GUESS: A Language and Interface for Graph Exploration

Eytan Adar*

University of Washington, Computer Science and Engineering
101 Paul G. Allen Center, Box 352350, Seattle, WA 98195
eadar@u.washington.edu

ABSTRACT

As graph models are applied to more widely varying fields, researchers struggle with tools for exploring and analyzing these structures. We describe GUESS, a novel system for graph exploration that combines an interpreted language with a graphical front end that allows researchers to rapidly prototype and deploy new visualizations. GUESS also contains a novel, interactive interpreter that connects the language and interface in a way that facilities exploratory visualization tasks. Our language, Gython, is a domain-specific embedded language which provides all the advantages of Python with new, graph specific operators, primitives, and shortcuts. We highlight key aspects of the system in the context of a large user survey and specific, real-world, case studies ranging from social and knowledge networks to distributed computer network analysis.

Author Keywords

Graph visualization, domain-specific embedded language

ACM Classification Keywords

D.2.6 Programming Environments, H.5.2 User Interfaces, D.2.11 Domain-specific architectures

INTRODUCTION

Graphs models are in use today in domains as varied as social sciences, organizational behavior, physics, and biological sciences. With such wide ranging use it is not surprising that the number of visualization options available to researchers has become almost overwhelming. Researchers now must struggle to decide which tool is best suited for his or her needs. These tools are at times too general to handle the modeling of specific graph models or at times limited to one domain. The GUESS system was inspired by this need. Its design is in part the result of watching users of our previous system, Zoomgraph [2]. We have found many areas of common need in graph exploration and visualization systems, and in particular two main tasks: a) the creation of static images through exploratory data analysis, and b) the creation of dynamic visualizations that they would like to distribute to others. The common feature to both use cases is a need for a flexible way of dealing with graph data.

The GUESS language, Gython, extends the Python interpreter, or more accurately the Jython system [18], by adding new operators, so-called syntactic sugar, and data structures which are tightly bound to the visualization and database backend. Additionally, our customized, interactive interpreter lets users quickly access the output of their work. For example, passing the mouse over the textual output in the interpreter window will cause the corresponding graph objects to be highlighted. A contextual menu system in the interpreter window allows further control of graph properties (e.g. changing colors). GUESS is distributed with many commands and network algorithms ranging from layout algorithms to shortest path to clustering algorithms. A number of these are contributed by the JUNG library [17] which we use for the underlying data structures.

Since its release, the GUESS system has been downloaded over a thousand times and has an active mailing list of users. It has been used in applications ranging from computer network analysis to biological networks to social networks to water-line networks. These uses have frequently been undertaken by users with very little programming experience and demonstrate the viability of our approach. In addition to our discussion of a number of case studies, we have also collected a survey from 76 individuals (primarily in the social network community) that describes their use of alternative systems and GUESS.

Below we briefly cover related work and then we delve into the Gython language with particular attention to exploratory analysis tasks. We describe the GUESS GUI (Figure 1) and interactive interpreter and conclude with further discussions of our user survey and two case studies in which users have developed visualizations using GUESS.

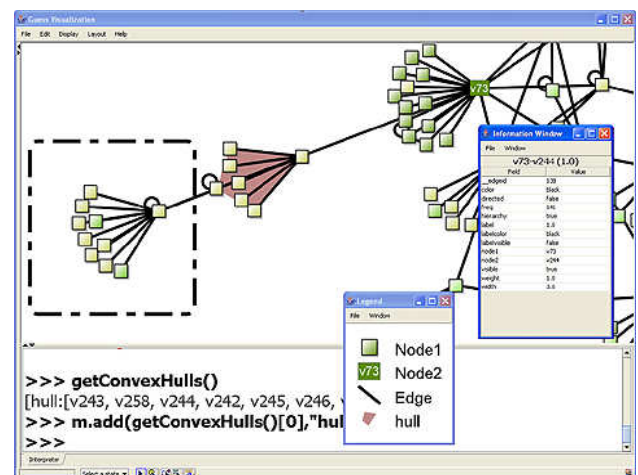

Figure 1: A screenshot of the GUESS system.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CHI 2006, April 22–27, 2006, Montréal, Québec, Canada.
Copyright 2006 ACM 1-59593-178-3/06/0004...$5.00.

* Work done while author was at HP Labs

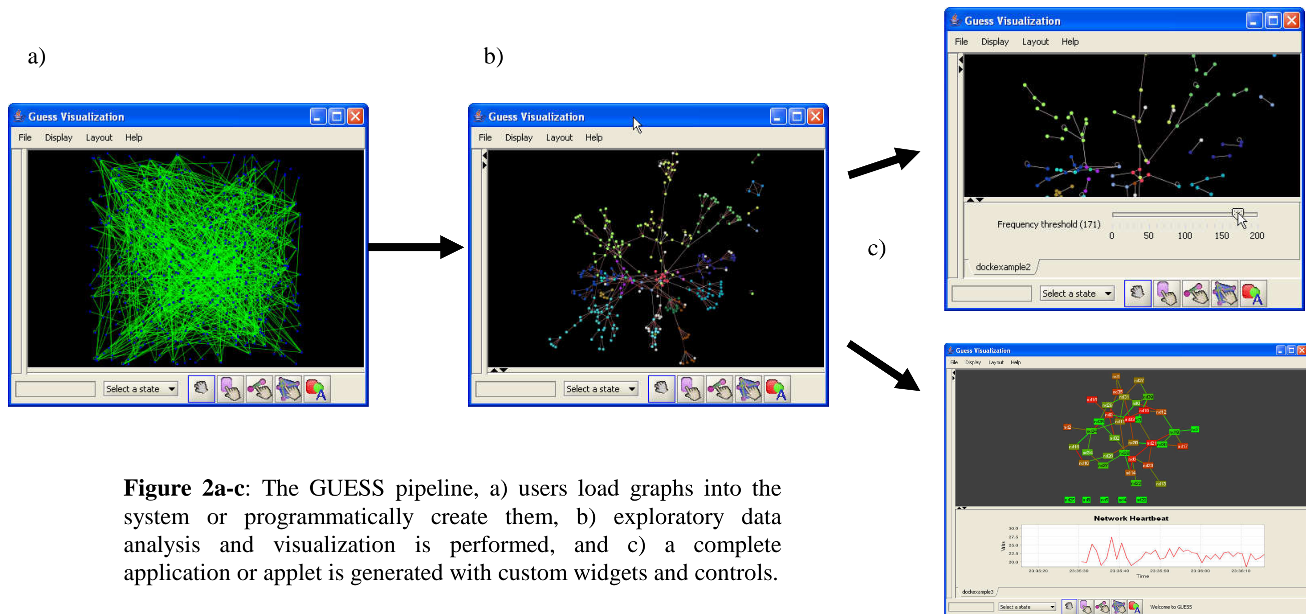

Figure 2a-c: The GUESS pipeline, a) users load graphs into the system or programmatically create them, b) exploratory data analysis and visualization is performed, and c) a complete application or applet is generated with custom widgets and controls.

