to allow a user to explore the cluster hierarchy. By using a fisheye lens to smoothly reveal the subclusters in the graph a user can supposedly understand more about the latent communities in the small world graph.

This report discusses the implementation and the results of the above techniques in a freely available graph visualization toolkit called prefuse [3]. Because this toolkit factors so integrally into the report, we give a brief description of it in the next section.

2 PREFUSE

Prefuse is a software framework for information visualization in Java. As such, it is designed to dramatically simplify the creation of visualizations. A potential visualization builder need only to compose their product into a pipelined series of prefuse components. In order to make this simple, prefuse partitions the workflow of a visualization into the following abstract regions:

- **Data**–The nodes and edges of the graph itself and any supporting data.
- **Visual Form**–The data structures that define only those items which are scheduled to be drawn on the screen.
- **View**–This is the frame buffer and any user controls.

Likewise, prefuse partitions its actual software components into the following categories

- **Entities**–These objects make up the *Data*.
- **Filters**–These map *Entities* to *VisualItems*.
- **VisualItem**–These are the objects that compose our *Visual Form*.
- **Layouts**–These determing the positions/sizes of *VisualItems* on the screen.
- **Renderers**–These map *VisualItems* to the *Display*.
- **Displays**–This makes up the *View*.

Prefuse makes some assumptions about what kind of visualizations can be made. First, it assumes that the underlying data is a graph. This does not conflict with the goals of the project. Second, prefuse is built with Java2D and therefore assumes that the desired visual output is two dimensional. While [2] uses three dimensional data to achieve its visual goals, we argue below that we can achieve the same or similar visual goals in two dimensions. Therefore we believe prefuse makes an ideal candidate for our visualization.

In the following section we describe the design of our software into prefuse-style components.

3 DESIGN

The following is a table listing the components created in our implementation and the categories to which they belong:

Category	New Component Name
Entity	Cluster
Filter	LinLogAction, AverageLinkCluster
Layout	DOALayout
VisualItem	VoroNode
Renderer	VoroNodeRenderer, TubeRenderer

Here we define a new default Entity, Cluster, for our data to exhibit the hierarchical properties of the technique. Next we define the Noack layout, or LinLog, and the dendrogram clustering algorithm as Filters instead of Layouts because we are only determining the positions of Clusters, which are not visible. Determining what is actually visible depends on the local Degree of Abstraction for a given region of the screen. Thus we create a DOALayout to map Clusters to what we term VoroNodes (the choice for this name will be explained in section 4.2.1). Finally we devise a pair of Renderers to handle drawing our nodes and edges in a way compatible with [2].

4 IMPLEMENTATION

This section details relevant issues, some unexpected, encountered in the development of the above prefuse components.

4.1 PolyLog Layout

Van Ham [2] mentions that unoptimized force model simulators exhibit $O(N^3)$ complexity. Using such a force model on graphs of hundreds or thousands of nodes therefore seems prohibitive, but is justified as being a "preprocessing step." While this is true that it will not effect the interactivity of the visualization, it hurts the ability for a single user to run a series of visualizations. It also prohibits the existance of a graph node creation/deletion tool for the visualization because the simulation would need to be run after each graph edit.

Luckily, Noack has provided a Java-based implementation of the LinLog model with a Barnes-Hut optimization that brings the complexity down to $O(N^2 Log(N))$. Our task was thus significantly scaled down to merely building a prefuse Filter wrapper class that translates between underlying graph data structures.

4.2 Rendering

This sub-section details some of the significant problems we encountered in translating the visual model of our target to Java2D. We admit to relying on the use of clever, but inelegant hacks to achieve comparable visual results.

4.2.1 Nodes

One of the major techniques in [2] is the use of visual abstraction to convey clusters and subclusters in the graph. One such instance of this is in the rendering of nodes as spheres. For example, if the degree of abstraction for a cluster is high, it appears onscreen as a single sphere. As the degree of abstraction decreases for this cluster, it is decomposed into subclusters. Because this decomposition is interpolated smoothly, the geometrically clustered spheres still appear similar to the parent cluster, however the intersections of the spheres create a "shaded Voronoi diagram" revealing the substructure of the sphere.

