Visigoth: Visualizing the Gnutella Overlay Topology

CIS 422/522 Project 1, Winter 2007

Michal Young

Revision 1.5, produced January 8, 2007

Abstract

The first of two projects for CIS422/522 in Winter term 2007 is visualization support for Gnutella topology data. The project is due on Friday, 2 February at 5:00pm and so must be completed in approximately four weeks.

The data sets to be visualized were produced by a networking research project at UO. Given the tight schedule, the only realistic approach is to transform these into a form suitable for one or more existing data visualization tools. Features of the software to be produced includes transforming the format of the data sets, selecting portions of the data for display, selecting attributes to display, and controlling the visualization.

Among the most important goals of the project are to make these features flexible and extensible for *exploratory* data analysis. This places a very high premium on the *architectural design* of the visualization support.

Contents

1 Introduction **3**

2 Users and Tasks **3**
2.1 User characteristics 3
2.2 User tasks 5
2.2.1 Scenario: Exploring connectivity 5
2.2.2 Scenario: Adding a Graph Analysis 6

3 Required and Desirable Features **7**
3.1 Data Input 8
3.2 Data output 8
3.3 Data manipulation 8
3.3.1 Selection and Filtering 8
3.3.2 Attribute computation 9
3.4 Performance and Capacity 9
3.5 Robustness 10

4 Architecture Notes **10**
4.1 Existing Visualization Tools 10
4.2 Input Processing 11
4.3 Configurability and Extensibility 11

5 Deliverables **12**
5.1 Source Distribution 12
5.2 User Manual 13
5.3 Developer Documentation 13

1 Introduction

CIS 422/522, Software Methodology I, is a team project-oriented course. We complete two projects in the first 9 weeks of a 10-week term. The first of the two projects, as well as teams for that project, are chosen by the instructor. The second of the two projects, as well as teams, are chosen by students themselves (although I may intervene in some circumstances). The first project is given approximately 4 weeks, and the second project 5 weeks. (Week 10 is reserved for presentations and recovery.)

The first of two projects for Winter term 2007 is data visualization support for analysis of the Gnutella peer-to-peer network. The data to be visualized was captured by the Mirage research group at University of Oregon. The purposes to which visualization will be put are primarily exploratory, meaning that there is no single "right" visualization, but rather a range of questions that network researchers may seek insight into using visualization.

A product for which the primary users are researchers is a bit unusual for this course, and I regard it as an experiment. A risk of such a project is somewhat less clarity and stability in the definition of requirements for product features. On the positive side, such a project puts extraordinary demands on the architectural design, and somewhat less on the initial product feature set, which fits well with my pedagogical objectives for this course.

At least two students at UO have previously worked on visualization of the Gnutella data. Figures 1 and 2 illustrate this prior work.

Sections 2 and 3 comprise a preliminary requirements statement for Visigoth.

2 Users and Tasks

2.1 User characteristics

The primary users of the system for visualizing the Gnutella overlay topology (henceforth Visigoth) are networking researchers studying characteristics of peer-to-peer networks. The intial end users are members of the Mirage networking research group at University of Oregon, but a successful system should be usable by researchers elsewhere using data sets published by the Mirage group.

In addition to *end users*, Visigoth projects will be made available to participants in the peer-to-peer visualization seminar to be held in Spring 2007. Participants in this seminar are primarily graduate students with an interest in networking