RELATED WORK

A full discussion of the many graph drawing packages is beyond the scope of this paper but are extensively described in [23]. Because our survey asked users familiar with some of these systems to describe their impressions we will address a few specific systems in the discussion of the language and interface. Briefly, however, we believe that GUESS is related to three specific system categories: graph analysis/visualization systems, graph libraries, and database visualization.

The GUESS system is an attempt to combine analysis and visualization into one package that supports Exploratory Data Analysis (EDA) [29] for graphs. It thus distinguishes itself from solutions that require one system to perform analysis, such as partitioning (e.g. using Analytic Technologies' UCINET), followed by a different program for rendering (e.g. Pajek [4] or GraphViz [9]). Interactive exploration is difficult when a user is forced to go back and forth between the analysis and the visualization packages.

Similarly, systems using sophisticated APIs that require a program → compile → run (PCR) cycle are also unattractive. Frameworks in this domain include excellent graph analysis and visualization libraries that are ideal for low-level programmers. For example, in C/C++ we have LEDA [21] and in Java the GET/GLT [27], Prefuse [13], yEd [32], and JGraph [16] systems. GUESS does not seek to displace these toolkits but rather provides a new way to interface with them. For example, we have demonstrated that our visualization layer can be replaced with Prefuse and TouchGraph [28] (while still controllable through Gython). Our main use case is depicted in Figure 2. We believe that GUESS should allow users to simply load a graph and immediately begin controlling the visualization. Once the analysis is complete, more experienced users can go further and easily prototype more complex visualization applications.

The GGobi package contains a graph visualization plug-in that is embedded in the R system [26]. Other approaches, such as LINK [6] or Graphlet [14], use interpreted languages (Scheme and TCL, respectively) and are more related to our goals. Users of Graphlet are given access to various graph structures and algorithms and can manipulate the Tk GUI. Graphlet is in fact an evolution from the GraphEd [15] system, an older, C based system and illustrates the evolution to interpreted languages. Though Graphlet is similar to GUESS in a number of ways, we believe that by adhering to the unmodified TCL language, the system forces users to make use of complex, multi-line programs to achieve basic goals that could be satisfied with concise, domain-specific operators and primitives. Unlike other systems which simply implement graph data structures and algorithms in a specific language, GUESS augments the language to ease common user tasks. Additionally, by not providing a stronger connection between the interactive interpreter and the visualization, systems such as Graphlet add work for the user who must manually match the output of exploratory commands to the visualization.

Other solutions that make data exploration possible are not necessarily streamlined for data structures common in graph analysis. The GGobi/R solution, for example, requires users to manipulate matrices and vectors and frequently requires additional steps for very simple tasks. A different approach is the use of the general LEFTY graphics language [19] in GraphViz. We believe that a popular language such as Python, augmented with operators and syntactic sugar, reduces the learning curve while providing the right level of abstraction. Additionally, GUESS is based on Python because the language is familiar to many users and is similar to a number of statistical systems (e.g. S-plus and R). There are also a wide array of libraries implemented in Python that can be easily integrated into GUESS.

We also believe that our use of a modified Python core distinguishes GUESS from database visualization systems such as Tioga [3], Polaris [25], DEVise [20], and Rivet[7]. While these approaches are potentially able to handle graph structures, the generality of their approach limits their abstractions and forces more work for the user. By focusing on language structures and operations that are unique to graphs, we believe GUESS users can more rapidly develop visualizations on these types of structures.

Another interesting area of related research has been the work in graph database and schema designs. GML and XML-based approaches such as GraphML, and GXL, provide a mechanism for describing and passing graphs between applications, as well as flexibility for additional attributes [10]. However, while graphs in these formats can express many user-defined properties, storage in an XML database and querying (e.g. in XQL, XQuery, XPath, XUpdate) is difficult for novice users. Other solutions, such as GraphDB [11] and GOOD [12], were designed from the start to hold and query graph structures but lack visualization features.

THE GYTHON LANGUAGE

When we originally built Zoomgraph we drew inspiration from domain-specific languages such as GnuPlot and the statistical analysis system R. However, we quickly learned that this require a steeper learning curve for users. In designing Gython we have opted to use a domain-specific-embedded language (DSEL) instead. Specifically, we extend Python with features necessary for dealing with graph structures. Because Gython is an extension of Jython (an implementation of Python in Java), our users have the ability to rapidly add to the GUESS GUI. In fact, we have created a great many classes and examples to bootstrap visualization tasks.

GUESS Objects – Nodes and Edges

Because nodes and edges are the primary currency of any graph we chose those to be the primary (first-class) objects in GUESS. Each node has a *name* that is accessible from the global namespace. For example, in a social network we may have a node named bob_jones, or in a protein-protein interaction network we could have Hsp70. The user will not necessarily be accessing individual nodes, but the option is available to them. Edges are also uniquely identified, but these identifiers are not directly accessible to the user. Instead, edges are referenced by their two endpoints. For example, if we assume two nodes, *bob* and *alice*, the edge between them is *bob<->alice*. From the implementation perspective, the bob node will look up any undirected edges connecting it to alice.

In order to select edges that that represent different relationships, we have added additional operators into the language, ->, <->, <-, and ?. These are defined as:

- `alice<->bob` selects all undirected or bidirected edges between alice and bob (undirected and bidirected are equivalent in GUESS)
- `alice->bob` and `alice<-bob` selects all directed edges.
- `alice?bob` selects all edges between alice and bob.

Though we have added shortcuts operators, we have avoided overloading (i.e. redefining) existing operators to prevent user confusion. By making these operators a part of Gython, users have a concise way of selecting groups of edges. In other implementations, such as Graphlet, or many of the other toolkits, the user must iterate over edges to find specific matches (e.g. a multi-line "for each" program). This is a crucial design point of GUESS. Because a great deal of interaction with GUESS happens through an interactive interpreter, where users enter commands which are immediately executed, it is undesirable to have multi-line commands as these become difficult to enter and correct. Another solution would have been to create functions such as: *alice.edgesTo(bob)* or *getDirectedEdges(alice,bob)* to mask the iterations. However, we feel that these lack the conciseness of the more direct statement: *alice->bob* and force the user to remember longer commands (e.g. was it *getDirectedEdges* or *findDirectedEdges*?).

