TECHNICAL REPORT NO. 295

GraphView: An Extensible Interactive Platform for Manipulating and Displaying Graphs

Version 1.0

by

Bjarni Birgisson
Gregory E. Shannon

December 1989

COMPUTER SCIENCE DEPARTMENT
INDIANA UNIVERSITY
Bloomington, Indiana 47405-4101

GraphView: An Extensible Interactive Platform for Manipulating and Displaying Graphs [†]

Version 1.0

©1989
Bjarni Birgisson
Gregory E. Shannon

Department of Computer Science
Indiana University
Bloomington, Indiana 47405

Technical Report #295

December 1989

0. Abstract

GraphView is an extensible graphical tool for interactively creating, editing, manipulating, and displaying, graphs on the NeXT computer. The key features of GraphView are its easy-to-use interface and its easy-to-extend collection of graph manipulation tools. This technical report discusses GraphView's motivations, design goals, and implementation details.

[†] This research has been supported in part by a Hewlett-Packard Faculty Development Fellowship, NSF grant CCR–89–09535, an Indiana University Summer Faculty Fellowship, the Office of Research and Graduate Development at Indiana University, and NSF grant DCR-85-21497.

1. Introduction

In the process of working with graphs or designing graph algorithms, there often is a need for:

- displaying multiple graphs

- testing algorithms on different graphs

- analyzing the interaction between a graph's structure and various algorithmic transformations.

Until recently, pap-r and pencil were the primary tools used to address the above needs. Computers were used occasionally, but only when the benefits of this out weighted the typically large overhead for programming solutions. Such benefits have occurred for special application areas of graphs (e.g. PERT charts and interprocedural dependency graphs). However, the vast majority of research (and teaching) with and on graphs is still carried out with paper and pencil. GraphView is an attempt to fulfill the above needs with a general and easy-to-use software tool. With GraphView, we are trying to answer the question:

"Can we build an interactive and extensible software tool for displaying and manipulating graphs?"

This paper reports on our affirmative answer to this question. In particular, GraphView is an extensible graphical tool for interactively creating, editing, manipulating, and displaying graphs on the NeXT computer. GraphView's key features are its easy-to-use graphical interface and its easy-to-extend collection of graph manipulation tools. Graphs can be created directly on the screen, read in from a file, produced by a graph-family generator, or extracted from a database of graphs. Graphs and various characteristics can then be displayed, edited, and transformed using a dynamic external library of transformations written in Objective-C or any other description language. Any graph generated can be printed or stored in a file. (For more details, see the documentation for GraphView [BiSh 89].)

Previous work related to GraphView has involved either producing graph tools for widespread use or investigating issues in visualization and animation. Projects of the first type of include ISI Grapher [Robi 87], MacProject (for PERT charts) [Mac 84], and GDBX [Bask 85]. Though these did achieve some degree of widespread use, they are focused on special subclasses of graphs and problems; they lack full generality. Projects of the second type include BALSA I and II by Brown and Sedgewick [Brow 84, BrSe 84, BrSe 85] and TANGO by Stasko [Stas 89]. These systems focused on the issues involved in visualizing and animating discrete objects such as graphs. The design concepts from these systems are fundamental for the area of visualization, though, as research systems, they were not intended to serve as platforms for application systems.

In addition to the systems discussed above, there have been many other application-specific and locally targeted systems for manipulating and displaying graphs. Most of these recent systems are interactive. For a summary of these systems, see the excellent survey by Tamassia and Eades [TaEa 89]. The system GMB by Jablonowski and Guarna [JaGu 89] is the most similar to GraphView.

None of the systems discussed so far are easy to extend. In comparison, GraphView is the first one to be extendable without having to recompile or restart the whole system. In the future, we plan to implement a high-level language facility (based on Scheme/Lisp) inside GraphView so that new transformations can be added to GraphView from inside the program. Currently, new transformations must be written and compiled as independent tasks. As discussed in Section 5, we expect GraphView will become highly portable by the end of 1990. GraphView is not intended to be a research system in itself; we are working towards making it a widely-used general system for graphs.

Given our success in meeting our design goals and the generality of GraphView, we expect it to serve as a platform for significant applications. In particular, work is already underway to use GraphView for:

- graphical descriptions of database queries
- dependence graphs in program transformers for parallel programs
- graph layout problems
- parallel algorithms for low-genus graphs
- visualizing graph-oriented data structures in a visual debugger
- algorithms courses on basic computer algorithms and data structures
- artificial intelligence courses involving semantic and neural networks.

