WiGis: Web-based Interactive Graph Interfaces
A Scalable Approach

Brynjar Gretarsson* Svetlin Bostandjiev† John O'Donovan‡ Tobias Höllerer§

Department of Computer Science
University of California, Santa Barbara.

ABSTRACT

Traditional network visualization tools inherently suffer from scalability problems, particularly when such tools are interactive and web-based. In this paper we introduce *WiGis* –Web-based Interactive Graph Interfaces. *WiGis* exemplify the first web-integrated tool for visualizing large-scale graphs natively in a user's browser at interactive frame rates with no discernible associated startup costs. Drawing from the traditional web paradigm of client-server interaction, we apply a brokering technique between a client browser and a remote server for interactions with large graphs. We demonstrate fast, interactive graph animations for up to hundreds of thousands of nodes in a browser through the use of asynchronous data and image transfer. Empirical evaluations show that our system outperforms traditional web-based graph visualization tools by at least an order of magnitude in terms of scalability, while maintaining fast, high-quality interaction.

Index Terms: K.6.1 [Graph Drawing]: Constraints—Force Directed Algorithms; K.7.m [Web Technology]: Client-Server Algorithms—Scalability

1 INTRODUCTION

The proportion of people conducting their daily business on the web is increasing rapidly. As a result of this unprecedented increase in user-provided content, we are now faced with an information overload problem where users are presented with increasingly large amounts of data, such as search results, social web updates, news feeds, and email. The same problem applies across a broad range of disciplines which produce graph and network data, from business (stock-market / customer-preference data) to medical (protein-structure / gene sequence data) and many more. The traditional mechanisms for data exploration on the web are largely text based. Text queries which return a ranked result list have proven demonstrably useful as a means to retrieve information, however there are many limitations to this approach when it comes to understanding larger information spaces. We believe that the current practices of information search, exploration and discovery on the web can be greatly improved through the use of dynamic, interactive data visualizations.

The focus of this paper is a system based around a novel technique for interactive visualization of large graphs in a web browser without the need for plugins or special-purpose runtime systems. By enabling users to visualize and interact with large scale network data, we provide a "big picture" of the information space they are dealing with. Through interaction, users can mold large scale data into their own mental model, which serves as a useful starting point for more fine-grained analysis. Our system enables a top-down approach to exploration of large information spaces.

We use the term *WiGis*, or Web-based Interactive Graph Interfaces, for our system, which is a new addition to the set of currently available tools providing "visualization as a service" [4]. *WiGis* can support interactions at fine granularity even with very large graphs of up to a million nodes (and 2 million edges). This is achieved by leveraging a novel client-server technique for processing computationally expensive graph layouts across different machines through asynchronous data transfer. For the end user, the result is a seamless interactive browsing experience, presented natively in a client browser without the need for any external plug-ins. An advantage of the web-based design is that remote collaborative interactions with large graphs are facilitated. This paper describes and evaluates the new system with a range of test datasets. In addition, we present a comparative experiment in which our system scales by an order of magnitude above popular graph visualization systems in terms of number of nodes and edges visualized interactively.

The remainder of this paper is organized as follows: Section 2 outlines our aims for a scalable web-based graph visualization tool. A critical analysis of current relevant work in the area of large graph visualizations with a focus on web-based approaches is presented in Section 3. Section 4 describes the architecture of our *WiGi* tool in terms of design and implementation of interaction algorithms and the client-server computational model. Section 5 discusses an empirical evaluation of our visualization tool (and its component parts) in terms of scalability and speed with respect to popular graph visualization tools. Section 6 contains a brief discussion of the benefits and limitations of our technique as well as various deployments of the system. The paper concludes with a summary of the main contributions.

2 VISUALIZATION AIMS

At the outset of this project we defined a list of ten salient properties which we believe are central to a good web-based interactive visualization tool for large graphs. The list also forms the key implementation goals for our *WiGi* tool:

Scalability - The system should be capable of interactive visualization of very large graphs (ideally millions of nodes) with a wide variety of edge connectivity in reasonable time.

Interaction - A smooth and scalable interaction experience should be supported in real time.

Synchronization - To achieve scalable interactions, the system cannot rely fully on potentially variable client processing capabilities. To harness the power of distributed computing for scalability, synchronization must be applied in some way between local client and remote server processes.

Accessibility - The system should be platform independent and accessible natively in all major web browsers with no external plug-in (e.g: Java Applets, Flash etc.). In addition to the end-user benefit, this eliminates the security risk of third party plug-in code, a concern for any large organizations with sensitive data.

Ease of Use - The system should provide easy mechanisms for users to import and visualize their own data.

Initialization - Initial load times for the tool itself and for visualized data (irrespective of scale) must be short. This should be similar to load times for a standard html page, which precludes most plug-in web applications such as Applets or Flash. The system should be accessible from a single click.

Collaboration - The system should facilitate data communication, provenance, and collaborative data interaction.

Layout - A range of layout algorithms should be available for use on various graph types.

Navigation - The system should provide intuitive navigation mechanisms, for example: overview/detail representations, zooming and panning.

*e-mail:brynjar@cs.ucsb.edu
†e-mail:alex@cs.ucsb.edu
‡e-mail:jod@cs.ucsb.edu
§e-mail:holl@cs.ucsb.edu

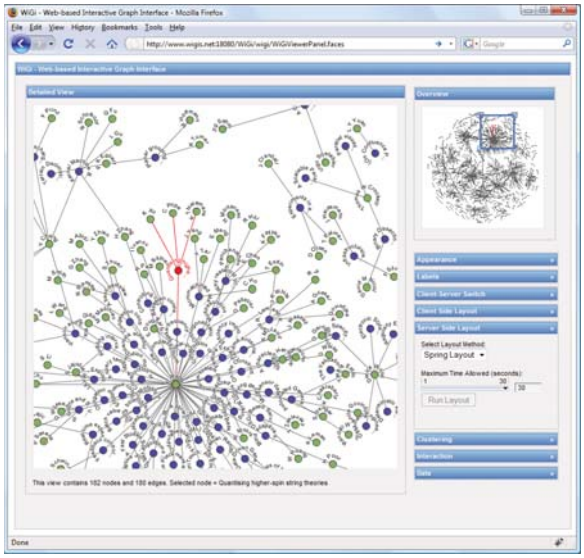


Figure 1: A screenshot of the *WiGi* tool displaying results expanded from the seed query "Graph Visualization" on the Citeseer dataset. Graph shows 1104 author and article nodes, with 1125 associations. An overview window of the entire graph is shown in the top right corner with the detail view highlighted by the zoom box.

Personalization - The system should have customizable look and feel. For example, functionality for varying node and edge colors, sizes, labeling styles etc.