In addition to nodes and edges, the other main structure in GUESS is the set. This is based on our observation that users most often deal with groupings of nodes or edges rather than those edges themselves (e.g. all edges representing relations of a certain type, all nodes representing certain employees, etc.) Although Python has the notion of sets at the most basic levels we extend this in Gython to speed up certain tasks. For example, if we have five nodes, N1 through N5, we can create the following two groups of nodes:

```
group1 = (N1,N2,N3)
group2 = (N3,N4,N5)
```

We can find the intersection by using the "&" operator or the union with "|" which are both novel to Gython (e.g. *group1 & group2*). This is useful because users are frequently interested in groups that fall in multiple groups (e.g. department 1 employees that are managers or vice presidents). Another common task is finding the edges between groups of nodes. For example, we would like to find all routes of communication between computers in China and computers in the US. To facilitate this, the same edge selection operators that worked for individual nodes will also work on sets. So given our two groups, a user may use the command *group1->group2* to find all directed edges between the two (note that *(N1,N2,N3)->(N3,N4,N5)* will generate the same results.

The working graph object exists in the global namespace as the variable "*g*". The set of all nodes and edges in the graph are accessed as *g.nodes* and *g.edges* respectively. Using these sets a user could find all outgoing edges from N1 by using the command: *N1->g.nodes* (though there are alternative methods).

Node and Edge Fields

Based on our experiments with Zoomgraph and our survey we have found that users require graphs structures to support properties. While graphs can be described as a simple matrix, most users require graph structures where nodes and edges have more complex properties. Of the 55 survey participants that answered this question, 35 (or 64%) indicated that they worked with graphs with complex properties (e.g. employee identifier on nodes, relationship type on edges, etc.) and 14 (or 25%) indicated needing at least simple properties (e.g. weights on edges, labels for nodes, etc.), with only the remaining 6 (or 11%) satisfied with simple matrices.

In GUESS, a user has the option of associating *fields* (i.e. properties or attributes) with nodes and edges. Currently, fields can be textual, numerical, or Boolean (with some additional exceptions for shapes and images). For example, one of our sample applications involves a social network within a company. Nodes, representing employees, have a textual department field associated with them called *dept* and job function field called *jobfunc*. For example, Bob, Alice and Jane are Manager, Designer, and Intern respectively. Similarly, edges have a *freq* field representing the frequency of communication between two employees (e.g. Bob and Alice communicated 10 times).

To be consistent with Python we decided that attributes could be accessed by appending the field name to the variable name (e.g.

variable.field). This is standard in Python, but required modifications to the Jython implementation. For example:

- `bob.jobfun` returns "Manager"
- `(bob<->alice).freq` returns 10
- `(alice<->jane).freq = 21` updates the amount of communication between alice and jane.

Fields come in two flavors, data fields, and visual fields. The 3 examples above are simple examples of data fields. Visual fields correspond to properties that are used by GUESS to visualize the graph. For nodes these include, *style, height width, fixed, visible, color, label, labelvisible, indegree, outdegree, totaldegree, x,* and *y*. For edges, GUESS currently utilizes *color, width, label, labelvisible, directed*, and *weight*. When loading a new graph into the system, a user may define the initial values for any subset of these attributes. Certain properties (e.g. *indegree, outdegree, totaldegree*) are calculated dynamically with structure changes. These are calculated as needed in order to prevent overhead to graph operations. Other measures, for example graph centrality measures [30] or PageRank [24], are also generated the first time those fields are accessed on a node (e.g. *alice.betweenness* or *bob.pagerank*).

Changes to visual properties cause an immediate change to the display. For example, *bob.color = red* will set the bob node to red (GUESS has namespace definitions for nearly 80 colors but will also accept an RGB triplet). Users can make use of Python's iterators to modify the fields of groups. The for-loop, *for temp in g.nodes: temp.color = red* will set all graph nodes to red. However, because such functions are used so frequently and the syntax is cumbersome, an additional feature of Gython is the application of setter operations to groups. For example:

- `g.nodes.color = pink` will set all nodes in the graph to pink
- `(alice,bob).style = 2` will make the alice and bob nodes circular (default is square)
- `(bob,jane).size = 20` will change the size of bob and jane to 20 pixels (this is syntactic sugar that sets height and width at the same time)
- `(alice-bob,bob-alice).width = 3` will change the line width on the two edges.

In addition to fields, nodes and edges have a number of methods that can be used to find various graph features. For example, *alice.unweightedShortestPath(bob)* will calculate the length of the shortest path from alice to bob. As we begin to stabilize these methods and find which are frequently used we may replace them with operators (i.e. *alice<*>bob* may come to mean shortest path). Many other functions exist for selecting neighboring edges, nodes, and other graph properties but are beyond the scope of this paper.

Though both types of fields, visual and data, exist in GUESS, users access both in the same manner. This is very different than most toolkit implementations such as JUNG or Prefuse in which a Model-View-Controller model disentangles data from visualization. While useful from a programming perspective, such models frequently require additional code. For example, we could define an *EdgeWidthRenderer* object which tells the visualization system what width to set each edge to (perhaps based on some edge property). We believe that in exploratory situations users simply wish to modify visual properties directly and not implement rendering objects.

While GUESS is not intended to scale to Internet sized graphs we have successfully loaded and manipulated networks of tens of thousands of nodes and edges on a standard PC. Layout algorithms are by far the most expensive operations though the usual graph sizes (< 5000 nodes) are laid out nearly instantaneously and standard operations such as coloring and grouping are nearly real time for much larger graphs.

Filtering Fields

In addition to controlling node and edge properties, fields are also used in GUESS to filter graph elements. In order to make filtering commands directly accessible to users a unique object is automatically created for every field that is either defined by the system or user. That is, for every field in the system a field object is created and made accessible in the global namespace.

These objects have overloaded ("==", ">", "<", ">=", "<=", and "!=") and added operators ("like", "roverlaps", "rcontains", "rexact", and "rcontained") When the operators are applied to a field object, GUESS will find all nodes or edges matching the filter and return a set of matching objects. For example:

- `freq > 10` will select all edges with communication frequency greater than 10, and `(freq > 10).color = blue` will make those edges blue
- `name like 'al%'` will find all nodes whose name starts with "al".
- Users can also make queries between fields. For example `x > y` finds all nodes whose x location is larger than their y location.