It was our original intention to explicitly compute the Voronoi diagram of the intersecting subspheres. This is what lead to our sublclass of prefuse *VisualItem* to be called *VoroNode*. We first compute the Delaunay Triangulation of all the clusters and take the dual of this to yield the Voronoi regions of each cluster. Then, using Constructive Area Geometry, we subtract from a node A's circular shape the shapes of its neighbors B minus A's Voronoi cell. We

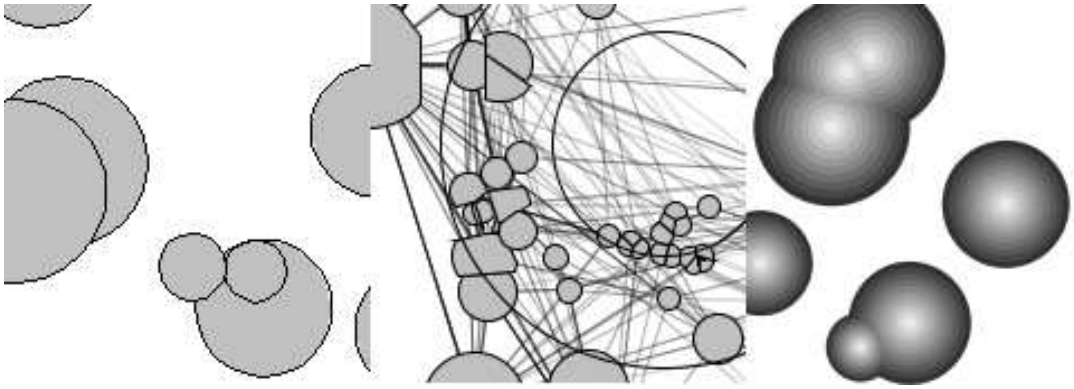

Figure 2: Examples of porting node drawing to 2D. From left to right we have no modification, computing Voronoi regions (with bugs), and using a simple hack.

then render the new shape for *A* using traditional 2D draw routines. Thus we capture all the information in the 3D diagram.

Unfortunately there are two things wrong with this approach. First, the computation of the triangulation is slow unless significant optimization is used such as in [1]. Second, and more importantly, this technique breaks the rendering pipeline of prefuse. Prefuse follows a policy of zero object creation in the rendering phase of the pipeline. Unfortunately Java2D requires objects of type *Shape* be passed to its draw routines. Thus, to draw our VoroNodes, we must create new shapes for every frame of animation. But this will eventually provoke the ire of the Java Virtual Machine which then forces the garbage collector to reclaim these un-used shapes and temporarily destroys our interactivity. This left two options:

1. Devise a pre-allocated Shape back-end to our renderer
2. Try something simpler

Opting for number 2, we used the following, simpler strategy. Reasoning that graphics cards are quite fast enough and our biggest enemy is the garbage collector, we simply *fake* 3D nodes. That is, each of our nodes is just a set of concentric circles whose color follows a gradient from black to white. The key is to draw a single layer of all the nodes before drawing the next layer. This entailed augmenting prefuse with a new class of *Renderer* we call a *MultipassRenderer* and its corresponding *MultipassDisplay* to replace the standard prefuse *Display*. These two classes extend the graphical possibilities of prefuse for applications outside of this visualization.

4.2.2 Edges

The original paper employs 3D edges to indicate edge orientation more clearly in the case of edge occlusion. Likewise, shorter edges are given priority over longer edges by keeping the volume of edges constant, so short edges are fatter and longer, less significant edges are barely perceptible. In the transition to 2D, edges suffer from a similar problem as nodes, but resolve to a less graceful conclusion.

Our first attempt to create 2D edges with shading was to simply use rectangles filed with a gradient paint. This strategy ignores the zero object creation policy and suffers from severely degraded performance. Unfortunately we cannot simply resolve this problem as we did with nodes. While creating a series of concentric polygons may work visually, unlike circles, polygons are anisotropic. This means we need to create a new polygon to align with each edge axis. There unfortunately may be as many axis angles as there are edges, so we cannot pre-compute our edge polygons without serious memory commitment. We thus must resort to a much more complex edge rendering system to maintain the same effect.