In today's complex information spaces it is important to provide a user with a good understanding of the broader data-universe that they are interested in. By providing good interaction mechanisms for large scale data, we enable a tailoring to occur which reflects a user's underlying mental model of the broader data [19]. For example, during observations of user interactions with citation data in our *WiGi* tool, it was noted that users tended to pull the authors and publications that they were familiar with away from the main body of the graph, allowing them to see movement of affiliated nodes. A layout algorithm alone can severely influence the users conception of the broader information space. Interaction allows the user to "mold" the data into their own mental model [19], resulting in better understanding (and recollection) of the data. Figure 1 shows a screenshot of the tool[1] visualizing an authorship network of the graph visualization community.

3 Background

Background for this work is divided into two areas. Firstly, a discussion and analysis of graph drawing literature is presented. This is followed by a discussion of the techniques and technologies used in web-based graph visualization, including an analysis of existing applications.

3.1 Graph Visualization

Ware et al. [15] highlight the fact that visualization allows for faster and easier interpretation of data than text-based presentation because it leverages preattentive processing, which is lacking in text-based communication modes. Ware's argument is reinforced in a recent work by Kim et al. [17] which presents a user study indicating that text-based search can be both frustrating and imperfect. Hachul et al. present a comparative analysis of commonly used graph-drawing techniques in [12]. Among these are force directed layouts– one of the most commonly used techniques for graph drawing. The approach was first developed in 1984 by Eades [7] and later refined by Fruchterman and Reingold in [10]. Over the years, many adaptations and refinements of Eades' original force-directed model have been applied, among these multi-level [11, 13] and constraint-based techniques [5, 6]. Much research has been conducted on large scale graph visualizations, e.g. [14, 8, 20, 12], although mainly focused on desktop applications. On the web, there have been some efforts at scalable graph visualizations, e.g. [4, 25, 22], but in general these do not scale past thousands of nodes.

3.2 Web-Based Graph Visualization

Traditionally, graph visualization applications have been desktop based. For example, Cytoscape [20], Pajek [2], and some implementations of Tom Sawyer Visualization [21]. Over the past few years, increased web-accessibility and bandwidth improvements have triggered a general shift towards rich internet applications (RIAs) capable of providing interactive and responsive interfaces through a web browser. This shift has a potential benefit for resource-intensive graph visualization, and applications which take advantage of the rich-internet paradigm are beginning to emerge for visualization of graph and network data. Examples of such applications include Touchgraph [22], Tom Sawyer Visualization [21] and IBM's Many Eyes [4].

3.2.1 Thick v/s Thin Clients

RIAs can be loosely classified into thick and thin clients. A thick client typically provides rich functionality that is largely independent of a central server with the majority of processing done on the client, whereas a thin client requires constant communication with a server to provide functionality. Client-based visualization [25] can be considered a thick-client solution since data is downloaded from a server and the visualization and rendering are done at the client side. The popular graph visualization tool Touchgraph Navigator [22] is a good example of a client-based tool, since it processes graph interactions locally in Java. Visualizations can be created on a remote server and passed across a network to the client. This is referred to by Wood et al. [25] as "server-based" visualization, and is an example of a thin client solution since the network connection is essential for the system to function. The *WiGi* system utilizes both client-based and server-based visualization techniques and the key innovation is the way the system can automatically and transparently switch between the two modes while allowing smooth interaction in both.

3.2.2 Plug-in v/s Native Applications

RIAs can be further classified based on the manner in which they are deployed. Many RIAs are implemented using some form of browser plug-in, for example Java Applets, Adobe Flash or Microsoft Silverlight. The majority of graph visualization tools available on the web are plug-in based, e.g. [22, 4]. There are some fundamental drawbacks with the plug-in approach however. Firstly from a scalability perspective, a plug-in based RIA is limited to the processing capabilities of the plug-in itself, e.g. the default memory limit for Java Applets is usually around 60-90 MB. Secondly, from a security and accessibility perspective, plug-ins usually have startup costs and need to access client resources, making them a potential security threat. The alternative approach to plug-in based RIA implementation is to provide functionality natively in the browser through a combination of DHTML and AJAX. Examples of native RIA's include Google Maps and the JSP and ASP.net implementations of the graph visualization tool, Tom Saywer Visualization [21]. For the *WiGi* system, we are concerned about scalability, accordingly we opted to design it as a native RIA to avoid the inherent limitations and security drawbacks of browser plug-ins. The following section explains how scalability is achieved in our system through a novel client-server architecture for interactive graph visualization.

4 Architecture

The main contribution of this work is a new and scalable technique for providing smooth interaction with very large graphs in a user's web-browser. There are a variety of existing solution attempts to this problem, for example [22], [4], and [21]. However, most current solutions implement some form of plug-in based thick client to manage graph models, for example, client-side Applets or installer applications. We believe that such approaches lack flexibility since they rely too heavily on the processing capabilities of the client, which can be unreliable and vary greatly. The one exception we found was Tom Sawyer Visualization [21] which has implementations that run natively in a browser. According to our understanding, however, the system does not allow for interaction with graphs

[1] www.wigis.net

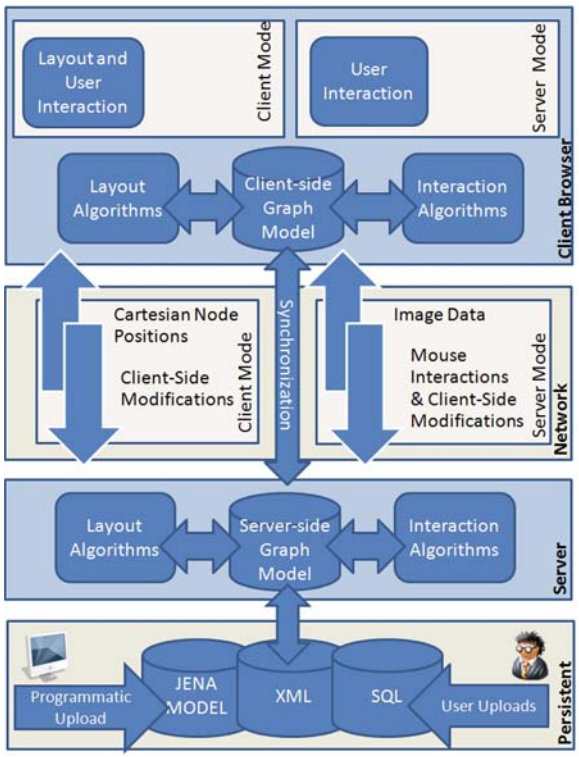


Figure 2: A scalable web-based architecture for interactive graph visualization, as used in the WiGi system

larger than a few hundred nodes without zooming into a smaller subgraph. Even when interaction is allowed, the image of the graph is not updated continuously. Instead, updates are performed on mouse release. In contrast, our system allows fine grained interaction with graphs of any size at all zoom levels. Figure 2 describes the novel, lightweight and flexible architecture we use for interactive graph visualization. This architecture does not rely heavily on a client's resources, and requires only a basic browser with no external plug-ins.