We believe that while including the query syntax in the language is common in database visualization systems, it is novel in this context. Because the bulk of graph systems make use of languages that are not targeted at querying datasets, users are forced to make database queries, and remap the results back to the graph data. Presuming a connection to a database, a user may have to do something of the form:

```
matchingRows =
 db.query("SELECT * from edges where freq>10")
for each row in matchingRows:
 matchingEdge = mapRowToEdge(row)
```

While we may be able to hide this in a function (e.g. *findMatching("freq > 10")*) this becomes more cumbersome when users begin to require unions, intersections, or connections between sets. If we relied on only existing language constructs such as those provided in Java or TCL we may have to do:

```
intersection((alice,bob),(findMatchingNodes("job
func == 'manager'"))
```

to find who between alice and bob is a manager. In Gython the equivalent query would be:

```
(alice,bob) & (jobfunc = 'manager')
```

A user could also find all edges connecting alice with a frequency of communication greater than 15 by doing:

```
(alice<->g.nodes) & (freq > 15)
```

If a naming conflict exists between node and edge fields, the node field is bound in the namespace (e.g. *width*). To specify

which width field one is interested in, we prepend Node. or Edge. to the field name (e.g. *Node.width* vs *Edge.width*).

Fields are objects themselves and have properties including simple measures like "max" and "avg" (e.g. freq.max). Fields are also passed as arguments to various functions including *sortBy(fieldname)* and *groupAndSortBy(fieldname).* Respectively, these functions generate a set of nodes or edges sorted by a particular field or a set of sets in which objects with equivalent field values are grouped together. Notably we can also use the syntax *group1.sortBy(fieldname)* to sort all items in group1 by some field.

Loading Graphs

Unlike Zoomgraph, users of GUESS have more options for loading graphs into the system. A simple comma separated format called GDF is the easiest for new users. However, GUESS now supports GraphML and the Pajek file formats for importing from other applications.

In addition to loading in various graph description files, users of GUESS can also create and remove nodes, edges, and fields on-the-fly. Various primitives support these functions and the data is appropriately saved into the backend database.

Because GUESS uses a database as a backend, advanced users can switch in their own databases by implementing a simple API. One user of GUESS, who had previously created a large system for analysis, elected to use a network module (built in GUESS) which allowed him to connect remotely and simply execute scripted commands to construct and manipulate a graph.

Graph States

There are two main reasons we may be interested in preserving graph states. The first is the necessity of undoing events in exploratory tasks, and the second is in the analysis of time-sensitive data (e.g. dynamic graphs).

Although we noticed this in Zoomgraph users, participants in our survey confirmed for us the importance of reverting to previous versions of the visualization. When asked to name their main issues with systems such as Pajek and UCINET, users responded with statements such as: "[In UCINET] you have to reload a network for every operation. Horrible." and for Pajek "once you have applied a spring embedding algorithm you can never return to the previous sociogram display." Clearly, when performing exploratory visualizations it is crucial to revert to previous layouts if the current one is unsatisfactory (this is especially important since many layout algorithms require a long time to compute). In GUESS we allow users to issue a "save state" command, *ss(state name),* or to easily retrieve a saved state through the load state command, *ls(state name).* The argument to both methods is a either a string or integer which is becomes the name of the state. While we could do this automatically for the user after every command to support full undo, we disable this option in the distributed version as it has a computation cost and may disrupt the user's flow[1].

While there are many instances of graphs that are static, frequently users are interested in graphs that are changing over time. In our own work we have had to visualize various dynamic social networks and have generated movies to depict various phenomena. In our survey population we asked the participants if they required visualizations of dynamic graphs (yes or no). Of the 59 that answered, 47 (or 80%) indicated a need to visualize such graphs. On the other hand several noted in their criticism of systems such as Pajek and UCINET as well as toolkits such as JUNG that very little support is available for this task.

The Gython language supports querying and access of fields at different states. For example, if we had defined a state for every year we would type *bob[1999].jobfunc* to find Bob's job function in 1999. We could also find all communication edges where the frequency of communication increased from 1999 to 2000 by doing: *freq[2000] > freq[1999].*

In order to preserve a mental map model [22], GUESS, like Zoomgraph, implements a *tweening* algorithm that smoothly transforms a graph from state to state. A user may specify the amount of time to spend on this transformation. Because nodes and edges can appear and disappear between states and are distracting in their transition we have added additional controls that define how quickly in the cycle nodes and edges should disappear and how late they should appear. When a user is satisfied with their animation, GUESS can export the visualization in QuickTime format. In fact all user interactions with the visualization system can be saved. If a user wants to manually move nodes and edges or apply transformations these can be saved as a movie as well.

Sometimes dynamic graphs have a complex notion of state. For example, a social network graph may have different communication frequencies for each time period. For this type of dynamic graph, the state mechanism described above is appropriate. A far simpler type of state is a network in which nodes and edges exist or vanish depending on the time (e.g. the network link is up or is down). To represent this we allow users to define a range field which is a comma delimited list indicating when a node or edge exists. A range field such as "1,5-10,20," for example, indicates that a graph element existed during time 1 and 20 and during the period between 5 and 10 (inclusive). Users can query on this using the familiar query syntax with the operators roverlap, rcontains, rcontained, rexact. For example:

- `Node rcontains 5` returns all nodes that exist at time 5
- `Edge roverlap (4,6)` returns all edges that overlap the time period 4-6.

Now that we have defined the fundamentals of the language, we can start to see how they can be used in visual exploration.

Functions and Programming in Gython

In the original Zoomgraph system we had implemented all graph functions as reserved keywords. One of the criticisms of this was that it was difficult for users to add their own functions or modify ours. Users had to program the primitive in Java so it would be accessible to them in the Zoomgraph language. Instead, in GUESS we have bundled some functions into the appropriate objects (e.g. graph functions are part of graph objects, display window functions in the main window object, etc.). A user wanting to generate a random layout would use the command *g.randomLayout()*.

While packaging of this type is understandable to programmers, in observing our users it emerged that they were frequently confused about which functions belonged to which object.

---

[1] Independently of this feature, GUESS allows users to log their interaction sessions to file (these can later be "replayed").

Rather than having them look up these functions every time we implemented an automated system that generates a number of wrapper functions at compile time that are then defined globally. Users can now type *randomLayout()* (in fact they can do *randomLayout*, omitting the parenthesis–another Gython feature) and the system automatically knows which object to invoke the function on. In this way, we have provided both primitives and functions which the user may select from depending on their comfort level with the language.