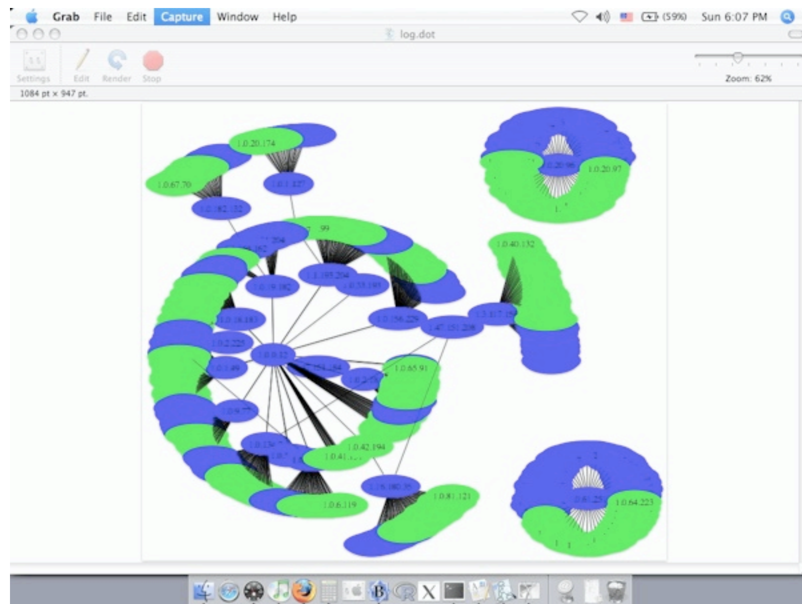

Figure 1: Linda Sato's Gnuviz, described at `http://www.uoregon.edu/~lsato/gnuviz.html`

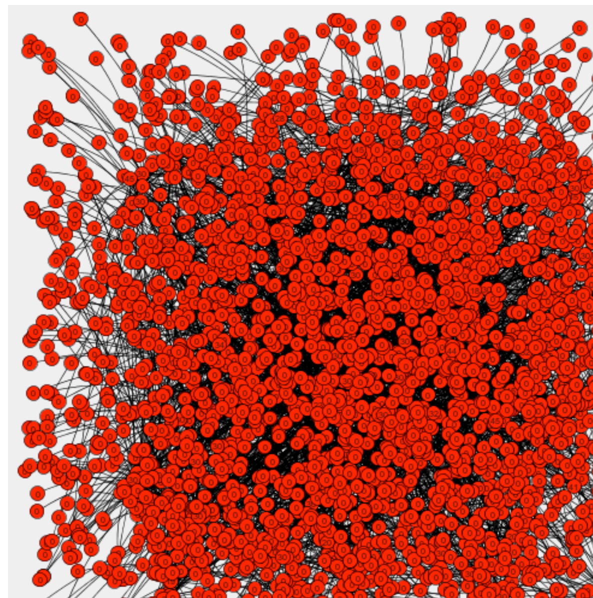

Figure 2: This 200-node diagram from Jason Lam shows how difficult to read even moderately complex graphs can be.

and/or data visualization. Seminar participants may use Visigoth to understand capabilities and limitations of existing visualization tools, to get ideas for the tools they may construct, and to understand characteristics of the data sets and their impact on visualization tools. In the best case, one or more of the visualization systems produced by CIS 422/522 students will be suitable as a starting point for projects constructed by seminar students.

Typical interactive use of Visigoth will be with conventional computer display devices, with resolutions less than 2000 by 2000 pixels. However, the researchers using Visigoth will have occasional need for output in other formats. One of these is small but high resolution vector graphics in printed form, for inclusion in technical research papers. Another is very large format display devices, such as arrayed displays or arrayed projectors.

2.2 User tasks

These tasks are described in the form of scenarios. It is not required that interaction with Visigoth follow exactly the steps described here. Rather, these scenarios are intended to illustrate user needs, which the software system may meet in different ways.

2.2.1 Scenario: Exploring connectivity

The Gnutella overlay network classifies participating hosts as leaves, peers, and ultrapeers.

A network researcher wishes to better understand the connection structure of ultrapeers. Because the graph is very large, it is not possible to view the entire topology at once. The researcher begins by configuring Visigoth to

- select a focus ultrapeer node at random.
- limit the display to 20 ultrapeer nodes, 50 total nodes, and 100 edges.
- condense leaf and peer nodes to connection counts (i.e., instead of showing 5 individual links to leaf nodes, display a single link to a single leaf node labeled "5").
- display ultrapeer nodes whose distance from the focus node is less than four.
- display IP address and port of each ultrapeer node.