In the next section we give an expanded description of the system from the user's point of view. In Section 3 we discuss GraphView's design goals. In Section 4 we discuss the implementation details. In Section 5 we conclude with comments on current and future work for GraphView.

2. The GraphView System

In this section we summarize the GraphView system. Complete user's documentation for GraphView is in the companion technical report *GraphView Documentation* [BiSh 89]. It includes a tutorial, user's manual, and important implementation specifications.

GraphView is an extensible graphical tool for interactively creating, editing, manipulating, and displaying, graphs on the NeXT computer. The key features of GraphView are its easy-to-use graphical interface and its easy-to-extend collection of graph manipulation tools. Graphs can be created directly on the screen, read in from a file, produced by a graph-family generator, or extracted from a database of graphs. Graphs and various characteristics can then be displayed, edited, and transformed using a dynamic external library of transformations written in Objective-C or any other description language. Any graph generated can be printed or stored in a file.

GraphView enables the user to work on multiple graphs at once, each appearing in a separate window on the screen. Several display attributes, such as the display of labels or characteristics, are controlled by the user. Subgraphs are copied between different graphs or within the same graph using the standard Copy/Cut/Paste mechanism.

In GraphView, graphs can be arbitrarily manipulated using external programs which communicate with the main program. These transformation programs can be written in any suitable programming language capable of communicating with the main application. In this version, plain ASCII files in a special format are used to transfer information between the programs. Classes of external programs include:

1. Regular transformations which operate on a given input graph, returning one or more graphs as output.

2. Generators, which create new graphs.

3. Displayers, which attempt to lay out a given graph geometrically. <<to be implemented>>

GraphView automatically keeps track of the history of each graph, i.e. which external programs and graphs were used to create it. Some other features available include intelligent printing, dynamic display of characteristics, automatic version numbers, a preference panel for setting program defaults, automatic labeling of graphs, and scalable views.

3. Design Goals

The primary objective in GraphView's design was to provide an interactive tool that would be flexible and natural to use both for research and education purposes. It should make the user feel free to do things in his or her own way as much as possible by trying to avoid imposing constraints on implementing the user's ideas. GraphView is meant to extend the functionality of paper and pencil allowing the user to draw and edit graphs in a natural way, eliminating the need for an eraser and the tedious process of redrawing everything when the graph changes. In addition GraphView is to be able to apply algorithms designed and implemented by the user, to transform existing graphs or generate new ones, and display the results. Therefore, GraphView must be:

- *Easy to lean and use*. We expect GraphView to be used by researchers, teachers, and students from a broad range of backgrounds and levels of expertise with computers. GraphView must be easy to use for both novice and experienced users.

- *General and complete.* Any type of simple graph and graph algorithm should be natural to work with. Any type of transformation on a graph should be possible.

- *Efficient and reliable.*

- *Extensible.* With a high-level programming language, users should be able to naturally add new transformations, generators, and displayers. to the standard library.

- *Interactive.*

- *Graphically oriented.*

- *Straightforward to implement, maintain, and extend.*

4. Implementation Details

In this section we discuss GraphView's important implementation details. In particular, we show how its user interface was made easy to use and how extensibility was achieved. For the more detailed documentation necessary to write transformations, to modify, or to extend GraphView, the complete class specifications and file format specifications for GraphView are in the companion technical report *GraphView Documentation* [BiSh 89]. It also includes a tutorial and user's manual.

Given the design goals above, we produced the following implementation goals.

1. GraphView should be an interactive graphical graph editor, capable of manipulating graphs in a natural way, saving/reading to/from disk, printing, and providing dynamic control of display attributes.

2. It must have some means of communicating with external programs or transformations written by the user and applying them to the graphs created by the editor. This might involve showing partial results after various steps in the user's transformation.

3. All of the above should be integrated into a complete graphical application program managing several graphs at once and provide an easy–to–use interface.

In the following subsections we describe the development environment, translation of graphs to objects, higher level objects, and transformations. It is assumed that the reader has some knowledge of object oriented programming concepts such as inheritance, methods or messages, classes and instances, sub- and superclasses, and instance variables.