Simple Visualizations and Exploration

While the GUESS interface provides a number of features for the exploration and manipulation of the graphs (panning, zooming, etc.), it is frequently desirable and potentially easier, to achieve this programmatically. For example, GUESS provides a *center(…)* function which will take any set of nodes and edges and center the camera around those objects, panning and zooming as needed. A user can, for example, zoom in on a department by typing *center(dept == 'IT').*

One of the most used features of GUESS are functions that map data properties to visual properties. For instance, we may want to map the communication frequency field to the width of an edge (ranging from 1 to 5 pixels). Programmatically, this would correspond to something the following:

```
for e in sortBy(freq):
     prop = (e.freq – freq.min)/
            (freq.max-freq.min)
             e.width = 1 + 4 * prop
```

Similarly, to color each department differently we could use the random color generator (GUESS also has a function to produce a sequence of colors in a range) and do:

```
for group in groupBy(dept):
    group.color = randomColor()
```

Since commands such as these are used so frequently we have created shortcut functions called colorize and resize which take a field as an argument and optional arguments such as starting and ending colors and sizes and visually transform the nodes or edges. The commands for the previous two examples would be *resizeLinear(freq,1,5)* and *colorize(dept).* Figure 3, for example, was generated using the command *colorize(totaldegree,yellow,red).*

Since GUESS is built on top of the JUNG library we can also make use of various pieces of code available that perform graph based clustering. One application where GUESS has been used is in finding communities of nodes and visually highlighting those communities. Using the betweenness clustering method, a popular community finding algorithm, and GUESS' ability to create convex hulls around sets we apply the following function:

```
for clust in edgeBetweennessClusters(7):
 createConvexHull(clust,randomColor(120))
```

In this example we are asking for 7 clusters and the *randomColor* method takes an optional alpha variable to indicate transparency. Various other clustering algorithms are implemented and available to the user. For Figure 3 we applied *groupBy(dept)* instead of the betweenness clustering method to generate the hulls.

Layout Algorithms and New Visualizations

It is rare for a user visualizing a graph to have already specified the coordinates for all the nodes. More frequently, the user will depend on the visualization system to layout the graph in a way that conveys some interesting aspect of the graph. GUESS provides a number of standard layout algorithms including Fruchterman-Reingold, Kamada-Kawai, Sugiyama, GEM, ISOM, radial layouts, and various spring based implementations (these are fairly standard techniques and are surveyed in [8]) as well as a few of our own creation. For iterative layouts, such as the spring-based techniques, users may specify the number of iterations to run. Supplying no argument will result in a query to the user every 30 seconds asking if they would like to continue.

While our algorithms are frequently sufficient for users, they may also create their own using quite easily. A custom visualization such as Figure 3 takes under 10 lines. The equivalent algorithm would have taken far more work directly in systems such as JUNG or Prefuse as we would have had to implement and compile new layout algorithms, and potentially new renderers, in order to control placement and display.

While we have not attempted extensive scaling experiments we have successfully loaded and applied the GEM algorithm to graphs with 10000 nodes and 12000 edges. Larger graphs seem very plausible and seem only limited by memory.

THE GUI

At the visual layer, the GUESS system supports the display of very large graphs through the use of the Piccolo framework [5]. Piccolo provides an infinite visualization plane with infinite zoom for 2D objects. However, in building GUESS we have opted to disentangle the data from the visual representation, thus allowing integration with Prefuse and TouchGraph.

Beyond simple pan and zoom features, users have access to a number of commands that change the display of the graph, export images, and perform basic layout operations. Users can also annotate the graph with basic 2D objects. Additional features include a property window and basic charting. Automatically generated legends are a crucial feature in exploratory environments as they help annotate visual data in a useful way. This was one of the most requested features in Zoomgraph and is now available in GUESS.

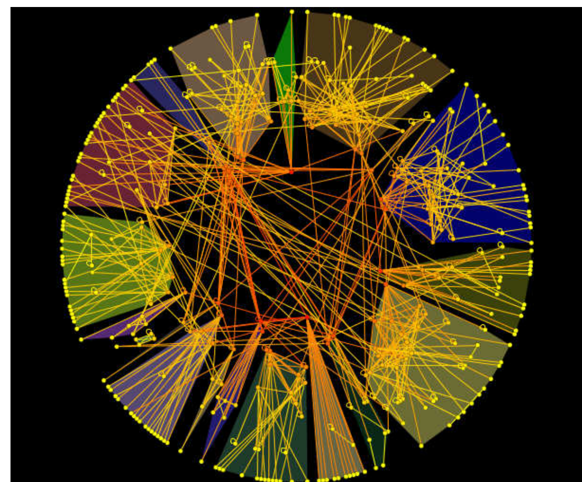

Figure 3: A sample visualization where nodes (individuals) are placed around the circle by department. The more connected nodes are pulled into the circle and colored a deeper red (edge colors are the average of the node color). A convex hull groups departments together.

Our belief is that the default interface to GUESS should be as plain as possible. Network and graph analysis in various fields has resulted in an incredibly large collection of algorithms and methods. Frequently, the methods are similar but the language and names to describe them are completely different. Even within fields such as social network analysis the number of tools is overwhelming. Pajek, for example, has 17 menus in the menu bar with an average of 8 items in each. These are themselves submenus which may go down 4 levels. Of the 47 survey participants who described their issues with Pajek and UCINET, 17 (or 36%) made specific note of the overwhelming UIs and lack of guidance as their main issue with these systems.

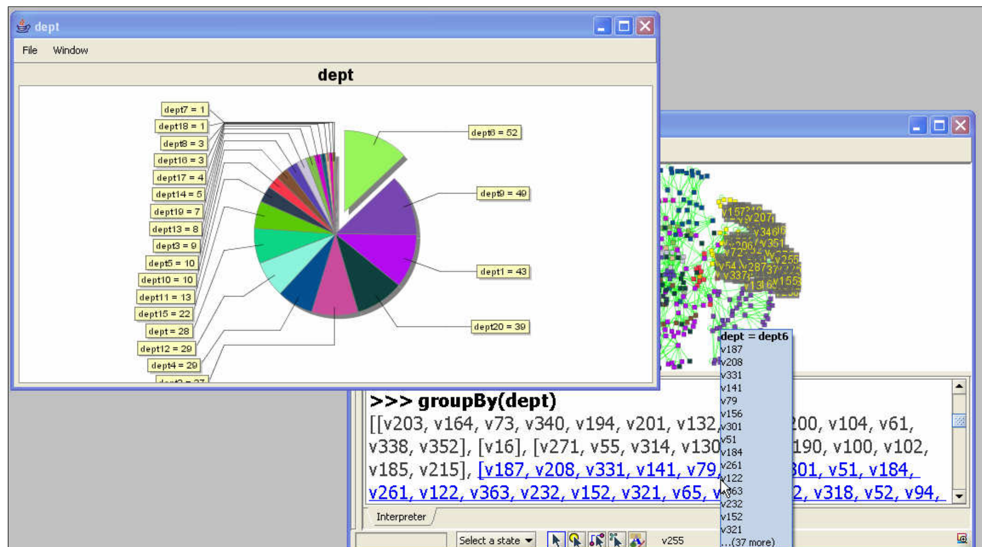

Figure 4: A demonstration of the connection between the interpreter and various visualizations. A tooltip window displays additional information. In addition to the highlighting in the graph display, note that the pie segment containing moused-over nodes is slightly pulled out.

