A Graph Drawing and Translation Service on the WWW*

Stina Bridgeman, Ashim Garg and Roberto Tamassia

Department of Computer Science
Brown University
Providence, RI 02912–1910, USA
{ssb,ag,rt}@cs.brown.edu

Abstract. Both practitioners and researchers can take better advantage of the latest developments in graph drawing if implementations of graph drawing algorithms are made available on the WWW. We envision a graph drawing and translation service for the WWW with dual objectives: drawing user-specified graphs, and translating graph-descriptions and graph drawings from one format to another. As a first step toward realizing this vision, we have developed a prototype service which is available at `http://loki.cs.brown.edu:8081/graphserver/home.html`.

1 Introduction

Motivated by numerous applications, new graph drawing algorithms are continually being developed. By making implementations of graph drawing algorithms available on the WWW, we can help both practitioners and researchers to use the latest technological innovations. This, however, also requires tackling the problem of the over-abundance of formats for describing graphs and drawings. While there are efforts in this direction [3], there is still no single universally-accepted format. Therefore, researchers typically define their own formats when implementing an algorithm. Because a user cannot be expected to know the format used by each and every implementation, it is advantageous to have translators that can convert the descriptions of graphs and drawings from one format to another. This will allow users to employ a large number of algorithms while knowing only a few formats.

We envision a graph drawing and translation service on the WWW that offers two kinds of services:

- a *drawing service* for constructing a drawing of a graph given by the user, using an algorithm chosen by her, and
- a *translation service* for translating the description of a graph/drawing from one format to another.

Such a service would benefit both practitioners and researchers. From a practitioner's viewpoint, she gets a central facility from where she can "shop around"

*Research supported in part by the National Science Foundation under grant CCR–9423847 and a graduate fellowship, by the U.S. Army Research Office under grant DAAH04–96–1–0013, and by a gift from Tom Sawyer Software.

for an algorithm appropriate for her application. From a researcher's viewpoint, she gets a repository of algorithms where she can study and compare their properties and performances. In addition, both practitioners and researchers can use this service just for translation purposes. Potential uses of this service include:

- drawing graphs from user-applications,
- studying and comparing graph drawing algorithms,
- translating between the formats for describing graphs and their drawings,
- creating a database of graphs occurring in user-applications, and
- demonstration purposes in educational settings.

Researchers have recognized the need of making implementations of graph drawing algorithms available on the WWW. A bibliography (and URLs) of many implementations is maintained by Georg Sander at `http://www.cs.uni-sb.de/RW/users/sander/html/gstools.html`. Stephen North has designed a service (visit `http://www.research.att.com/dist/drawdag/mail_server` for details) that accepts a graph sent to it by email and returns, also by email, a drawing constructed using *dot* [4]. This service also maintains a data base consisting of the graphs sent to it by the users [5].

We have developed a prototype graph drawing and translation service on the WWW. To the best of our knowledge, there is no other translation service over the WWW, and the closest approximation to a WWW graph drawing service is the email service provided by Stephen North. The other implementations available on the WWW are source code packages/executables which must be downloaded and installed locally, rather than being interactive servers running on a remote machine.

Our service is located at `http://loki.cs.brown.edu:8081/graphserver/home.html`, and is very easy to use. The user can interact with the service in two ways, by using an HTML form, or a Java-based graph editor embedded in a Web page. In a typical scenario, the user provides the server with the input graph, and selects the format of the input, the type of service desired (drawing/translation), the format of the output, and the drawing algorithm (if the graph is to be drawn). The server receives the request, performs the desired service, and sends back the result — a drawing or a translation — to the user.

In Section 2, we present the software architecture of our prototype service. In Section 3, we give some example interactions between users and our prototype service. Finally, in Section 4, we describe future work.

2 Software Architecture of our Prototype Service

We believe that a graph drawing and translation service should satisfy the following requirements:

- The input/output format used for a graph should be as independent as possible from the algorithm used for drawing it.
- It should be easy to add new formats and algorithms to the service.