On the first attempt, Visigoth displays 20 ultrapeer nodes, including 19 at distance 1 from a randomly selected focus node, and reports that 14 additional nodes at distance 1 could not be displayed because they exceeded the limit on ultrapeer nodes. Only a small portion of the resulting display fits on the display screen.

The researcher modifies the configuration parameters to select up to 40 ultrapeer nodes but to display them as dots, except that the selected focus node is an oval labeled with its IP number (but not port). The researcher re-executes Visigoth and obtains a more useful graph display. Re-executing it several more times and panning the graphical display in the window, she is almost always able to see at least all the immediate neighboring ultrapeer nodes, and to notice that many of the immediate neighbors are also neighbors of each other.

The researcher saves the configuration script and walks down the hall to the distributed system visualization laboratory. The large stereoscopic display is not currently in use, so she logs in and begins to use it. She loads her saved configuration into Visigoth.

Finding that the high resolution, three-dimensional display can accomodate many more nodes than her conventional computer screen, she increases the limit on ultrapeer nodes to 50 and the limit on edges to 300. For the next 30 minutes she executes the visualization script several times to see a variety of different ultrapeers, spending a few minutes manipulating the display for each selection.

During the interactive exploration she saves three selections for later use in a paper. These will be re-displayed using different display parameters for use in a paper, but using exactly the same set of nodes and edges.

2.2.2 Scenario: Adding a Graph Analysis

A network researcher is interested in studying the characteristics of paths from one node to another. Of particular interest is when some intermediate node lies on all good paths between them — she speculates that nodes whose best communication paths are forced through these bottlenecks may have less reliable communication than nodes that have many choices among equally good communication paths. This is close related to the notion of *dominators* from graph theory, which is widely used in program analysis. She decides to define k-domination as follows: Node D k-dominates the pair (A, B) iff D lies on all paths from A to B with length at most k.

The researcher wants to select pairs of nodes, find their distance d, (the length of the shortest path between them), and then display all minimum length paths between them with d-dominators highlighted. However, since she has just defined k-

dominators, there is no current Visigoth feature for selecting k-dominating nodes for a given path. She must create one.

Alternative sub-scenario A: The researcher begins with a sample component for graph analyses. Choosing among samples written in Python, C++, and Java, she chooses the Python version because she needs to quickly evaluate the concept, and she does not anticipate performance problems. She makes a copy of this component, keeping the interfaces for reading the intermediate graph format, transforming it into an in-memory format, traversing and annotating it, and writing back the intermediate graph format. She writes a page or so of Python code for the new analysis, names her new component, and adds it to the Visigoth component collection.

She begins by testing the component on a few small data sets, noting failures (both crashes and incorrect results), and editing her component. When the component seems to be working correctly, she begins to use it as intended on the real Gnutella data. She also writes basic documentation so that other researchers can use her new graph analysis component.

Alternative sub-scenario B: The researcher opens the Visigoth scripting manual and looks for examples as similar as possible to her needs. After studying these examples, she creates a k-dominator identification script and installs it in her Visigoth script library.

She begins by testing the script on a few small data sets, noting failures (both Visigoth error messages and incorrect results), and editing her script. When the component seems to be working correctly, she begins to use it as intended on the real Gnutella data. She also writes basic documentation and places her new script and usage instructions in the shared library of Visigoth graph analysis scripts.

3 Required and Desirable Features

From the scenarios above, we can identify several required features and several that, though not absolutely required, are very desirable. For brevity below, we use the word "shall" to indicate something that is required, and "should" to indicate something that is not required but is highly desirable.

Required and desirable features are numbered in a single sequence. These numbers may change as requirements are added and rearranged. All references to

these requirements (in design documents, for example), should be by description, not by number.