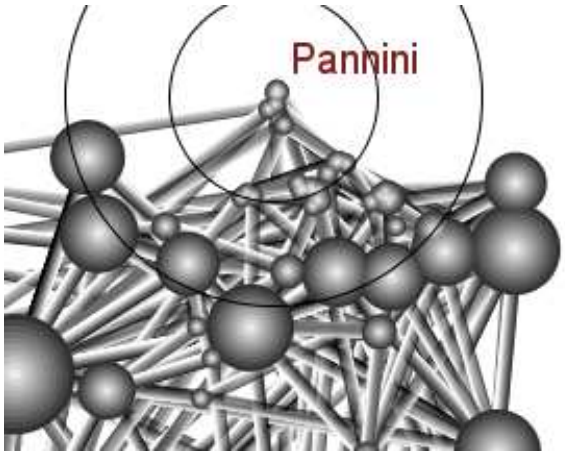

Figure 3: The first attempt at mimicking 3D edges was unsuccessful.

Rather than take this drastic measure, we make a compromise. We forgo the orientation-revealing advantages gained by shading and simply draw the edges as lines using the established prefuse edge renderer with a single modification; the transparency of a given edge is proportional to its length. Thus the shortest edge is the most opaque and the longest edge is the most transparent. This compromise is in the same spirit as the "constant-volume" tubes in the original paper, but much simpler and quicker to render.

4.3 Clustering

Clustering decides which two nodes become merged into a single cluster. This is vital to the visualization for at least two reasons. First, it is by the smooth, animated composition/decomposition of clusters that community structure is conveyed to the user. If the clusters decompose in an unintuitive way, we lose one of the major objectives of the visualization. Thus clustering should favor neighbors that are geometrically closer. Clustering is also important in that it allows us to only process the set of visible nodes. Depending on the constant degree of abstraction (explained in section 4.4) we will only process $O(Log(N))$ nodes, keeping the visualization running at an interactive rate.

We don't simply cluster the two nearest neighboring clusters. This technique, called Minimum Link, causes what Van Ham calls "long chains". Long chains are undesirable because they substitute the exponential decomposition of clusters for linear decomposition

of clusters. In a nutshell, this means that some neighbors in a community will emerge on the graph far after some of their neighbors in the same community. To best achieve an exponential decomposition, we use Average Link clustering which entails clustering two neighboring clusters whose *average* shared edge length is the smallest. This technique is $O(N^2)$ in the worst case.

We further consider an alternate clustering algorithm by Newman [8] designed specifically for detecting communities in small-world graphs. This algorithm, a greedy approximation of the more robust [7], seeks to maximize a modularity measure of the graph. This algorithm only regards the topology of the graph, but can be modified for weighted graphs. In this algorithm, a large edge weight is considered good, therefore we weigh the edges from our LinLog layout inversely proportional to their length before feeding them to this algorithm.

We implement both of these algorithms using an efficient sparse matrix formulation. In practice we use the sparse matrix facilities provided by Colt, a set of Open Source Libraries for High Performance Scientific and Technical Computing in Java.

4.3.1 Alternate Layout

Since the Newman clustering algorithm relies solely on the topology of the graph and not its geometry, we briefly consider the idea of using this clustering and deriving a $O(Log(N))$ layout solely from the resulting dendrogram. This layout recursively and proportionally subdivides space until we are at a leaf node. The dendrogram layout is summarized by the following pseudocode

```
Position dLayout( Node, Rect ) {
  if |child.children| > 0
    foreach child in Node.children
      childrect = Rect / |child.children| // unique
      pt += dLayout( child, Rect / |child.children| )
    end
    return pt / |child.children| // average of children
  else
    return random point in Rect
  end
}
```

4.4 Layout

The definition of degree of abstraction is simple, but we have found using this measure to determine the position of points more complex and subtle than van Ham betrays in [2].

The measure of the degree of abstraction or DOA is simply how high up the tree one goes when considering which nodes to draw. A DOA of 1 indicates that we are at the root of the tree and a DOA of 0 indicates we are at the leaves of the tree. Because we cluster using distances and distance increases monotonically as we go further up the tree, van Ham describes an interpolation parameter λ based on a varying DOA and the proportional distances between parent and child. That is, he describes a method to smoothly interpolating the positions and sizes of nodes based on some intermediary $DOA \in [0,1]$.

If DOA is constant, then we don't experience anything interesting. Our visualization thus depends on *varying* the DOA over the graph interactively. This is accomplished with a 2 part, mouse-directed lens. For the initial part of the lens of distance r to the mouse cursor where $r_{DOA} > r > r_0$, we set the DOA as a linear function of r or

$$DOA = DOA_{constant} \frac{r - r_0}{r_{DOA} - r_0}$$

If $r > r_{DOA}$, then we set $DOA = DOA_{constant}$. If $r < r_0$ then $DOA = 0$. Now that we know our DOA for a given position on the screen, computing the position of a node should be straightforward. Unfortunately there are three subtleties that complicate this process.