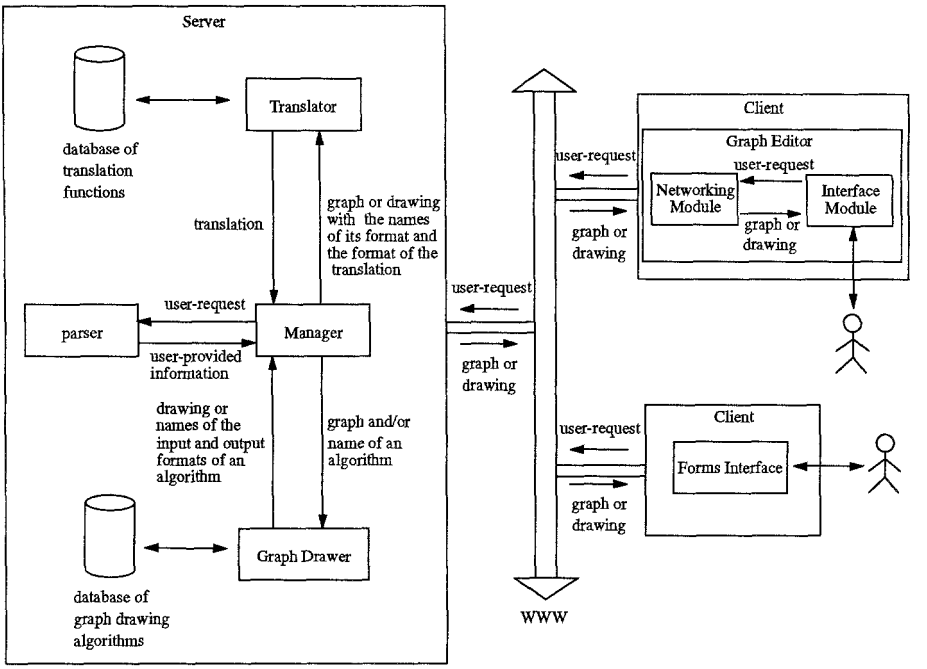


Fig. 1. The software architecture of our graph drawing and translation service.

- The user interface should be simple and intuitive.
- The architecture should be client-server based with the assumption that the client machine and the network may be slow, whereas the server is reasonably fast. This assumption is important for a network such as the WWW.

Figure 1 shows the software architecture of our prototype service. The service has five main logical components: the *client*, the *manager*, the *parser*, the *translator*, and the *graph drawer*. Only the client component runs on the user's machine; the other components all reside on the server.

Client. The client is responsible for maintaining the interface presented to the user. There are currently two versions of the client — the forms interface and the graph editor.

Forms Interface. The forms interface is simply an HTML page with a form that the user can fill to place her request. No code is needed to handle server communications or manage the form, as this is done by the user's Web browser.

Graph Editor. The graph editor is implemented as a Java applet and consists of two modules, the *interface module* and the *networking module*. The interface module is responsible for creating and managing the graph editor window, supporting basic graph editing operations such as insertion, deletion, and movement of vertices, edges, and edge bends, and enabling the user to interactively create a graph to be sent to the server. The networking module is responsible for communicating with the server — it sends the request, provides the interface with

updates on the current status of the request, receives the response, and ensures that if the user sends a request, a response will be received even if the server goes down.

Manager. The manager is the central component of the service, and is responsible for coordinating the parser, translator, and graph drawer. It is multithreaded to handle potentially simultaneous requests from multiple clients, and typically spawns many threads for handling the user requests.

Parser. The parser receives from the manager a character string that contains a user request and extracts from it the following information: graph to be drawn, format of the input, type of service desired (drawing/translation), format of the output, and drawing algorithm to use (if the graph is to be drawn).

Translator. Each graph drawing algorithm supported by the service defines its own input/output format which may be different from the format of the graph given by the user. Therefore, there may be a need for translating from one format to another on drawing requests. In addition, the user may explicitly request a translation. The translator takes as input a graph or a drawing and the names of the current format and the format to which translation is to be done, and performs the desired translation. It maintains a database of *basic* translation functions. Given a translation request, it constructs a sequence of basic translation functions which is then applied to the input graph or drawing to carry out the desired translation. Notice that because different formats have different expressive powers in general, some loss of information may occur during a translation. Currently, the sequence of basic translation functions needed is explicitly coded so that the problem of translating from a more powerful format to a less powerful one and then back to a more powerful one is avoided.