Figure 2 applies the client-server paradigm to the problem of large-scale interactive graph visualization. The key innovation of this architecture is the use of fully synchronous graph models that exist on both a client and a (potentially very high-powered) remote server, allowing for completely transparent hand-offs between client-side and server-side computations and renderings. We now discuss these two distinct modes of operation, and follow this with descriptions of each of the component layers in Figure 2.

4.1 Visualization Modes

4.1.1 Client-Mode

When a graph in the viewing window (shown in Figure 1) is sufficiently small, all layouts, interactions and renderings are performed in the client browser. This can be either the whole graph or a zoomed in part of a larger graph. The top layer in Figure 2 represents the browser, which contains a model of the graph, referred to as the client-side model. As this model is updated by JavaScript layout and interactions, its state is asynchronously transferred to a remote server, which updates a server-side graph model accordingly (shown in the third layer from top in Figure 2). Rendering is performed by JavaScript using SVG for edges (replicable in VML for Internet Explorer) and HTML image tags for nodes, since this was the best performer out of a multitude of image and div based rendering options over a combination of metrics in our preliminary tests. In "client-mode," the *WiGi* can still make use of server-side functionality such as clustering for instance. The client simply calls the remote function on the server through AJAX, the server runs a process, updates its model and passes it to the client.

The following list outlines the motivations for, and benefits of using client-side graph processing:

Smooth Interaction - Client side computations provide fast interactions for small graphs because there is no direct network overhead.

Network Independent - Client-side processing does not need a fast network to function well, and can even function in an off-line state.

Easy on Server Resources - With a potentially large user base, *WiGis* can be heavy on server resources. Utilizing client resources wherever possible eases the load on a centralized server or server set. We note that in client-mode, the remote server still holds a model of full the graph, so only CPU load is reduced, as opposed to memory.

4.1.2 Server-Mode

For large graphs, the *WiGi* automatically switches into "server-mode". In this mode, all computations for both layout and interaction are processed on the remote server. Instead of passing a graph model back to the client browser for reconstruction, the server generates a bitmap image of the updated graph. This image is passed across the network and rendered in the browser. Swapping from client to server mode is a *seamless transition* for the end-user, with no jumpiness or image differences. While in server-mode, interaction is facilitated by capturing mouse motions on the image of the graph using JavaScript. Mouse motions are passed asynchronously to the server and the interaction/layout algorithms are triggered on the server-side graph model based on the new input. The server computes an updated graph, renders it, and sends an image of the rendered graph back to the client. The key success of our tool lies in the fact that this *entire process occurs at interactive speeds* giving very smooth desktop-like interactions with very large graphs.

Our system achieves update rates of 10 frames per second for graphs up to the order of 10K nodes, while graphs of the order of 100K nodes are rendered at approximately 1 to 2 frames per second (c.f. results in Section 5.6.1). Theoretically, with sufficient hardware resources on the server-side, the upper bound for the number of nodes a *WiGi* can usefully display in an interactive fashion approaches the pixel resolution of the client display.

Server-side operation of the *WiGi* tool can loosely be compared to a Google-Maps interface with the difference that transmitted images are not static or pre-defined. Instead, images are computed on-the-fly based on a combination of user input and the existing graph state. The following list shows the benefits and drawbacks of using the server-side approach for large graph computation.

Scalability - Client side graph visualizations generally fail as the graph size approaches thousands of nodes and edges. Using our server-side technique we can interactively visualize graphs of up to 1 millon nodes natively in the browser.

Remote Resources - Server-side processing extends the power of the browser well beyond the resources of the local machine by using a thin client implementation.

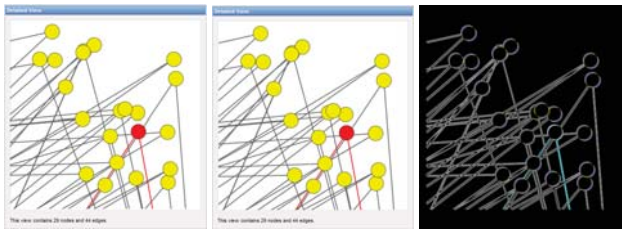
Bandwidth Limitation - Server side graph processing relies heavily on network resources, and can perform poorly on slow networks. While many universities operate very fast connections, home and wireless broadband connections typically range from 64 kb/s to about 1 Mb/s. Our evaluations show that the network overhead becomes negligible for graphs of over 100 thousand nodes.

4.2 System Architecture Layers

Following is a description of the architecture based around the four layers in Figure 2 from top to bottom. These layers represent physical locations or communications between them, as opposed to the previously discussed client-mode and server-mode, which are modes of *operation* spanning across all layers.

4.2.1 Client Browser Layer

The top layer in Figure 2 represents a web-browser running on a client machine. Depending on the current mode, the browser holds either a JavaScript model and an SVG/HTML visualization of the graph (client-mode) or a single bitmap image of the graph in its current state (server-mode). The browser contains a JavaScript implementation of a selected layout algorithm and a selected interaction algorithm, *both of which are scripted "replicas" of server-side algorithms.* Depending on the current operation mode (client or server), the browser layer communicates either graph model data or mouse interaction data across the network to the remote server.



(a) Client-Mode (b) Server-Mode (c) Difference Image

Figure 3: Seamless transition between client and server renderings. (a.) Client-Mode. Rendering and layout done with DHTML and SVG. (b.) Server-Mode. Rendering and layout done in Java on a remote machine. (c.) Difference image between a and b.

4.2.2 Server Layer

The server *layer* is the "powerhouse" of the *WiGi* tool, where most of the heavy processing occurs while in server mode. The server holds a model of the full graph (in memory if possible), a set of graph layout, clustering, and interaction algorithms (currently implemented in Java, but extendable to any language). The key concept of the architecture is that the client layer mirrors the server graph model to the capacity of its available resource. Again, depending on the scale of the visible part of the graph and resources available on the client, the server either accepts mouse interaction data (server-mode) or an updated graph model (client-mode) from the client browser. In return, the server communicates either graph model data or GIF images back to the browser depending on the current mode of operation. The graph model on the server is always kept in synch with the client model through AJAX updates.

4.2.3 Network Layer

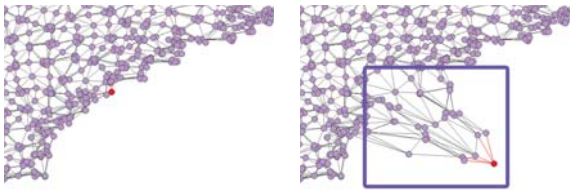
The network layer in Figure 2 represents the communication between the server and client layer. Depending on the mode of operation, image data and interaction data (server-mode) or graph model data (client-mode) is sent across the network to maintain synchronization between the client and server layers.