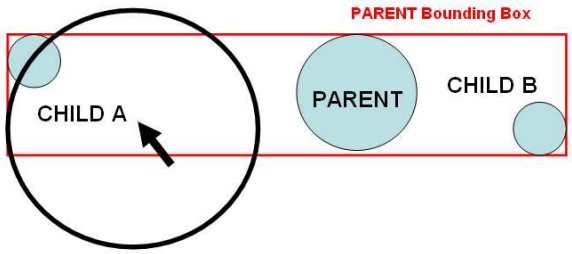

Figure 4: An illustration of the first subtlety of using degree of abstraction in practice.

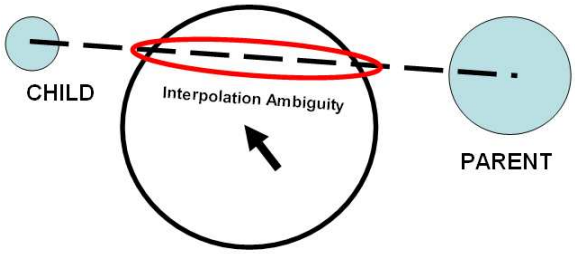

Figure 5: An illustration of a situation where the exact position to interpolate is ambiguous when using the degree of abstraction in practice.

The first subtlety comes from computing the position for a node. As was previously mentioned, we want to ideally consider only $O(Log(N))$ nodes in our layout algorithm to maintain interactivity. Therefore it seems like we only want to consider those clusters whose DOA is greater than the DOA at their position and cease recursion if this isn't true. However, following this policy will yield an incorrect layout. This is because a parent node may be far enough away from its children that it will have a DOA at its position that is less than the DOA of its children who are under the lens. If we stop processing nodes at the parent node, we will never learn that its children are under the lens.

We resolve this problem by computing the bounding box for a node. The bounding box of a given cluster is defined by the extreme coordinates of its leaf nodes. We then can make intelligent processing decisions based on whether the DOA varies over the contents of a node's bounding box. Note that we could achieve even better performance by using the convex hull of leaf nodes instead of a box, but neglected to implement this due to time constraints.

The second subtlety comes from smoothly interpolating a position. Consider an example where a parent and one of its children are farther than the diameter of the lens apart from each other. Now imagine that the lens passes over the region along which the child's position is interpolated from its parent but never intersects either the parent or its child's coordinates. It is unclear from [2] what to do in this case because the path of interpolation intersects the DOA lens twice. We suspect using a large enough lens eliminates this problem, but we have encountered it in practice several times.

We resolve this problem by hacking the DOA function a little bit. Instead of computing the DOA as a function of the distance between the focus of the lens to the coordinates of a node. We consider the DOA instead as a function of the distance between the focus of the lens and the parent's bounding box. In practice this yields good enough interpolation while resolving the complexities

of interpolation as simply as possible.

The third subtlety comes from using the clustering distance measures to determine our interpolation parameter λ. Sometimes there is a discrepancy between geometric cluster distance, which is a function of the averages of the leaf node positions, and average link cluster distance which is a function of the averages of the edge lengths between two clusters. If this discrepancy is large, which happens regularly at high degrees of abstraction, then lambda changes very quickly and the interpolation is not smooth. We left this problem unresolved, but can imagine a technique to correlate these two distances.

5 RESULTS

This section details the results of our implementations. We first consider the performance issues of the software by looking at runtimes and responsiveness. Next we try to gauge the overall usability of the software by walking through some scenarios of use.

5.1 Performance

5.1.1 Runtime and Response

We have analyzed the performance of our system on four small world graphs of differing sizes. The nodes of the first graph represent airports in the United States and an edge is constructed between nodes only if there is a direct flight between them. This graph contains 332 nodes and 2126 edges. Our second graph is taken from [2] and contains 500 nodes and 2486 edges. Our third and fourth graphs were culled from the Internet Movie Database. In these two graphs each node is an actor and an edge is constructed between two actors if they have appeared in 10 movies together. The smaller of these has 419 nodes and 5651 edges, while the larger has 3648 nodes and 24987 edges. This last graph will hereafter be called the huge graph. All of our experiments are performed on a machine running Microsoft Windows XP with 1 gigabyte of RAM and a 2.27 GHz Pentium 4.