Our prototype service currently supports the following input formats: *parenthetic*, *GDS*, *Tom Sawyer*, and *MALF*, and the following output formats: *parenthetic*, *GDS*, *postscript*, *GIF*, *gnuplot*, *mif*, *fig* and *MALF*.

The parenthetic format [7] consists of nested lists of keyword-value pairs enclosed by parentheses. It is very flexible in the sense that it allows the users to define their own keywords. GDS, Tom Sawyer, and MALF are minor variations of the formats used by GIOTTO [9], the Tom Sawyer Graph Layout Toolkit, and GD-Workbench [1], respectively. Gnuplot, mif and fig are the formats of `gnuplot`, Frame Maker and `xfig`, respectively.

Graph Drawer. The graph drawer accepts as input the name of a graph drawing algorithm and a graph described in the algorithm's input format, and constructs a drawing of the graph using the algorithm. Currently we support the following three algorithms: GIOTTO, PAIRS, and Planarizer.

GIOTTO [9] is a general-purpose drawing algorithm based on the *planarization* approach and a bend-minimization method [8]. In addition to the original implementation of GIOTTO, we offer a variation of it called GIOTTO-*with-labels*, which expands the vertices to accommodate labels. PAIRS [2] is our implementation of the algorithm of [6], which uses *st*-numberings. Planarizer is not

actually a drawing algorithm: it is the first step of GIOTTO, which constructs a planar embedding of the input graph by replacing crossings (if any) with dummy vertices.

Our prototype service does not require any special hardware or software on a client machine other than a commonly available Web browser. The forms interface requires a browser, such as Lynx or Netscape, that supports HTML forms. The graph editor requires a browser that supports the Java 1.0 API, such as Netscape 2.0 or higher for most platforms. Java support is being added to more browsers so that this requirement is becoming less restrictive.

The server should be reasonably fast with some kind of support for multi-threading, and with sufficient memory for storing the users' graphs and drawings.

Our service satisfies the requirements stated earlier in this section as follows:

- The user is free to specify any input/output format independent of the algorithm requested by her. This decoupling is made possible by the presence of the translator.
- It is very easy to add new formats and algorithms. Also notice that an implementation of an algorithm can define its own input/output format provided that it is equipped with a translator to/from a format already supported by service. We believe that in most cases it will be sufficient to give a translator between a new format and the parenthetic format.
- Both the forms interface and the graph editor are simple and intuitive.
- All the computationally intensive work is done by the server. Hence, the service can be used by clients with limited resources as well.

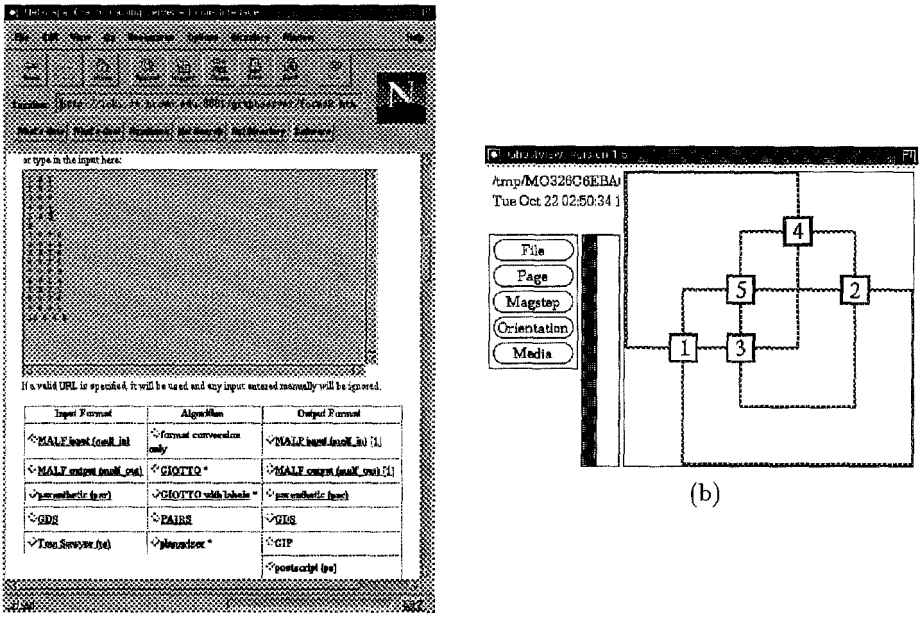
3 Using our Prototype Service

We now give two scenarios to show how a user interacts with our prototype service and how the service satisfies a user request.