3.1 Data Input

• *R1* Visigoth shall read data in the format provided by Gnutella crawler.

• *R2* The original attributes of a node include IP number, port, client program, and class (leaf, peer, ultrapeer, or failed).

3.2 Data output

• *R3* Visigoth shall produce graphical visualizations of network topology.

• *R4* Visigoth shall label nodes with host IP#, or not, at the user's option.

• *D5* Visigoth should label nodes with any original or computed attribute, at the user's option.

• *D6* Visigoth should select different display attributes, including color, shape, size, and labeling, depending on original or computed attributes.

• *D7* Visigoth should be compatible with conventional computer displays, printing, and very high resolution display devices.

3.3 Data manipulation

3.3.1 Selection and Filtering

Filtering means removing nodes or edges that do not meet a criterion from the data set, and selection means designating a node or nodes for some particular role (e.g., a focus node for a neighborhood display).

• *R8* Visigoth shall apply user-specified filtering criteria to nodes, based on original or computed attributes.

• *R9* Visigoth shall apply user-specified capacity filters, e.g., limiting the number of displayed nodes or displayed edges. Triggering of capacity filters (i.e., if other

filtering rules would result in displaying more than the user-specified number of nodes or edges) shall be reported to the user.

• *D10* Visigoth should allow filtering of nodes based on a wide range of user-specified criteria.

• *R11* Visigoth shall apply user-specified selection criteria, including at least the ability to choose a focus node uniformly at random from the data set.

• *D12* Visigoth should allow selection of nodes meeting a wide range of user-specified criteria involving original and computed attributes.

3.3.2 Attribute computation

• *D13* Visigoth should, at user option, compute a variety of new attributes of nodes and edges from the original attributes.

• *D14* Visigoth should support creation of new nodes and edges (e.g., condensing leaf nodes as in the first scenario).

• *D15* Visigoth should support ranking of nodes and edges based on user specified criteria (e.g., the 20 closest neightbors to node N, or the 20 immediate neighbors having maximum out-degree).

• *D16* Visigoth should make available new, user-written attribute computations through the same interface as the standard attribute computations provided by Visigoth.

3.4 Performance and Capacity

The data sets describing Gnutella topology at different points in time are rather large. Actually visualizing a whole data set is not practical, but Visigoth must at least be capable of processing a whole data set and then displaying a small part of it.

• *R17* The wall time required for Visigoth to read and process an input file with at most 500,000 nodes and 10,000,000 edges, selecting at most 100 nodes and 100 edges for display, shall be under 2 minutes on a typical desktop computer.

• *D18* Nearly all simple visualizations should complete in under 10 seconds.

3.5 Robustness

These criteria are listed in order of increasing robustness. A production-quality system should achieve all four.

• *R19* Visigoth shall not crash or hang when executing any script or configuration that conforms to documented usage rules.

• *D20* Except for internal errors in user-written components, Visigoth should not crash or hang regardless of errors in user instructions (e.g., scripts and configuration parameters).

• *D21* Visigoth should catch and report even failures that occur within user-written components and extensions to Visigoth.

• *D22* Visigoth should provide debugging support to aid in detecting and diagnosing errors in user-written scripts and configurations.

4 Architecture Notes

Considering the second scenario above, in which a user creates a new analysis to incorporate into Visigoth, it should be clear that an architecture designed for extensibility is particularly crucial, and is probably the most challenging aspect of this project. A requirements document does not normally prescribe the design of a software system, but may impose a number of requirements that make some designs more suitable than others.

My notes here on the architecture of Visigoth do not limit the approaches you can take, provided your approach is better than what I describe in at least one respect and not horribly worse in others. You should carefully reason about your design choices, and explain them, whether you follow my suggestions or not.