4.2.4 Persistent Layer

Graph data can be uploaded to the system through a web interface by users or programmatically by other systems to add interactive visualization capabilities to them. User uploads are provided in several common formats including XML, GraphML and a simple CSV representation. Regardless of the original source, all data is converted to an XML representation and read into the graph server. The persistent layer of the WiGi system is kept modular to allow data from a broad range of sources to be plugged in easily. For instance, current data sources include citation data from a publication search tool and dynamically generated topic models from New York Times articles.

4.3 Client-Server Synchronization

Each algorithm in the client browser is coded in JavaScript to exactly mimic the corresponding server-side java version. The algorithms are designed to be identical with one exception: the client side algorithm operates only on a subgraph containing all visible nodes and their neighbors. This constraint is necessary because of the scalability limitations of JavaScript and the potentially limited resources on the client machine. Since we must use different platforms and implementation languages, there are also small differences between the resultant graph layouts. However, in most cases these differences are too small to be discerned visually by the end-user. Figure 3 depicts a sample graph visualized in (a.) client-mode and (b.) server-mode. It is clear from Figure 3 that the two representations are very similar, although they might be misaligned by one pixel as a result of floating point errors in the conversion between different coordinate systems. Other minor differences exist in the anti-aliasing of the lines and circles. The system automatically switches from server-mode to client-mode when zoomed into a sufficiently small part of the graph and back to server-mode when zoomed out to a larger portion of the graph.



(a) Before (b) After (effect param=3)

Figure 4: Example of interpolation algorithm (a.) Before interpolation (b.) After interpolation, with the selected subgraph highlighted

4.4 Layout

The *WiGi* architecture supports plugging in a range of different layout algorithms and the user can select them from a drop down box. The focus of this paper is on scalability with regard to interaction, and accordingly, details of various layout algorithms are not included here. For the purpose of our analysis we use an efficient implementation of a simple force-directed graph layout algorithm [7] [10].

4.5 Interaction

Interaction with large graphs is not as straightforward as interaction with a small number of nodes, since moving one node at a time can be very time consuming when molding a layout of thousands of nodes. Moreover, the commonly used rectangular area selection of multiple nodes that happen to lie close to each other is not ideal when interacting with large graphs because the selected nodes do not necessarily have meaningful associations with each other apart from proximity as computed by a layout algorithm.

4.5.1 Interpolation Method

To facilitate user interactions with a large graph, the system uses a fast and scalable interpolation method originally developed by Trethewey and Höllerer [23] for use in a desktop application. Additionally, the method gives the user an intuitive control over the layout of the whole graph by the means of interpolation. A target node is selected and moved with the mouse pointer, while other nodes are moved by a relative amount which is proportional to the inverse of their graph distance from the selected node. Accordingly, when applied to a full graph, the farthest node from the selected node remains stationary.

Time complexity of the interpolation method is measured in two parts. Firstly, upon *selection* of a target node, a breadth first search is performed to find the graph distance from the target node to all other nodes in the graph. This results in a time complexity of $O(|V|+|E|)$ for the node selection process, where V is the set of vertices and E is the set of edges. The second part of the interaction process is mouse *movement*. When dragging is performed, each node is visited only once and its new position is computed, resulting in a $O(|V|)$ time complexity. It is important to note here that the selection process occurs only once per click-and-drag interaction, whereas computations relating to mouse-movement occur on an ongoing basis as dragging occurs.

4.5.2 Effect Parameter

The original interpolation method [23] has been extended in our implementation to incorporate additional functionality. By allowing the user to specify the locality of the interpolation method via a distance parameter to the farthest affected node, the effect of user interactions can easily be varied from global to local. This enhancement is useful for molding large and complex graphs into more clear and comprehensible layouts. Figure 4 shows an example of the interpolation method running on a graph with effect parameter set to three. The affected subgraph is highlighted in Figure 4(b) and contains all nodes within a geodesic distance of three from the target node.

In the case where the effect parameter is less than the maximum geodesic distance between the selected node and any other node in the graph, the effects of moving a node are more local. The extreme case being when the parameter is set to zero and only the selected node is moved. Setting the parameter to a relatively small value reduces the depth of the breadth first search which is performed

during the selection process, and in turn the number of repositioned nodes during the movement process, resulting in a faster interaction. Setting the effect parameter to a relatively high value, i.e. higher than the maximum geodesic distance, makes it easy to interactively figure out and arrange connected subgraphs in a graph which consists of many disconnected components. On drag of a single node, all connected nodes will be moved by almost the same distance, making it easy to drag a connected component all at once. Based on our own usage and observations of others interacting with the system, we believe that this interaction technique allows the users to gain a thorough understanding of the data while molding their own mental model of the graph.

5 Evaluation

Now that we have described our technique for enabling interactions with large graphs on the web, we focus on an empirical evaluation of the technique in terms of speed and scalability. To properly evaluate our system we must break down the interactive visualization process into its component pieces and present evaluations of each individually, before testing the system as a whole. As a precursor to this, we define a test dataset of graphs at different scales and discuss their properties. All of the following experiments use the same test data. For the purpose of this evaluation we define three steps (potential bottlenecks) in our interactive visualization process:

Step 1: Rendering - Drawing graphs after a change has been made.

Step 2: Interaction - Capturing user input and calculating modifications to the graph.

Step 3: Network - Passing graph and/or image data across the network from a remote server.

After evaluating each step in isolation, we combine our results to produce our estimated overall time for the variety of graphs in our test suite. As a sanity-check this is then compared against recorded times for interaction with the system as a whole. A discussion of the relative impact of each step is presented. Our evaluation concludes with a comparison against three popular interactive graph visualization systems.

5.1 Setup

All experiments were performed on a 64 Bit Dell Inspiron 530 with an Intel Q9300 2.5GHz quad core processor, 8GB of RAM, an ATI Radeon HD 3650 video card and a serial ATA hard drive with 7200 rpm. The operating system was Windows Vista SP1. No other heavy processes were allowed to run during experiments. A Dell 24" UltraSharp flatscreen monitor with a refresh rate of 60Hz was connected with a DVI cable. Screen resolution was constant at 1920x1200 pixels for all experiments. Graph window sizes were kept constant at 600x600 pixels. This value was chosen because it will fit in most browser windows with 1024x768 resolution. Frame rates were recorded either by the *WiGi* tool itself or by FRAPS[2]. On our multi core machine, FRAPS did not introduce significant delays in any renderings. All web-based experiments were performed in Mozilla Firefox 3.0.3 and in Google Chrome 1.0.1 with no plug-ins or add-ons running. The *WiGi* system is Java-based and was hosted on a JBoss 4.2.2 server running on the same machine as the client browser. This was done to eliminate network overhead which allowed us to determine the theoretical network overhead of all connection speeds, based on the the exact size of the data which is passed across the network.