We believe the interface that an end user is exposed to should contain the functions needed for their task. As such, we have allowed programmatic control of menu options, toolbars, and other areas of the GUI. Our hope is that users will load interface views that address their specific needs. A biologist, for example, may not see the social network algorithms that are available to the social analyst or the network flow algorithms for the computer scientist.

The Interactive Interpreter

One of the most novel features of GUESS has been the interactive interpreter. We started with the basic Jython interpreter[2] console and extended from there. The console provides a simple way to enter commands. As commands are entered they are evaluated immediately and the output is displayed in the console. Figure 4 shows a slightly enlarged view of this console in the bottom left of the visualization window. If a user ends their line with a colon, the Python symbol indicating the start of a code block, the interpreter allows the user to enter additional lines. Hitting enter on a blank link leaves the code block mode. This is particularly useful for defining loops. The interpreter also provides the usual cut and paste operations as well as a history function to cycle through previous commands.

If we look at the output of the *groupBy* command in Figure 4 we notice that the result is fairly overwhelming. The response is a large grouping of node groups (one for each department). This is useful for a programmer, but simply looking at the results may not tell us much. A user faced with this list may want to match what is in the list to the visualization. To help, we drew inspiration from trends in development environments and systems such as Matlab. When a user moves the mouse cursor over text in the interactive window and the text under the mouse corresponds to a node or edge, that edge is highlighted in the visual display. If the mouse moves over a variable representing a group of nodes or edges the complete group is highlighted. More interestingly, GUESS makes a distinction between items in sets and sets. For example, if the system returns a group of two groups such as [[V1],[V3,V4], GUESS forms the following table in memory:

1	2	3	4	8	9	10	11	12	13	14	15	16	17
[	**[**	**V**	**1**	**]**	**,**	**[**	**V**	**3**	**,**	**V**	**4**	**]**	**]**

When the user mouses over positions 3 or 4, node V1 is highlighted. Similarly, when the user moves over positions 11 or 12, V3 is highlighted. However, if the user moves over position 10, 13, or 16, both nodes V3 and V4 are highlighted in the display. Moving over positions 1, 9, or 17 causes highlighting of all three nodes. The "matched" text is always underlined to provide the user with a visual indication of the area they are moving over. From an implementation perspective we make use of an interval-tree data structure which is highly optimized for these tasks.

Due to our event management infrastructure, highlighting a node does not necessarily mean simply highlighting it in the graph window. If the node, edge, or group is part of any other visualization (e.g. a chart) that area will be highlighted as well. Figure 4 is an example of all this in action.

Tooltips that are displayed are contextualized to the type of the object being moused-over. The top entries 15 entries are shown for lists, exception logs are shown for errors, and documentation is displayed when mousing over functions. GUESS also allows functions that create lists to annotate them. For example the grouping operators, will annotate lists with the grouping criteria. In Figure 4 at the top of the tooltip box we find a note that "dept == dept6." This annotation helps the user quickly identify the reason a group was formed.

[2] Strictly speaking, we started with the graphical one implemented in the YaTiSeWoBe system.

A last feature that allows for integration between visualizations and the interpreter are contextualized menus. When the user right clicks on text or on items in the visualization a popup menu appears with items specific to the selected items. For example, for nodes or groups of nodes, users can select the style of the nodes. For edges the menu may include a setting for edge width. These menus enable quickly modifying the visual aspects of graph elements without extensive typing. In all menus users have the option to define a variable name in the interpreter with the contents of the selected item. A user may select a number of nodes in the graph, right-click for the menu, and set the variable "foo" to the content of the selection. Subsequently, commands like *foo.color = black* would be understood and executed by GUESS.

Handling User "Mistakes" in the Interpreter

In watching our users interact with the system it became apparent that they were frequently overwriting the association between namespace names and objects. For example, a user would type *g = 5* and would lose access to the graph object. This is extremely dangerous in an exploratory system because it is difficult to back out of namespace changes gracefully (at least without extensive modifications to the language subsystem). Instead, we settled upon a simple scheme in which we distinguish between user variables and immutable system variables. Nodes and the graph object, *g*, for example, as well as colors and fields are of this immutable type and an error message is returned if a user tries to modify their content.

Other issues, such as infinite loops are helpfully handled by Java. This ensures that user error does not cause a disastrous crash of the system. Furthermore, when GUESS is run in "persistent" mode all changes to the graph are flushed to disk and can recovered at the restart of the application.

A previous issue was that deleted nodes and edges were gone from the system. In GUESS, nodes and edges that are deleted from the graph become part of a special *_deleted* state which can be queried as all states. Those nodes can be easily re-added to the working state with a simple command.

Building Applications

During the exploration stage users may save their visualizations into a persistent database, a simple file, or export the image into any number of formats (including JPG, PNG, GIF, PDF, EPS, and others). Users may also save a log of their program so that it can be rerun at a later time on either the same data or new graphs. While this is in some cases sufficient (e.g. a user simply wishes to generate an image for paper), there are frequently times where a user would like to build a new application or augment the GUI. Gython, like Jython, can be compiled into Java code and our applications can be used in Applets.

GUESS allows users to create new "toolbars" that are docked either vertically or horizontally in the GUI. Because we are using a modified Jython core, users are able to make use of standard Java widgets while ignoring the messy details of implementing complex event handlers. Users can also make user of functional programming techniques in defining GUI reactions. For example, a user can create a button and have the display center every time the button is clicked by this command:

```
testButton.actionPerformed = lambda event:
center()
```

A slider bar can be used to control which edges are display in the social network example:

```
testSlider.mouseReleased = hideshow
def hideshow:
      val = testSlider.getValue()
      (freq < val).visible = 0
      (freq >= val).visible = 1
```

For users familiar with Java, where they would have to define a listener object to handle mouse events, query databases or filter nodes, this is far simpler. Once the user has completed designing their visualization and/or GUI modifications these can be deployed either as an application or as an applet. Users may define highlighting behavior in the same way, augment menus (both main menus and contextualized popups), and control what happens when items are clicked on (e.g. open a webpage or zoom in).

As we primarily rely on Jython for these features we will not cover the full details of writing GUI extensions and instead concentrate on a few sample applications that illustrate GUESS in action.

SURVEY AND GUESS CASE STUDIES

We are aware of many groups either using or evaluating GUESS in applications ranging from social networks to model checkers to computer networks to biodiversity networks. Some use GUESS independently of other systems, but we are aware of at least one where a simple network interface has been created so that GUESS can respond to remote commands. Below we briefly describe our survey in more detail and conclude by concentrating on two real-world examples of GUESS in use.