4.1 The development environment

The NeXT computer was chosen as the computing platform for several reasons. First, it provides an excellent environment for developing applications requiring a graphical user interface. It also provides good support for 2-D graphics, runs a Unix-based operating system, and was primarily targeted (at the time the project was started) at universities and research environments, making it a feasible choice for a project like this. The main drawbacks of using the NeXT computer are the lack of color and the fact that programs developed on the NeXT are not easily ported to other machines (yet).

GraphView is written in Objective C, using Display PostScript as an image specification language. At the time the project started there was no other choice of a programming language if one wanted to use the full features of the graphical interface. Since then, Allegro Common Lisp has become available and might even be a better choice for reasons outlined below.

The NeXT/Objective C environment is almost entirely based on the object oriented programming concept. Graphs can be represented nicely as a hierarchy of related objects simplifying the tedious manipulation of the data structures traditionally used for representing graphs. The user also benefits from the encapsulation of the graph components as objects, since when writing transformations access is provided to individual components through predefined high level methods so that minimal knowledge of the underlying data structures is needed. This also helps maintain the integrity of the internal data structures since the user is not allowed direct access to them.

4.2 The implementation of graphs

For the purposes of GraphView, graphs are considered to be built from two types of objects: vertices and edges. Although these two object types are usually considered to be quite different, they share some common properties in the program. For example, both types of objects can be defined by labels, selected, and accompanied by attached information. This last point deserves a few comments. Frequently when working on graphs, one wants to assign some characteristics to edges and/or vertices in the graph. Examples include colors of vertices and weights of edges. At the lowest level this is simply implemented in such a way that a character string of any length can be assigned to each object in the graph. It is then left to the upper level objects to interpret this information depending on the situation. Given this, the classes `Edge` and `Vertex` were made subclasses of a common parent class, `GraphObject`, providing the basic functionality common to both object types.

In addition to being a `GraphObject`, a `Vertex` contains a list of `Edge` objects and its geometric location in the graph. For `Edge` objects however, we only need to add instance variables representing the source and destination vertices.

The `Graph` class itself is finally defined as a list of vertices along with some other necessary variables. Following is a short summary of the functionality of each class (for details see the class specifications).

GraphObject:

- Manages the state of the selection for each object.
- Manages the assignment and retrieval of labels and characteristics attached to objects.
- Manages temporary variables, used for example for numbering objects when saving/reading graphs to/from disk.

Vertex:

- A list of edges and the geometric location of the vertex.
- Provides methods for drawing the vertex.
- Provides methods for writing the information associated with the vertex to disk.
- Capable of detecting whether it is hit, given the position of the mouse.
- Provides various methods for moving the vertex, computing a bounding rectangle, and sorting the edge list according to embedding.

Edge:

- Consists of a source and destination vertex (and a pointer to the containing graph).
- Provides methods for drawing the edge.
- Provides methods for writing the information associated with the edge to disk.
- Capable of detecting whether it is hit, given the position of the mouse.
- Provides methods for reversing, computing its angle with respect to either vertex (for sorting edges according to embedding) and computing a bounding rectangle.

Graph:

- Has a list of vertices, a list of selected vertices, and a list of selected edges.
- Provides methods for saving/reading the whole graph to/from disk. The graphs are saved as text files in a special format.
- Provides methods for reading and writing to and from the pasteboard. This is done in response to messages from higher level objects as a part of a cut, copy or paste operation. Graphs written to the pasteboard are in a form similar to graph files on disk, but somewhat simpler.
- Provides methods for adding and removing vertices and edges.
- Provides various methods for operating on the current selection.
- Stores characteristics attached to the graph. This could be some descriptive text or some other facts about the graph. This is implemented by having an instance of a `GraphObject` as an instance variable and using it to store the characteristics.
- Stores information on which characteristics to display for the objects in the graph (see discussion below). This information is stored as two strings of characters and again we use `GraphObjects` to access it.

It is at the `Graph` level that meaning is assigned to the characteristics attached to individual graph objects. The program can currently interpret characteristics of the form <keyword> <some_value>. An example would be: `"color blue"`. The `Graph` class stores for each graph, two lists of keywords whose values are to be displayed next to the objects to which they are attached. One list is used for vertices and another for edges. The user can easily change the characteristics to be displayed by simply specifying the proper keywords. Note that although the program interprets characteristics of this particular form, the interpretation can be turned off and the characteristics used to represent whatever the user chooses.

Figure 1 shows the relationship between the basic graph classes.