5.2 Description of Test Data

There are many possible approaches for testing a web-based system for large-graph layout and interaction. Our system works well with real world data, for example citation networks, computational provenance graphs and topical relations among newspaper articles. We have also applied our system successfully to specific graph types such as meshes, trees, highly connected and highly sparse data. For this paper we have chosen to perform our tests on "small world" data [1] [24], because it is abundant in the social web, financial, biological and many other naturally occurring networks [18]. Small world networks are connected graphs in which most nodes are not direct neighbors but can be reached in a small number of hops from most points in the graph. Our test data was generated using the Barabási-Albert (BA) Model [1] for creation of small world networks. The BA model uses preferential attachment [1] for the addition of new nodes. Table 1 describes our test graphs $G1,...,G6$. Graph size is increased exponentially from $G1$ (10 nodes, 20 edges) to $G6$ (1M nodes, 2M edges). To confirm the small world nature of our test data, the degree of connectivity versus number of nodes for all graphs was plotted on a log-log scale. This test produced linear trends in logarithmic space for all graphs, exhibiting the trademark power-law distribution of small world networks [1]. Our test data and graph analysis are available for download online[2].

[2]www.fraps.com

Graph	G1	G2	G3	G4	G5	G6
Nodes	10	100	1K	10K	100K	1M
Edges	20	200	2K	20K	200K	2M
Avg. Degree	2	2	2	2	2	2

Table 1: Description of generated small-world data

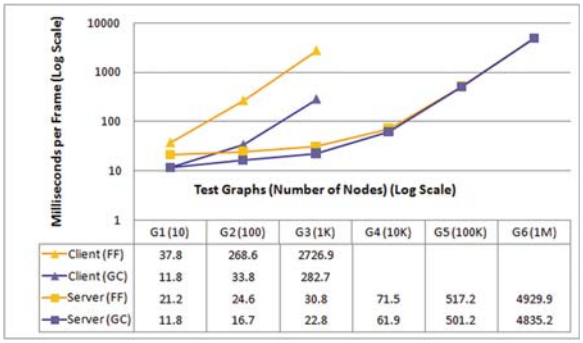


Figure 5: Results of the rendering experiment showing milliseconds per frame at various graph sizes.

5.3 Rendering

A potential bottleneck for our system occurs while re-rendering the graph after it has been modified. Modifications can happen either on client or server side, directly by the user, or by interpolation and layout algorithms. To evaluate rendering speeds in both client and server modes, each of our test graphs $G1,...,G6$ was rendered and the average time per frame was recorded. Graphs were redrawn at every frame and all nodes and edges were constrained to the viewing window for the entire test. Node positions were randomized prior to each run.

As described in Section 4.1.1, client-side rendering was performed by JavaScript using SVG for edges and HTML image tags for nodes. In server-mode, graphs were rendered into GIF images using Sun Microsystems Java2D graphics library from Java 6.0 and those images were passed to a browser on the same machine. Our standard setup (outlined in Section 5.1) was applied. For this test, edge width was kept constant at 1 pixel and node size was a constant 4x4 pixels.

Figure 5 shows the results of the rendering experiment. Since our test graphs increase exponentially in size, results are presented on a log-log scale. Note that on this scale, *small differences at upper parts of the graphs represent significantly larger differences on a linear scale*. The four plots in Figure 5 represent the client and server side frame rates in ms/frame for rendering of test graphs $G1,...,G6$ in both Mozilla Firefox (FF) and Google Chrome (GC). Despite what looks like a polynomial curve, the graph does show that both client and server methods eventually scale approximately linearly with number of nodes, however the client side method is notably slower than the server side, because the browser takes longer to update position data in the document object model as graphs size is increased. The smaller graphs create the curve effect because of various overheads, but importantly, the larger graphs $G4,...,G6$ show a linear trend.

In fact, the server side method performs better than linear, as the overhead of loading the image into the browser and displaying it is almost constant for all graph sizes. At $G1$ the methods have identical performance, while at $G3$ the difference is 260 ms for GC and approaches 3 seconds in FF. For small graph size, typically less

than 100 nodes, client side rendering takes about the same time as server side rendering, in the range of 0 to 260 ms/frame. The client-side approach could not scale past test graph $G3$ (1K nodes). This occurs either because the max number of SVG lines the browser is capable of rendering is exceeded and the page fails to load (FF), or because the browser slows to an unusable state (GC). Irrespective of these failures however, there is stronger motivation for using the server side method for large graphs: at about 1000 nodes it simply becomes more efficient to pass an image of the rendered graph across the network as opposed to sending raw node/edge data and rendering it on the client.

Client side rendering consists of two parts: drawing of SVG lines for edges and HTML image tags for nodes. In addition to the results in Figure 5, these two parts were tested individually. Averaged across all test graphs, line drawing took 53% and image repositioning took 47% of the total rendering time.

Clear performance differences were exhibited between browsers, Chrome was significantly faster than Firefox, which took an average of 6 and 1.3 times longer for client and server methods respectively across all test graphs. This result is as expected because more work is being done by the browser while in client-mode, and Google Chrome's V8 JavaScript engine is faster than the Firefox 3.0.X engine. Looking forward, improvements to JavaScript engines are underway in most major browsers, and we expect that our technique will perform better as faster engines are released.

This experiment shows that our technique is capable of rendering graphs in a browser in the order of hundreds of thousands of nodes and edges in a fraction of a second. This is clearly indicated by the results in Figure 5, which show that $G5$ (100K nodes) takes about half a second while $G6$ (1M nodes) takes less than 5 seconds.

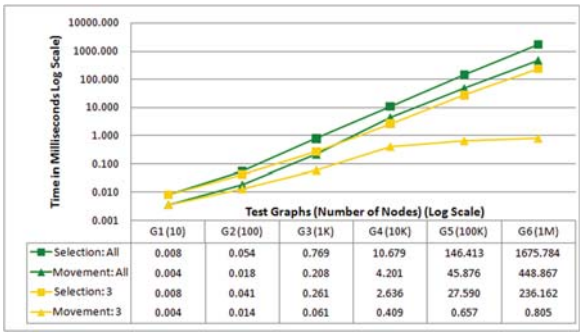
We note that for our system, this test represented a worst-case analysis since the position of every graph element is changed at each frame. In regular use many elements do not require re-rendering because they are not moved between frames. For server side method, it represents the average case since the entire graph is re-rendered regardless of the number of altered node positions.