Graph Software Survey

Although we have described some of the survey results in the context of the system description it is worth mentioning some other key facts. The survey was collected over a one week period by advertising to a number of mailing lists (the bulk of responses came from SOCNET, a large social-networks mailing list with some from the JUNG and GUESS mailing lists). During this time, 77 users completed at least one portion of the survey. Of the 63 users answering this question, 49 (or 78%) indicated that they had no experience with GUESS at all. Thus the bulk of responses were more useful for understanding user needs and experiences with other graph visualization and analysis packages. Only 37% (28 out of 76) were from a field where programming knowledge was to be expected (e.g. Computer Science and Physics). The rest were from primarily social science fields (e.g. Sociology and Ethnography).

The goal of the survey was primarily to identify issues, both positive and negative, users had with other systems. The bulk of participants had no (49 participants) or very limited exposure (6 participants) to GUESS (of 63).

The survey collected free form answers for three categories of graph tools: Menu-based tools (Pajek, UCINET, etc.), toolkits and programmable systems (JUNG, Graphlet, Prefuse, etc.), and math systems with graph functions (Mathematica, R, etc.). Participants indicated which systems they had used and were prompted to enter positive and negative impressions. These responses were coded and appear throughout the paper. Positive responses tended to be short statements concerning availability or price (free was always good). Participants were far more willing to provide negative reactions. These results are encouraging for us as they confirm a number of our decisions. For the menu-based tools the top complaints were (of 47 respondents): complex interfaces (17 instances), not batch oriented/extensible (11 instances), and issues with data formats

(5 instances). Fewer respondents (31) were familiar with toolkit systems and the only repeated complaint was on the learning curve (4 instances).

Because so many different systems exist for visualizing and analyzing graph structures we feel that these responses are mostly useful for identifying issues that GUESS can improve on. We feel that a better indication of GUESS' usefulness is in the positive responses in real case studies.

Proximity Estimation and Load Transfer

One of the groups to make use of both Zoomgraph and GUESS is the Networking Research Group at HP. One of their current projects is the estimation of network latencies in large networks [31]. Because pair-wise computation of latency is very time consuming, the authors have developed algorithms for estimating which network nodes are closest (in terms of latency) to any source computer. In order to demonstrate the effectiveness of the algorithm they have used GUESS to generate a custom visualization (Figure 5a).

Their implementation loads the fully connected graph (all pair-wise connections). When a user selects a node that node becomes the center of a star graph with all neighboring nodes around them. All other nodes and edges are removed from the graph. The connected nodes are laid out in a circle around the central node at a distance proportional to the real network latency. Those nodes that are predicted to be close are colored differently based on the predicted distance.

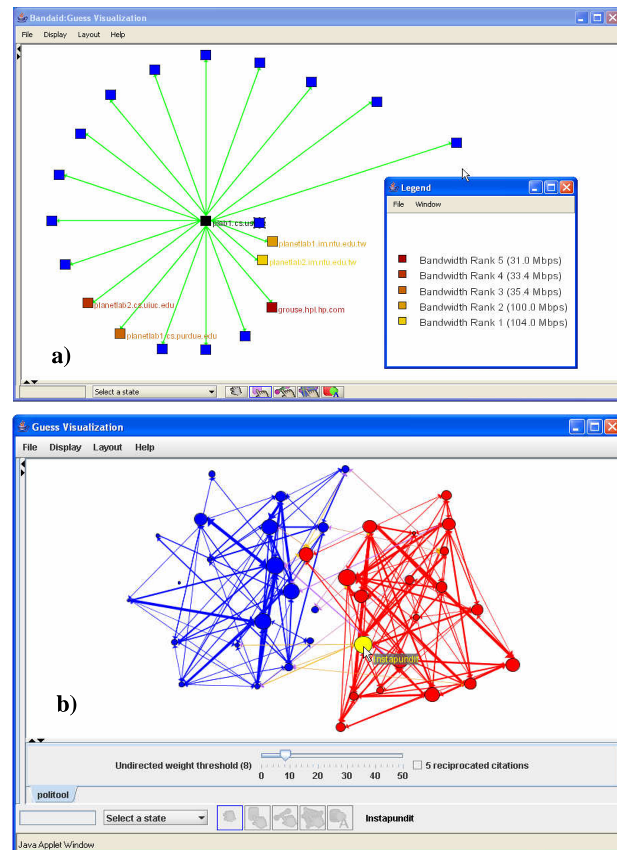

Figure 5a,b: Figure 5a is a screenshots of a network estimation algorithm. Figure 5b captures the political weblog visualization applet (www.hpl.hp.com/research/idl/demos/politicalblogdemo.html). Note the additional "toolbar" at the bottom which was added by the user.

One of the requests of users of Zoomgraph was the ability to create a legend. This feature is now available in GUESS. Users may create as many legends as they want and "insert" node, edges, or convex hulls into the legend with some textual annotation. The legend used in this visualization indicates how colors correspond to latency values. Users of the visualization can quickly assess the quality of the prediction and understand where and how failures happen.

A different visualization for this group demonstrates a resource allocation system in which work on overloaded computers is moved to other resources. This tree based visualization is created dynamically through simulation. As a node in tree becomes overloaded it is divided into two other nodes with the edge being labeled with "reason" for the split.

Both visualizations were built in few days of part time effort and were well received in demonstration sessions.

Political Weblog Network

After the US elections in 2004 there was a great deal of interest in the web research community in the study of online social structures in the political context. A recent project attempted to understand political webloggers [1], and specifically to compare liberal and conservative blog network structures.

By using GUESS, the authors of the study were able to create high-quality, static, visualizations of these networks for their paper. They were also interested in making the data and the visualizations available to their readers.

The graph and a new toolbar were deployed with GUESS as an applet. Figure 5b is a screenshot of this visualization. The graph was laid out using a GEM layout which clearly separates the red (conservative) and blue (liberal) nodes. Nodes are sized according to their in-degree and edge width is defined by the number of reciprocated citations between the blogs. Edges internal to the liberal side are colored blue, and similarly those on the conservative side are red, with those that cross in yellow. Up to this point the program is 8 lines of code.

Nodes and Edges may have event handlers attached to them that wait for the mouse to move over or the mouse button to be clicked. In this case, the author has elected to respond to shift-clicks on nodes by opening up a web page with the associated blog homepage. The toolbar at the bottom of the screen allows users to select a threshold (i.e. the number of times one blog links to another) for displaying links. As the slider is moved to the right edges that do not meet the threshold are hidden. In all, the script is under 60 lines of code.