4.1 Existing Visualization Tools

You should almost certainly use an existing graph visualization tool for display. Tools you might consider include

Graphviz	`http://www.graphviz.org`
H3Viewer	`http://graphics.stanford.edu/papers/h3/`
Otter (2D)	`http://www.caida.org/tools/visualization/otter/`
Walrus (3D)	`http://www.caida.org/tools/visualization/walrus/`
Tulip	`http://www.tulip-software.org/`

Ideally you would have interfaces to multiple display tools. For example, you might give the user a choice between displaying with graphviz, davinci, or another tool. This would be particularly important for moving between a conventional computer display and other display technologies, such as the high resolution stereoptic 3D display in the distributed systems visualization lab. Note that this implies that neither the input formats nor the capabilities of any particular display tool is wired deeply into the guts of Visigoth. Transforming the internal form into the form required by a particular visualization tool should be a well-encapsulated module.

4.2 Input Processing

For the present, the input format of the Gnutella trace log files is the only input format you are required to accept. Ideally, though, that should also not be too deeply wired into your system. It would be better if there is a well-defined graph-reading interface into which other readers can be inserted.

4.3 Configurability and Extensibility

There are at least two ways to achieve the flexibility and extensibility described above. One is to structure your system as an interpreter. One module reads an input file, another emits output in the right format for an existing visualization engine (such as the graphviz tools), and in between you manipulate an in-memory data structure by interpreting a scripting language. The operations in your scripting language are small modules with a well-defined interface to the shared graph data structure, and it is easy to construct and incorporate additional operators. Although building an interpreter can be a pretty major undertaking, you can keep it simple by using a very simple syntax (e.g., postfix) or by embedding an existing interpreter.

The second way (which I recommend) breaks the system down into a number of small programs that communicate through intermediate files. There is no shared in-memory representation of the graph, but there should be a single, fixed intermediate representation of the graph that passes from program to program. It is still necessary to interpret some kind of configuration specification to chain these steps together.

If you choose the first approach — interpreting scripts to manipulate a single in-memory representation — then you should probably write the whole program in one language, and that language should probably be something like Java or C++

(though Python might do). If you choose the second approach, you'll probably want to write at least the first draft of most components in a scripting language like Awk, Perl, or Python.

The second approach will definitely be slower to execute than the first, because file I/O is slow compared to computation. However, it will be easier to debug, provided you design a human-readable intermediate data format or build tools to inspect the intermediate form. It is more difficult to assign blame to one among many modules manipulating the same data structure in memory, no matter how righteously modular and object-oriented your data structure code is.

5 Deliverables

5.1 Source Distribution

The source distribution is a single archive (.tgz or .zip file) containing all parts of the system in source form. The archive should expand into a single directory. It should contain

- Source code of all programs, except external components that must be obtained and installed separately. This includes source that does not become part of the end-user programs, such as Makefiles, debugging scripts, configuration scripts, etc.

- All user and developer documentation in its source form. If the source form is not an open format editable with free and widely available tools, then open format derived versions of documents must also be provided. For example, if you use MS Word to create a user manual, then in addition to the .doc file (which is not in an open format) you should include a PDF, HTML, or plain text version of each file.

  See below for more on required developer documentation.

- Test suites and scaffolding, with test documentation. Unit, subsystem, and system tests should be as automated as possible, and should be portable (i.e., they need to run for developers who install your system in their own environment).

- Project web site (optional but recommended).

5.2 User Manual

A manual for end users is required. It may be divided into a tutorial and a reference manual. A single combined user manual (tutorial and reference together) is also possible, provided it is usable in both ways.

A tutorial is task-oriented and proceeds from the simplest, most common tasks up through more complex and less common tasks. Each task is described step-by-step, with examples. A tutorial often omits some options for the sake of simplicity. Tutorials are usually read sequentially, starting from the beginning, although a user may skip over sections that appear to be irrelevant to his or current task.

A reference manual describes each feature and option of the system clearly and precisely, with examples. It is arranged or indexed to accomodate random access, and is seldom read sequentially.