The following table contains the average runtime in milliseconds of the LinLog layout on our 4 different graphs.

Graph	Run Time (ms)
Airport	10141
Artist	13140
IMDB (small)	11563
IMDB (huge)	206485

This table contains the average runtime in milliseconds of the Average Link clustering algorithm on the graphs.

Graph	Run Time (ms)
Airport	281
Artist	360
IMDB (small)	484
IMDB (huge)	31797

Note the quadratic leap in the runtimes of our linlog and clustering algorithm even when our edges have increased by only a factor of 12.

The visualization runs interactively with nodes and edges for all but the huge graph. The huge graph runs interactively only if we elide edges, otherwise it is 10 to 20 seconds between frames. This can be attributed to only considering $O(Log(N))$ nodes per iteration, while we do no such handling of edges. Also note that these graphs contain many more edges than nodes. This means we reduce the cost of rendering significantly per iteration when we only consider the nodes. Because LinLog places clusters close to each other, it is sometimes "good enough" to only consider rendering nodes.

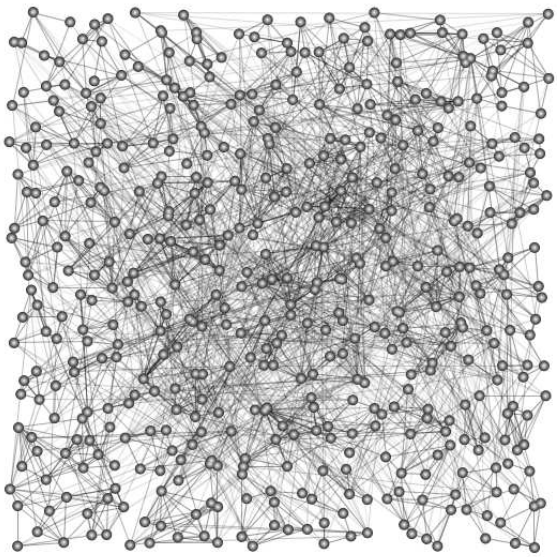

Figure 7: A screenshot of the alternate layout with constant zero degree of abstraction.

5.1.2 Newman Clustering and Alternate Layout

Newman Clustering achieves similar runtimes to Average Link. However, it is ineffective when used in conjunction with LinLog layout. If we use weighted graph edges inversely proportional to their distance, then Newman behaves very similar to Minimum Link clustering. That is, we note long chained clusters with a linear decomposition rate. If we use an unweighted graph then we achieve an exponential decomposition, however, by ignoring the geometry of LinLog, clusters are sometimes formed that violate the DOA's monotonically increasing distance assumption. The end result is a loss of smooth cluster interpolation as we approach a cluster with our DOA lens.

The performance of our Alternate Layout is obviously superior to LinLog in time complexity. However it results in a uniform, unintuitive distribution of nodes. We find that only the clusters on the lowest level of the tree end up close to one another, but other, unrelated clusters can as well. There are not enough divisions of the plane to separate different communities from each other.

5.2 Usability

In this section we run through three scenarios of use, one for each of the sources of our datasets.

5.2.1 Scenario 1: Airline Clusters

In this scenario we consider a user in a more remote part of the United States, like as Nome, Alaska. The goal of their using the visualization is to explore different travel routes from Nome to the rest of the country. The user first loads the graph in the visualization and waits for LinLog to finish. Their first task is to locate their city in the graph. Since Nome is a unique city, only connected to local Alaskan cities by air, the user first scans the graph for smaller clusters. They then examine these outlying clusters for Alaskan cities. Sure enough, when the user finds Kodiak, Nome is the next