Scenario 1. Using the Forms Interface (see Figure 2).

Suppose the user wants to draw a graph described in the MALF input format using the GIOTTO algorithm, and wants the drawing to be in postscript format. She loads the service's home page into her browser, clicks on the *forms interface* link, and gets an HTML form. She can specify a graph in one of two ways — either by giving the URL of the graph or by typing the description of the graph in the form itself. In Figure 2(a) she chooses the latter option and enters the graph (in MALF input format) into the text area provided in the form. She then clicks on the appropriate selection buttons to specify that the input format is MALF input, the drawing algorithm is GIOTTO, and the output format is postscript, and submits the form to the server. Figure 2(a) shows a part of this user-filled form.

The server receives the form and passes it to the manager encoded as a character string. The manager gives this string to the parser which extracts from it the input graph, the type of service requested (DRAWING), the names of the input and output formats (MALF input and postscript, respectively),



(a) (b)

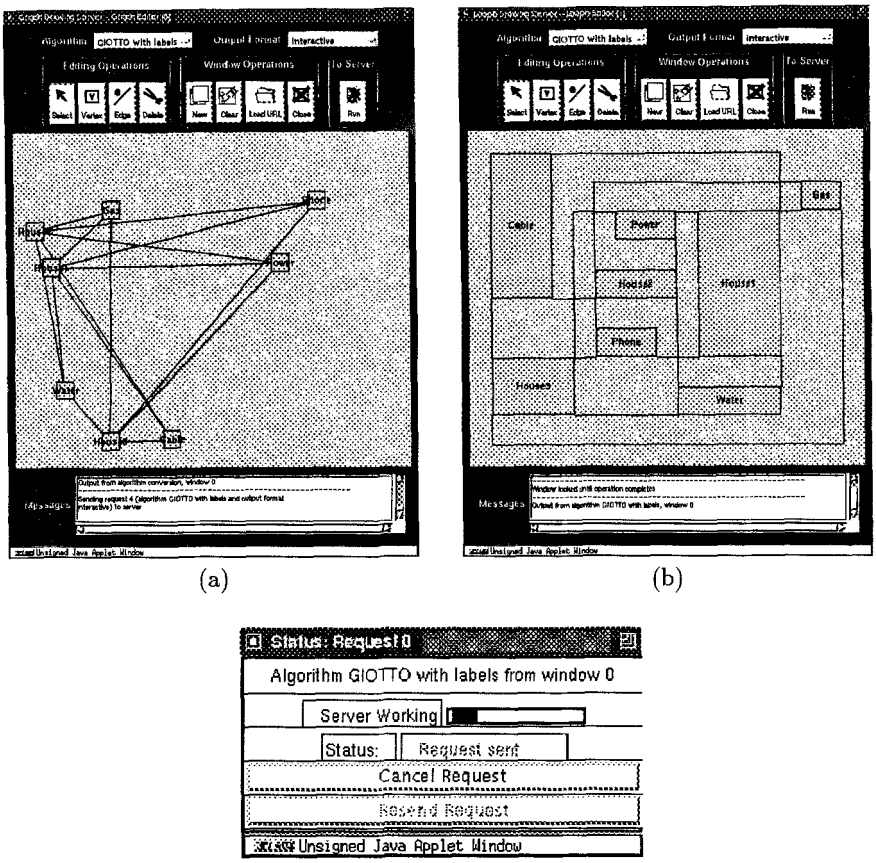
Fig. 2. *Scenario 1. Using the forms interface.* Drawing a graph described in MALF input format using GIOTTO, with output in postscript format. (a) Part of the user-filled form; (b) Drawing in postscript format returned to the user and displayed by the browser with **ghostview**.

and the drawing algorithm to use (GIOTTO). The manager then consults the graph drawer to determine the graph format required by the drawing algorithm (GIOTTO uses GDS) and calls the translator to convert the graph from MALF input to GDS. The manager sends the graph description in GDS format to the graph drawer, which uses GIOTTO to construct a drawing of the graph. The drawing is returned to the manager in GDS, the output format of GIOTTO, so the manager must once again invoke the translator to convert the drawing to postscript. The final drawing is saved to a file on the server and the URL of this file is returned to the client.