Both tutorials and reference manuals should include a short introductory section describing the purpose of the described system and any assumed background knowledge of users.

If the system is designed for installation by end users, there should also be a separate installation manual, or else installation instructions can be the first part the tutorial. (See under developer documentation below for more on installation instructions.)

Often a system has more than one set of users. For example, an email system might have end users and system adminstrator users. Usually each user audience will need its own set of documentation.

5.3 Developer Documentation

Developer documentation should include, at a minimum:

README.txt: This plain text file should appear at the top level of the project directory (i.e., in the top level directory expanded from the project archive file). It should contain, at a minimum:

- The name of the system
- A brief description of the purpose of the system. Who should use it, and for what? This should be very brief, but enough to identify the system to a user.
- Version and date.

- Environment requirements (briefly, but enough so, for example, a Windows user doesn't waste a lot of time trying to install and use a Unix program.)
- A list of the authors (developers), with contact information for at least one.
- A guide to other developer documentation. For example, it might say that the user manual is found in the "manual" subdirectory and all other developer documentation is found in the "doc" directory.

Other information that typically appears in a README file includes license terms (please use an open source license, such as BSD, Apache, or GPL) and acknowledgment of sources (especially, but not only, if the current project is a modification or extension of another open source system.)

Requirements description: A clear description of the problem addressed by the software. (Sections 2 and 3 are a preliminary requirements statement and can serve as the first draft of your requirements statement.)

System requirements specification: Describes precisely what the system will do. If the user reference manual is sufficiently detailed, it may serve as part of the system specification. (In that case, the specification document would include the user manual by reference.) The requirements specification should also indicate what part of the requirements are met, what has been deferred to the future, and (as appropriate) what part of the perceived user requirements are omitted, with a rationale for those choices.

Note the distinction between a requirements description and a requirements specification: The former describes what is needed, and the latter describes what is to be constructed. A "requirements statement" may include both in one document, but both purposes are important and should not be confused.

System architectural design: Describes the overall organization of the system, including criteria for breaking it into subsystems and modules, as well as the resulting breakdown.

Detailed design: Much of the detailed design of an implementation is typically provided within the code itself, in comments, and is often processed into more usable form with a tool like javadoc.

How can you know that the architectural design and detailed design documentation are sufficient? The key is that together they must be adequate for a programmer otherwise unfamiliar with the system to plan and execute a modification. The architectural design document should be enough to determine whether a given modification is likely to be possible and approximately how difficult it will be, and which source files are likely to require changes. (A good architectural design minimizes the number of parts of a system that must be changed to make the most common changes.) The detailed documentation should confirm the practicality of the change and be enough (together with the architectural design) to plan and execute it. In short, the maintainer should be able to figure out where to look, and then find the information he needs there.

Anticipated changes: This may be a section in the requirements specification (anticipated changes in user needs) or, in the system specification (anticipated changes in system features), and in the architectural design (how anticipated changes will fit in the design). For example, if we anticipated a need to use the system in Senegal, we might anticipate adding an option to give all error messages in French, and we might note in the architectural design that all messages are routed through the Messages module, which produces message text from an external text file that can be replaced at system compile time with a file containing the same messages in a different language.

Build instructions: How to create a running system from the source files. Note that building is not the same as installing, so these are different instructions. Good build automation can make the build instructions very simple, perhaps simple enough to include in the README file.

Test documentation: Describe the overall test plan and provide instructions for unit and system regression testing after changes.

Manifest: A guide to all the files in the distribution. A first level manifest may be placed in the README file if it is not too large; otherwise it should appear in a separate document.

Acknowledgments. The idea for this project, and information on available graph drawing utilities, was provided by Prof. Virginia Lo. The Gnutella data was collected by Dr. Daniel Stutzbach in the Mirage networking lab under the direction of Prof. Reza Rejaie.