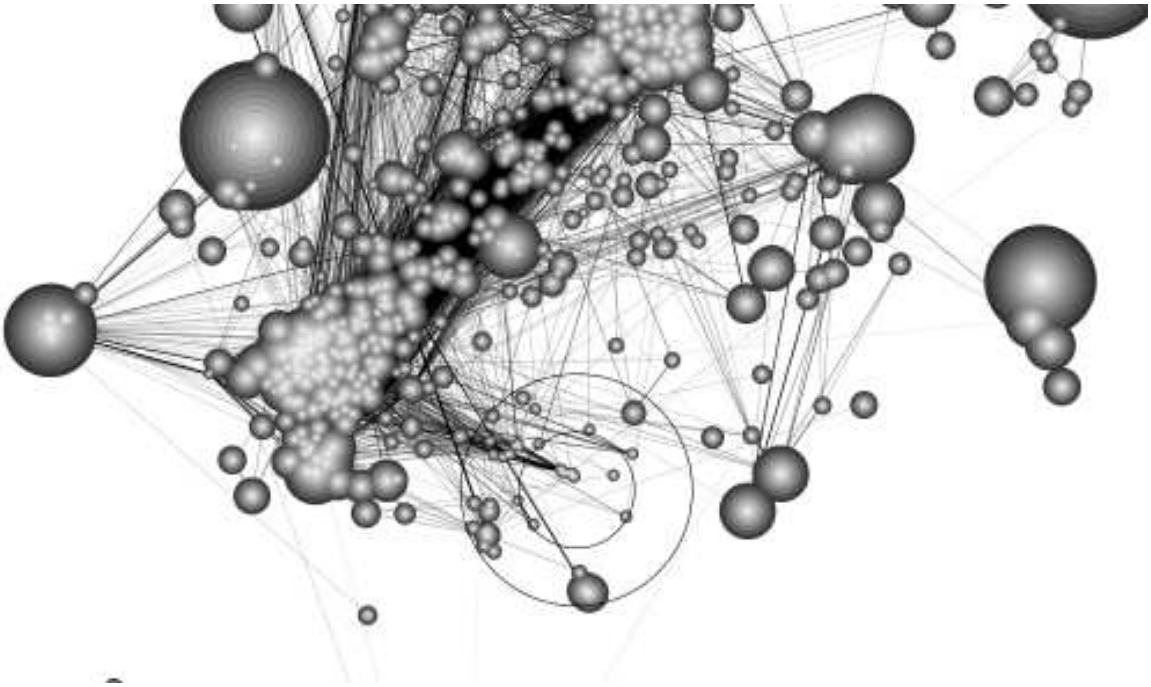

Figure 6: Resulting layout with huge graph.

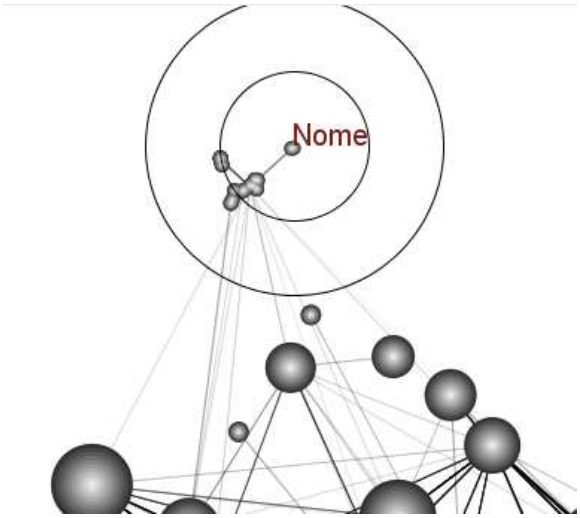

Figure 8: A screenshot at the beginning of Scenario 1.

cluster over. From here we need to find flights to other airports. We note that it is somewhat difficult to differentiate between Nome and its neighbors, so we increase our geometric fisheye distortion. Now we see there is a flight from Nome to a variety of cities like San Francisco and Salt Lake City and smaller cities on the Aleutian Islands.

Running through this scenario we note that following long edges isn't very easy using the system. It is simple to follow longer edges through sparse regions, however edge-occlusion become a problem if they cross tightly coupled regions of the graph, even with fisheye distortion. This scenario obviously conflicts with the maxim of de-emphasizing longer edges.

5.2.2 Scenario 2: Artist Clusters

Suppose we are writing a comprehensive survey on artistic movements. We want to understand communities of artists and how they have related through history. Perhaps we can use this visualization as an exploratory study tool. After loading the artist graph and computing a clustering, we choose a random cluster on the graph and advance our lens over it. We encounter a cluster of artists named, "Schwitters", "Duchamp", "Man Ray", etc. all members of the Dadaist movement. Also sharing in the nearby community are "Jasper Johns", "Rauschenberg", both Pop Artists. What do Pop Art and Dada have in common? We look them up in Google and find, "Like dada, its European forerunner, [Pop Art] challenged the concept of art by elevating the vulgar and ordinary to the status of art object." Well what do these specific artists have to do with one another? Well, digging a little deeper reveals that one of Rauschenberg's works, *Trophy II*, is dedicated to Marcel Duchamp. With only a few glances at the graph we're on to something.