5.4 Interaction

Up to this point in our evaluations we have defined test data based on small world networks, and tested rendering times in a variety of ways. The next step in evaluating our interactive graph visualization system focuses on the speed and scalability of our interaction algorithm, described in Section 4.5. A test suite was designed to sit on top of the interaction algorithm and simulate user interactions in a browser window. The test harness selects a node at random and "drags" it a to a randomly generated position in the viewing window. This triggers the interpolation algorithm which changes the graph layout accordingly. The time taken to generate new graph positions based on this simulated interaction was recorded. This automated test was performed on both client and server-side interaction algorithms (implemented in JavaScript and Java respectively). For the both client and server-side tests, 2000-fold cross validation was used to produce a very reliable result set.

The test harness was run using each of the graphs $G1, ..., G6$ from Table 1. Figure 6 shows the results for the server side algorithm. Axis values are logarithmic since our test graphs increase exponentially in size. Plots in Figure 6 represent tests of the two stages of the interpolation algorithm: selection and movement. These are done firstly for the full graph, and then for nodes within a graph distance of 3 from the selected node (i.e: the effect parameter is set to 3). Note that in the standard use-case (as described in Section 4.5.1), selection occurs only once per interaction while movement is performed on a continuous basis while a node is dragged with the mouse pressed down, meaning that movement results have much greater impact on system performance. As discussed in Section 4.5.1, runtime for the selection component of the interaction process increases linearly with respect to the number of nodes and edges in the test graphs. However, runtimes for the movement component scale linearly with the number of nodes only, resulting in a fast, smooth interaction with graphs up to hundreds of thousands of nodes.

The interpolation test produced excellent scalability results. In server-mode, for graphs up to 1K nodes ($G3$), selection and movement of the full graph and the 3-hop subgraph occur in less than



	G1 (10)	G2 (100)	G3 (1K)	G4 (10K)	G5 (100K)	G6 (1M)
Selection: All	0.008	0.054	0.769	10.679	146.413	1675.784
Movement: All	0.004	0.018	0.208	4.201	45.876	448.867
Selection: 3	0.008	0.041	0.261	2.636	27.590	236.162
Movement: 3	0.004	0.014	0.061	0.409	0.657	0.805

Figure 6: Results for server-side interpolation algorithm test, Selection and movement times for our test graphs are shown in milliseconds

Graph	$G1$	$G2$	$G3$	$G4$	$G5$	$G6$
Avg. Img Size	6.5	20.5	34	32	34	29

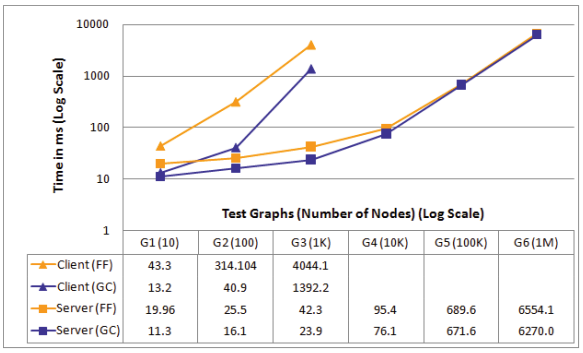
Table 2: Average image size in kB for each test graph

1ms. For the bigger graphs, selection for the subgraph is significantly faster than for the full graph. For example, $G6$ takes 236 ms for the subgraph versus 1.6 seconds for the full graph. This result is intuitive because a limited depth breadth first search is performed by the algorithm for the subgraph. For the movement process, there is a far greater timing improvement for the subgraph, because a much smaller portion of the graph is being moved. At $G3$ (1000 nodes), movement of the full graph takes 0.26 ms, compared with 0.06 ms for the subgraph. This is a relative reduction of 77%. However, at G5 (100K nodes), the full graph takes 46 ms and the subgraph takes 0.66 ms, giving a relative reduction of 98.5%. At $G6$ (1M nodes), movement takes 449 ms. The trends in Figure 6 clearly show that the selection process takes longer than the movement process for larger graphs, and that variance of the effect parameter has little or no impact on processing time for graphs of the order of $G3$ or less, but has a strong impact on larger graphs.

Similar trends were discovered for the client-side test, but on average the client method took 1.8 times longer than the server method, which is an expected result since the client-side JavaScript is an interpreted language and our server-side Java code is pre-compiled.

5.5 Network

The third and final factor in the component analysis of our technique is a look at various network delays that occur as we pass data asynchronously between the client and server. In client-mode, network delay is minimal since we are only passing initial layout data for graphs of the order of $G3$ or less. Additionally, rendering and interactions are computed locally, so network delay for interaction is non-existent. However, updates from the client model are passed across the network to maintain synchronicity between client and server graph models. This allows us to switch to server-mode at any time. In server-mode, network capacity has a severe impact on system performance since images of rendered graphs are constantly passed from server to client side. Table 2 shows the average image size in kB for all graphs in our test set. Assuming a network speed of 1000 kB/s (which is common for university campuses) the values in Table 2 are also equivalent to the transfer time in milliseconds for each image. Table 3 presents a breakdown of the total interactive visualization process with network delays included. For graphs of about 1 million nodes network delay represents less than 1% of the total processing time. For smaller graphs, e.g $G3$, the delay can account for about 59% of the entire process since the size of an image of the rendered graph is relatively stable across graphs $G3, ..., G6$. We also evaluated how much delay would be introduced by a slower connection of 1000 kb/s which is a common connection speed for residential areas in the USA. This would obviously introduce eight times more network overhead, resulting in about 330 ms per frame for $G4$ and about 890 ms per frame for $G5$. Interaction with $G6$ would still be under 6 seconds per frame.



	G1 (10)	G2 (100)	G3 (1K)	G4 (10K)	G5 (100K)	G6 (1M)
Client (FF)	43.3	314.104	4044.1			
Client (GC)	13.2	40.9	1392.2			
Server (FF)	19.96	25.5	42.3	95.4	689.6	6554.1
Server (GC)	11.3	16.1	23.9	76.1	671.6	6270.0

Figure 7: Results of interactive visualization experiment, showing average times per frame for the worst case scenario, where every node is repositioned in every frame.

5.6 Putting It All Together

Up to this point, we have focused on analysis of the various components of the *WiGi* tool at an individual level. Now we put them all together to evaluate the performance of the system as a whole. This evaluation is performed in two parts, firstly an analysis of speed and scalability is presented based on our test data. Secondly, we present a comparison of our technique against three popular graph visualization systems with respect to speed and scalability.