Back on the client, the user's browser receives the URL sent by the server and automatically loads the file, displaying it on the screen as shown in Figure 2(b) (assuming the browser is configured to automatically display postscript files; otherwise the user would be prompted to download the file).

Scenario 2. Using the Graph Editor (see Figure 3).

As before, the user loads the service's homepage into her browser. This time she first clicks on the *interactive applet interface* and then on the *start client* link. This causes a graph editor window to appear. Figure 3(a) shows a graph editor window with a sample graph constructed interactively by the user. To obtain a drawing of the graph the user must specify the algorithm and the output format



(a) (b)

(c)

Fig. 3. *Scenario 2. Using the graph-editor.* Drawing a graph with GIOTTO-with-labels, where the drawing is to be displayed in an editor window. (a) The interface of the editor and the input graph; (b) Drawing returned by the server; (c) Status window displayed while the request is being handled by the server.

by making the appropriate selections on the pulldown menus at the top of the window. The example shows the algorithm GIOTTO-with-labels and the output format "interactive". (Selecting "interactive" as the output format will cause the drawing to be displayed in another editor window; all of the other formats supported by the forms interface, such as parenthetic notation and GIF, are also supported by this interface and will be displayed in a new browser window.) Once the algorithm and output format have been selected, the user clicks on the "Run" button to send the request.

Sending the request causes a small status window to pop up, providing the user with information about the processing of the request and allowing her to cancel a running request (Figure 3(c)). The processing of the request on the server side follows the same basic pattern as for the forms interface (parsing, translation, drawing, translation); the differences are a slightly different front-

end for reading and parsing the request and some extra machinery to ensure that responses destined for the same client are sent back in the proper order.

When the request completes, a new graph editor window appears (since the chosen output format was interactive) with the resulting drawing (Figure 3(b)), scaled to fit in the window. This drawing is fully editable, and can be modified and resubmitted to the server. It can also be saved, by sending a translation request to the server to convert the graph into some other format (presumably a text format), which is displayed in a browser window and can then be saved using the browser's own "Save As" mechanism.

4 Future Work

There are several directions for further work on the service:

- Adding support for new formats such as GML [3].
- Adding new algorithms. We plan to start with implementations already available freely on the Web.
- Adding a programmer's interface, where a small Java or C++ library is provided to handle communications with the server, allowing a user to write a program with a call to the server as a subroutine.
- Adding support for incremental graph drawing.
- Providing a service (accessible through the Web) through which the users can conduct experimental studies on graph drawing algorithms.

References

1. L. Buti, G. Di Battista, G. Liotta, E. Tassinari, F. Vargiu, and L. Vismara. GD-workbench: a system for prototyping and testing graph drawing algorithms. *Graph Drawing (Proc. GD '95). LNCS*, 1027:76–87, 1996. F. J. Brandenburg, Ed.
2. G. Di Battista, A. Garg, G. Liotta, R. Tamassia, E. Tassinari, and F. Vargiu. An experimental comparison of four graph drawing algorithms. *Comput. Geom. Theory Appl.*, 1996. to appear.
3. M. Himsolt. GML: Graph Modelling Language. Manuscript, Universität Passau, Innstraße 33, 94030 Passau, Germany, 1996. Available at `http://www.uni-passau.de/~himsolt/Graphlet/GML`.
4. E. Koutsofios and S. North. Drawing graphs with *dot*. Technical report, AT&T Bell Laboratories, Murray Hill, NJ., 1995. *dot* user's manual. Available at `http://www.research.att.com/dist/drawdag`.
5. S. North. 5114 directed graphs, 1995. Manuscript. Available at `ftp.research.att.com/dist/drawdag`.
6. A. Papakostas and I. G. Tollis. Improved algorithms and bounds for orthogonal drawings. *Graph Drawing (Proc. GD '94). LNCS*, 894:40–51, 1995.
7. S. Singh. Documentation for Paren-to-GDS. Manuscript, Dept. of Comp. Sci., Brown University, 1991.
8. R. Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM J. Comput.*, 16(3):421–444, 1987.
9. R. Tamassia, G. Di Battista, and C. Batini. Automatic graph drawing and readability of diagrams. *IEEE Trans. Syst. Man Cybern.*, SMC-18(1):61–79, 1988.