In uncovering general trends, or subtle relationships, the tool seems well suited. As long as the absolute fidelity of the visualization, especially in regards to edges, isn't required then information is gleaned quickly and is likely to provoke further exploration.

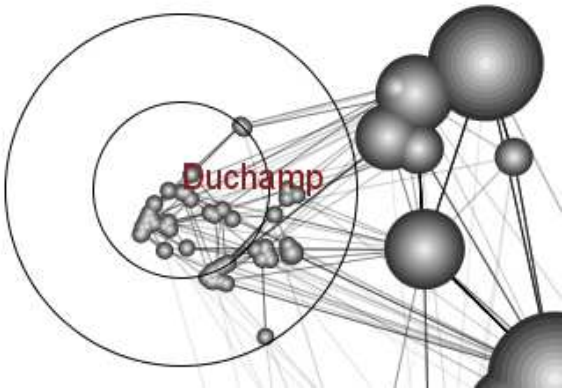

Figure 9: A screenshot of Scenario 2.

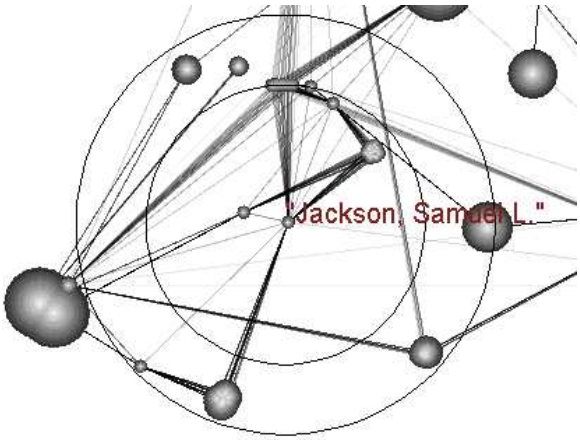

Figure 10: A screenshot of Scenario 3.

5.2.3 Scenario 3: Actor Clusters

In this scenario we are interested in Hollywood stars. We want to know what kind of networks are formed by actors who appear in movies together. As in the other datasets we load the graph and move the lens around and examine labels until we find something familiar. Perusing the grpah, we note that this data has a different structure than the previous graphs. Here the nodes form tight clusters with far fewer inter-cluster edges. What intercluster edges there are, are connected to a single actor. This tells us that stars are more likely to appear in the same movies with actors the same small clique. When they appear in movies with people outside this clique, then it is with a group of only three of four people who "bridge" different cliques. For example, we find Tom Cruise who is grouped with many other big names like Denzel Washington, Bruce Willis, etc. all in the same community. The only links outside of this group are to three actors, Arnold Schwarzenegger, Harrison Ford, and Samuel Jackson. Further inspection reveals more of these standalone, connecting actors.

There is more to be said about this kind of network, but using the visualization for only a few minutes says quite a bit about professional behavior in Hollywood. It is likely that a static graph could yield similar information about the social network. However the interactive techniques keep the frame rate manageable when using focus plus context techniques like the fisheye lens.

6 Discussion

The interactive techniques implemented in this report are a useful and potentially scalable technique quickly learning the structure of small-world graphs. They provide a good union of algorithm and visualization, matching the graph clustering abstraction with visual clustering while exploiting the exponential structure of the dendrogram to make significant performance gains in interactivity. The technique certainly has visual appeal, important in making an exploratory interface inviting. A simple desire to watch the clusters fall apart gives us a good overview of the structure of the latent communities in the graph as a side effect.

As it currently stands, the algorithm is only potentially scalable. As graph sizes increase, it becomes necessary to elide edges which results in a loss of important information. Also, edge drawing in particular is still difficult to read. The technique is very sensitive to the parameters of the graph, thus it is important to maintain controls to adjust the parameters (like fisheye radius) of the visualization accordingly. Labelling is a persistent problem inherited from other graph visualizations. Here it is especially difficult because individual nodes agglomerate as regions of the graph become more abstract, making difficult to remember which node we were originally inspecting.