CONCULSIONS AND FUTURE WORK

In this paper we have introduced the GUESS system. We have illustrated how the Gython language can be used for exploratory data analysis of graph structures. We believe that the design decisions we have made in language design make the tool useful for rapid exploratory data analysis. We also believe that the use of a targeted, domain-specific language is broadly applicable in the design of direct manipulation systems in which fine grained user control is desirable. Furthermore, the GUESS system presents an environment for users with arbitrary graph datasets and a single mechanism for generating static and dynamic visualizations as well as applications. Finally, we believe that our GUI design represents a novel integration of textual and

visual interaction which can be applied to other visualization systems where multiple modes of interaction are necessary.

At present, our major goals are to improve the system's ability to handle dynamic graphs and to add path finding semantics that provides users with a more flexible grammar for finding graphs.

As we continue to work with users we hope to be able to generate custom interfaces for different users. By working with biologists or social scientists we hope to determine which features they find most useful for their tasks and provide a custom view into GUESS that supports those needs.

AVAILABILITY

GUESS is available at: http://www.graphexploration.org

ACKNOWLEDGEMENTS

The author would like to thank David Feinberg and Joshua Tyler without whom GUESS could not have been built. Additional thanks to Bernardo Huberman, Lada Adamic, the Netvigator team, and all the users of Zoomgraph and GUESS. A great deal of thanks to the implementers of the free software which we were able to use in this system. Finally, thanks to Dan Weld and Kayur Patel for useful comments on this paper.

REFERENCES

[1] Adamic, L.A., and N. Glance, "The Political Blogosphere and the 2004 U.S. Election: Divided They Blog," 2nd Annual Weblogging Workshop, WWW 2005, Chiba, Japan, May 10, 2005.

[2] Adar, E., and J.R. Tyler, "Zoomgraph Manual," www.hpl.hp.com/shl/projects/graphs/doc/zg-manual.htm

[3] Aiken, A., J. Chen, M. Liu, M. Spalding, M. Stonebraker, and A. Woodruff, "The Tioga-2 Database Visualization Environment," *IEEE Vis. '95 Workshop. on Database Issues for Data Vis.,* Atlanta, Oct. 1995, pp. 181-207

[4] Batagelj, V., and A. Mrvar, "Pajek – Program for Large Analysis," *Connections*, 21:47-47, 1998.

[5] Bederson, B.B., J. Grosjean, and J. Meyer, "Toolkit Design for Interactive Structured Graphics," *IEEE Transactions on Software Engineering*, 30(8):535-546.

[6] Berry, J., N. Dean, M. Goldberg, G. Shannon, and S. Skiena, "Graph Drawing and Manipulation with LINK," *Lecture Notes in Computer Science 1353: Graph Drawing 1997*, G. Di Battista (ed.), Springer, 1997.

[7] Bosch, R., C. Stolte, D. Tang, J. Gerth, M. Rosenblum, and P. Hanrahan, "Rivet: A Flexible Environment for Computer System Visualization," *Computer Graphics*, 34(1), 2000.

[8] Di Battista, G., P. Eades, R. Tamassia, and I. G. Tollis, *Graph Drawing: Algorithms for Visualization of Graphs*, Prentice Hall, 2002.

[9] Gansner, E.R., and S.C. North, "An open visualization system and its applications to software engineering," *Software – Practice and Experience*, 30(11):1203-1233, 2000.

[10] "The GraphML File Format," graphml.graphdrawing.org

[11] Güting, R.H. "GraphDB: Modeling and Querying Graphs in Databases," VLDB '94, Santiago De Chile, Chile, Sep. 12-15, 1994.

[12] Gyseens, M., J. Paredaens, J. Van den Bussche, and D. Van Gucht, "A Graph-Oriented Object Database Model," *Proceedings of the 9th Symposium on Principles of Database Systems*, Nashville, TN, 1990.

[13] Heer, J., S. K. Card, and J. A. Landay, "Prefuse: A Toolkit for Interactive Information Visualization," CHI 2005, Portland, OR, April 2-7, 2005.

[14] Himsolt, M., "Graphlet: design and implementation of a graph editor," *Software – Practice & Experience*, 30(11):1303-1324, 2000.

[15] Himsolt, M., "GraphEd: A Graphical Platform for the Implementation of Graph Algorithms," *Lecture Notes in Computer Science: Graph Drawing 1994*, Springer, 1994.

[16] JGraph homepage, http://www.jgraph.com

[17] JUNG homepage, http://jung.sourceforge.net

[18] Jython homepage, http://www.jython.org

[19] Koutsofios, E. and D. Dobkin, "Lefty: A two-view editor for technical pictures", Graphics Interface '91, Calgary, Alberta, 1991, pp. 68-76.

[20] Livny, M., R. Ramakrishan, K. Beyer, G. Chen, D. Donjerkovic, S. Lawande, J. Myllymaki, and K. Wenger, "DEVise: Integrated Querying and Visual Exploration of Large Datasets," *Proceedings of ACM SIGMOD*, 1997.

[21] Melhorn, K., and S. Naher, "LEDA: a platform for combinatorial and geometric computing," *Communications of the ACM*, 38(1):96-102.

[22] Misue, K., P. Eades, W. Lai, and K. Sugiyama, "Layout Adjustment and the Mental Map," *Journal of Visual Languages and Computing*, 6(2):183-210, 1995.

[23] Mutzel, P., and M. Junger, *Graph Drawing Software*, Springer-Verlag, 2003.

[24] Page, L., S. Brin, R. Motwani, and T. Winograd, "The PageRank Citation Ranking: Bringing Order to the Web," Stanford Technical Report, 1998.

[25] Stolte, C., D. Tang, and P. Hanrahan, "Polaris: A System for Query, Analysis, and Visualization of Multidimensional Relational Database," *IEEE Transactions on Visualization and Computer Graphics*, 8(1), 2002.

[26] Swayne, D.F., B. Andreas, and D.T. Lang, "Exploratory Visual Analysis of Graphs in GGobi," Workshop on Distributed Statistical Computing (DSC 2003), Vienna, Austria, March 20-22, 2003.

[27] Tom Sawyer Software, http://www.tomsawyer.com

[28] TouchGraph, http://www.touchgraph.com

[29] Tukey, J., *Exploratory Data Analysis*, Addison-Wesley, 1977.

[30] Wasserman, S., and K. Faust, *Social Network Analysis*, Cambridge University Press, 1994.

[31] Xu, Z., P. Sharma, S. Lee, and S. Banerjee, "Netvigator: Scalable Network Proximity Estimation," HP Laboratories Technical Report, HPL-2004-28, Feb. 2004.

[32] yEd by yWorks, http://www.yworks.com