5.6.1 Scalability Test

Figure 7 shows the time in milliseconds for the full interactive visualization process on Graphs $G1,\ldots,G6$, which includes rendering, interaction and network delays. These results are for interaction with the entire graph, i.e: the effect parameter was set to maximum value, making interactions effect every node. This represents the worst case scenario for our system since every node is repositioned in each frame. The test was run in Firefox (FF) and Google Chrome (GC) browsers in client and server modes. There is an obvious difference in scalability between client and server modes. At $G1$ (10 nodes) there is only a few milliseconds difference between them, but at $G3$ (1000 nodes) the client process is taking 96 times longer than the server (4044 ms compared with 42.3 ms). Again in this test we can see that for the client side process, FF is far slower than GC, taking 4.6 times longer on average. The surprising result in this test is that our novel technique for computing graphs remotely (i.e: the server side method) is actually faster than JavaScript for large *and* small graphs. (in GC, 1.2 times faster for $G1$, 2.5 times faster for $G2$, and 58 times faster for $G3$). The test was also performed with single node interaction and a similar trend was revealed. For full graph interaction in GC, the million node graph ($G6$) took about 6.3 s, while the single node interaction took 5.7 s. At this scale, both browsers exhibited very similar performance since the bulk of the processing time was server-side (as shown by the rendering test in Figure 5 and the percentage breakdown in Table 3). Despite the better performance of the server-side technique in this test, the motivations for using client-side mode (discussed in Section 4.1.1) remain valid for small graphs.

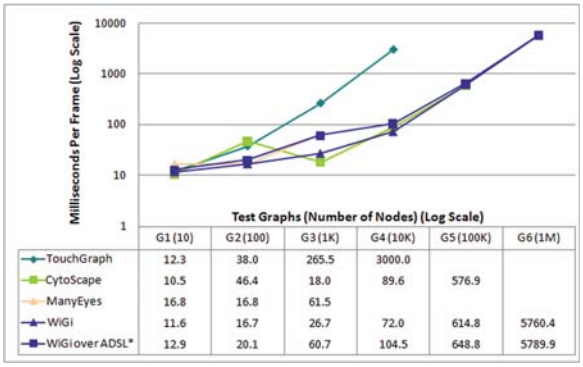
5.6.2 Delay Breakdown

To gain an understanding of the delays caused by each part of the online interactive visualization process we computed a percentage analysis for each step for all of our test graphs. Table 3 outlines the results for graphs $G1,\ldots,G6$. For $G1$ and $G2$, client-mode was used because this is the system default for small graphs and gives the best performance in most cases. The table shows the percentage time for rendering, interaction, and the expected network costs. The total column shows an empirically tested value for the entire process over each graph. The difference between the total and the sum of component pieces is shown as "Other". We suspect that this value is due to various system processes, browser overheads, other unmeasured parts of our system and other performance inhibiting overheads.

Image size is influential for the performance of our tool when operating in server-mode. For our evaluations, the graph window was maintained 600x600 pixels to fit in the browser at most screen

Graph	*G1*	*G2*	*G3*	*G4*	*G5*	*G6*
Mode	Client	Client	Server	Server	Server	Server
Rendering	89%	83%	39%	57%	71%	77%
Interaction	0.04%	0.04%	0.4%	3.9%	6.5%	7.1%
Network	0%	0%	59%	30%	5%	0.5%
Other	11%	17%	2%	9%	18%	16%
Total *ms*	13.2	40.9	57.9	108.6	705.6	6299.5

Table 3: Percentage breakdown of the online interactive visualization process in Google Chrome for our test graphs.



	G1 (10)	G2 (100)	G3 (1K)	G4 (10K)	G5 (100K)	G6 (1M)
TouchGraph	12.3	38.0	265.5	3000.0		
CytoScape	10.5	46.4	18.0	89.6	576.9	
ManyEyes	16.8	16.8	61.5			
WiGi	11.6	16.7	26.7	72.0	614.8	5760.4
WiGi over ADSL*	12.9	20.1	60.7	104.5	648.8	5789.9

Figure 8: Results of scalability and speed comparison against other systems. *Network delay over a 1000 kB/s network connection is estimated based on average image size.

resolutions, producing for example, an average image size of 34kB for $G5$. However, since we are interested in potentially huge graphs, which may require more screen estate to display adequately, we also considered the impact of bigger window sizes. When we increase the window size to 1200x1200 pixels (4 times the original area), the average image size becomes 154kB. Running the system in server-mode with 600x600 pixel screen size takes 648 ms per frame while the 1200x1200 size takes 1051 ms per frame over a 1000 kB/s network. This is due to network overhead, and increased rendering times since graphical primitives contain more pixels.

5.7 Comparison

To conclude the evaluation of our system, we now discuss a comparative test against three popular graph visualization systems: Touchgraph Navigator [22] (A Java Applet), IBM Many Eyes [4] (Java Applet), and Cytoscape [20] (a desktop application). A direct comparison with the plug-in free web-based version of Tom Sawyer Visualization was desired, but since this discussion focuses on scalable interaction, a direct comparison became infeasible because we were unable to interact with graphs in that system when more than a few hundred nodes are displayed. Our test dataset from Table 1 was parsed into appropriate formats for each system and interactions timings were recorded for each using FRAPS, while keeping all graph elements in the viewing window. We note that the primary focus for these applications is not necessarily on scalability as they have many rich data exploration features for a variety of specific tasks, but this experiment does highlight that some of these systems are quite limited in scale. Since the other systems did not support interaction with the full graph based on single node movements, we restricted our system to movement of one node only. However, we note that in the worst-case, when the entire graph is repositioned based on the interaction algorithm, the timings for *WiGi* increase only by a very small amount (See Figure 6 for examples). Since our system runs natively in the browser, FRAPS could not record timings. A JavaScript test harness was written to emulate a real user interacting with the graph. (Note: manual tests were also performed and similar results were achieved.) A click was simulated on a random node and it was moved to a random position in the view window, thus triggering selection and movement processes. The movement step was repeated 500 times and an average time was recorded. The experiment was repeated for graphs $G1,\ldots,G6$. Our system was tested with the browser running on the same machine as the server, and then network overhead with a connection

of 1000kB/s was projected based on the image sizes. The fastest mode was used, which was server-side for all except when network overhead was included on $G1$ and $G2$.

Figure 8 shows the results of the interaction experiment in Google Chrome. For graphs of size $G3$ or less, all the systems completed the test in less than 100 ms per frame on average, except Touchgraph which took 265 ms per frame. Our system showed a time increase with respect to graph size that is slightly less than linear. This occurs because overheads such as network time take up a smaller percentage of the overall process as graph size increases. The best performer from the other systems was Cytoscape, which took 570 ms for $G5$, which was 37.9 ms (6%) faster than our tool. An interesting trend in the graph occurs between $G2$ and $G3$ on the Cytoscape plot, where time per frame is reduced by 28 ms despite the increase in graph size. This occurs because Cytoscape renders nodes as squares instead of circles for graphs above a certain size. *WiGi* completed the test for $G6$ in an average of 5.7 seconds. These results show that the server side technique used in our system is more efficient than current graph visualization standards on the web.