When implementing the visualization, we found it best to err on the side of speed and simplicity. For example, in switching to 2D, creating a complex Voronoi shapes using fancy geometry became a liability. Results are much nicer using a simple, but effective hack. Likewise, in resolving unaddressed issues with the degree of abstraction layout, using simple approximations of the original intent of the paper yielded nearly indistinguishable behavior.

7 Future Work

This visualization has enormous potential, but first a few important obstacles need to be removed. The most important problem involves edges. An edge drawing routine that obeys the same complexity as the node drawing routine will greatly improve the scalability of the visualization. We propose an adjacency matrix that somehow intelligently spans the hierarchy in a scalable manner. This result is crucial for making this technique relevant to more diverse, larger datasets.

Another extremely crucial improvement is in implementing a labelling scheme for clusters. It is extremely tedious to mouse-over every leaf node in a cluster. One simple solution is to allow a user to "lasso" a cluster with the cursor and then write all its subcluster labels to an adjoining list box. Or cluster bounding boxes from section 4.4 could be used as a background area for drawing the names of the nodes in that cluster. After using the visualization for a while, anything seems better than mouse-over.

A strong addition to the visualization would be an intelligent use of color to disambiguiate clusters. In [2], they use *a priori* classification of nodes to determine the colors of individual clusters. A much better approach for a wider class of graphs is to compute colors using only the topology of the clustered dendrogram.

While classified as preprocessing steps, it would be nice to see advances in reducing the complexity of both the layout and clustering algorithms. Perhaps, one day, some of the force model optimizations made by Jourdan [6] for MDS can applied to the LinLog model. Likewise, perhaps fast, sparse clustering techniques like Nonnegative Matrix Factorization can be applied to adjacency matrices in such a way that exploits community structure [5]. Although it produced an inferior layout, computing layouts from dendrograms produced by [8] still seems like a promising direction. Perhaps the addition of a little complexity at a time to the layout can edge results towards being competitive while maintaining a superior runtime.

The visualization can be applied to any small-world graph, but it would be nice to see it applied in specific domains augmented with some useful functionality. For example, with some better labeling, it could potentially be used as a digital library browse interface [4] where each cluster is a "subject" in a document corpus. Such hierarchical examples abound.

Finally, a consequence of implementing the visualization in prefuse is that it is readily deployable to other projects. The source of this project will be distributed for use in other info-vis projects. Alternatively an applet that accepts URLs of graph files can be readily constructed for non-expert users to play with the visualization.

REFERENCES

[1] Guibas, L. and Stolfi, J., Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams, ACM Transactions on Graphics, 4:74-23, April 1985.

[2] Frank van Ham and Jarke J. van Wijk, Interactive Visualization of Small World Graphs, In NFOVIS '04: Proceedings of the IEEE Symposium on Information Visualization (INFOVIS'04), 2004, 199–206.

[3] J. Heer, S. K. Card, J. A. Landay, prefuse: a toolkit for interactive information visualization. In CHI 2005, Human Factors in Computing Systems, 2005.

[4] A. Krowne and M. Halbert, An Evaluation of Clustering and Automatic Classification For Digital Library Browse Ontologies, 2004, report, available at http://www.metacombine.org/reports/research/metacombine_a1_draft_20040704.pdf.

[5] D. D. Lee and H. S. Seung, Learning the parts of objects by nonnegative matrix factorization, Nature, 401 (1999), 788-791.

[6] Fabien Jourdan, Guy Melancon, Multiscale Hybrid MDS. IV London 2004, 8th International Conference on Information Visualization, London, UK, IEEE Computer Society, 2004, pp. 388-393.

[7] M. E. J. Newman and M. Girvan, Finding and evaluating community structure in networks, Phys. Rev. E 69, 2004, 026113.

[8] M. E. J. Newman, Fast algorithm for detecting community structure in networks, Phys. Rev. E 69, 2004, 066133.

[9] A. Noack. Energy-Based Clustering of Graphs with Nonuniform Degrees. Accepted for publication in: Proceedings of the 13th International Symposium on Graph Drawing (GD 2005, Limerick, Ireland, Sep. 12-14)

[10] Andreas Noack. An Energy Model for Visual Graph Clustering. Proceedings of the 11th International Symposium on Graph Drawing (GD 2003, Perugia, Italy, Sep. 21-24), LNCS 2912, 425-436.

[11] D. J. Watts and S. H. Strogatz, Collective dynamics of 'small-world' networks, Nature 393, 1998, 440–442.