The blank spaces in Figure 8 represent failed attempts to load data. Under the setup described in Section 5.1, the largest of our test graphs we could load in Many Eyes was $G3$. TouchGraph failed at $G5$, while Cytoscape failed at $G6$. *WiGi* was the only system to successfully load the million node graph $G6$. Furthermore, we were unable to find another web-based graph visualization tool that could display graphs of the order of $G5$ or higher.

Load times for each system were also noted, as they contribute greatly to the overall user experience. The *WiGi* outperformed all other systems for every graph. $G1$ and $G2$ were loaded by all systems in less than 1 second. *WiGi*, Touchgraph and Cytoscape loaded $G3$ in less than one second, while Many Eyes took 5 s. Only *WiGi* and Cytoscape loaded $G5$, taking 2.6 and 4 seconds respectively, making *WiGi* 1.5 times faster.

In addition to the scalability benefits outlined in this experiment, a key advantage our technique has over the others is that our interaction algorithm allows a user to mold the large graph into a form that fits with his/her pre-existing cognitive model of the data, based on its connectivity. In the other systems, interactions could only be performed on single nodes or groups of nodes selected based on physical proximity, making it difficult to manipulate very large graphs in meaningful ways.

6 Discussion

In addition to the scalability advantages of our system, the fact that it is fully web-based (i.e: native) gives it the flexibility and ease-of-use to easily be applied to solve real-world graph visualization problems. Despite the recency of the *WiGi* tool, it has already been adopted in several applications. The tool is currently deployed by the U.S government in Blackbook- a data integration and search system used for counter-terrorism [16]. In this tool, the *WiGi* visualizes interconnections between artifacts from a variety of diverse datasets, such as security reports or financial information. At the University of California, Irvine, *WiGis* have been deployed in a topic detection system [3] for newspaper articles. At the University of California, Santa Barbara, the tool is deployed to visualize results from a citation network analysis program for discovering connections among research papers and authors, and generating reference files. Also at UCSB, *WiGis* have been used by the Earth System Science Server (ES3) project [9], a (data provenance system) to visualize modifications made to files by various processes in a filesystem. The diversity and scope of these applications indicate that our system is a useful, adaptable interactive visualization service with broad reaching potential.

7 Conclusion

Web-based data visualization is rapidly gaining popularity in in many communities. The unprecedented growth in data available from the social and semantic web has brought with it the need for faster and more intuitive data exploration mechanisms. In this paper we have presented a new technique for web-based interactive graph visualization which harnesses the power of asynchronous client-server data transfer to achieve smooth, scalable interactions with large graphs ($>$ 100K nodes) natively in a users browser. To our knowledge,*WiGi* is the first system of its kind for large scale node-link graphs, and we believe that others will adopt our approach to large-scale visualization. We have presented empirical evaluations of our technique, comparing layout and interaction speed and quality against three popular existing graph layout tools. In all evaluations, our technique outperforms the existing systems in terms of speed and scalability.

References

[1] Albert-Laszlo Barabasi and Reka Albert. Emergence of scaling in random networks. *Science*, 286:509, 1999.

[2] Vladimir Batagelj and Andrej Mrvar. Pajek - program for large network analysis. *Connections*, 21:47–57, 1998.

[3] Chaitanya Chemudugunta, Padhraic Smyth, and Mark Steyvers. Text modeling using unsupervised topic models and concept hierarchies. *CoRR*, abs/0808.0973, 2008.

[4] Catalina M. Danis, Fernanda B. Viegas, Martin Wattenberg, and Jesse Kriss. Your place or mine?: visualization as a community component. In *CHI '08: Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, pages 275–284, New York, NY, USA, 2008. ACM.

[5] Tim Dwyer and Kim Marriott. Drawing directed graphs using quadratic programming. *IEEE Transactions on Visualization and Computer Graphics*, 12(4):536–548, 2006. Member-Koren,, Yehuda.

[6] Tim Dwyer, Kim Marriott, and Michael Wybrow. Topology preserving constrained graph layout. In *Graph Drawing*, pages 230–241, 2008.

[7] P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[8] P. Eades and M. Huang. Navigating clustered graphs using force-directed methods, 2000.

[9] James Frew and Peter Slaughter. Es3: A demonstration of transparent provenance for scientific computation. pages 200–207, 2008.

[10] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Softw. Pract. Exper.*, 21(11):1129–1164, 1991.

[11] Pawel Gajer, Michael T. Goodrich, and Stephen G. Kobourov. A fast multi-dimensional algorithm for drawing large graphs. In *In Graph Drawing00 Conference Proceedings*, pages 211–221, 2000.

[12] Stefan Hachul and Michael Jünger. An experimental comparison of fast algorithms for drawing general large graphs. In Patrick Healy and Nikola S. Nikolov, editors, *Graph Drawing, Limerick, Ireland, September 12-14, 2005*. Springer, 2006.

[13] David Harel and Yehuda Koren. A fast multi-scale method for drawing large graphs. In *Graph Drawing: 8th International Symposium (GD'00)*, pages 183–196, 2000.

[14] Herman, G. Melan, and M. S. Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43, /2000.

[15] Don House, Victoria Interrante, David H. Laidlaw, Russell Taylor, and Colin Ware. Panel: Design and evaluation in visualization research. In *Proceedings of IEEE Visualization Conference*, Minneapolis, MN, October 2005.

[16] Intelligence Technology Innovation Center (ITIC). Blackbook prototype framework for the knowledge discovery and dissemination (kdd) program. McLean, VA, USA, October 3–4 2006.

[17] Kyung-Sun Kim. Effects of emotion control and task on web searching behavior. *Information Processing & Management*, 44(1):373–385, January 2008.

[18] S. Milgram. The small world problem. *Psychology Today*, (2):60–67, 1967.

[19] Kazuo Misue, Peter Eades, Wei Lai, and Kozo Sugiyama. Layout adjustment and the mental map. *Journal of Visual Languages & Computing*, 6(2):183–210, June 1995.

[20] P. Shannon, A. Markiel, O. Ozier, N. S. Baliga, J. T. Wang, D. Ramage, N. Amin, B. Schwikowski, and T. Ideker. Cytoscape: a software environment for integrated models of biomolecular interaction networks. *Genome Res*, 13(11):2498–2504, November 2003.

[21] Tom Sawyer Software. Tom sawyer visualization, 2009.

[22] Touchgraph. Touchgraph navigator. Proprietary online application, Touchgraph inc. available at http://www.touchgraph.com.

[23] Peterson Trethewey and Tobias Höllerer. Interactive manipulation of large graph layouts. Technical report, Department of Computer Science, University of California, Santa Barbara., 2009.

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

[25] Jason Wood, Ken Brodlie, and Helen Wright. Visualization over the world wide web and its application to environmental data. In *VIS '96: Proceedings of the 7th conference on Visualization '96*, pages 81–ff., Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.