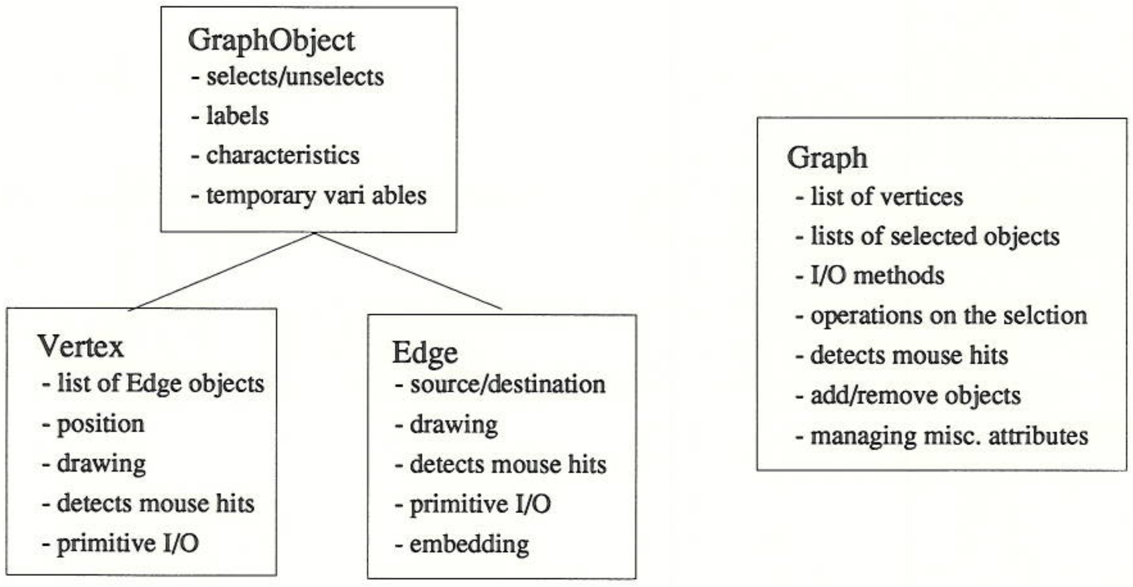

Figure 1. The basic graph classes.

4.3 The higher levels

Connecting the graph data structures described above to the user interface provided by the NeXT environment is relatively straightforward. The NeXT application kit supplies a `View` class which already does most of the lower lever tasks associated with displaying the graphs in a window and tracking the mouse. The graph editor is simply implemented as a subclass of the built-in `View` class. This new class, called `GraphView`, has a `Graph` as one of its instance variables. Its purpose is to respond to messages from the interface objects and apply them to the underlying graph. All operations performed on the graph in the GraphView application have to go through the `GraphView` class; the levels above do not have direct access to the graph itself. The main tasks performed by this class are:

GraphView:

- Responds to mouse clicks, for adding edges and vertices and selecting objects.
- Draws the graph when necessary (scrolling, resizing windows).
- Prints.
- Responds to Cut/Paste/Copy messages from the user.
- Responds to various other messages initiated by the user through the interface objects, affecting display attributes such as scale and display of additional information.

A few words on how the graph is drawn: at first the graph data structure was simply traversed (visiting each object once) each time the graph needed updating, sending each vertex and edge a message to draw itself. This is adequate for small graphs (less than 30-40 vertices) but becomes intolerably slow for very large graphs. As a first attempt to optimize the drawing procedure, each object in the graph was modified to know its bounding rectangle and the object was only drawn if the rectangle intersected the rectangle being drawn. By specifying rectangles as small as possible for updating and intersecting that rectangle with the visible rectangle, it was possible to reduce considerably the PostScript code needed for redrawing. We still needed to traverse the graph to determine which objects should be drawn. However, we discovered that the operations requiring the fastest response time (e.g. scrolling) usually don't need to update the graph itself at all. Therefore an off-screen cache window could be used to draw the whole graph and update to the graph could be done to that window. The actual display of the graph then simply requires copying the appropriate rectangle from the off-screen cache to the screen. This is several orders of magnitude faster than traversing the graph as before.

Finally an application class was created to integrate all of the above into a complete program. It manages any number of graphs on the screen at once and handles tasks such as passing messages from the user interface objects to the currently active graph, setting user preferences, applying user written transformations to the graphs, and managing auxiliary support windows[1]. The application also keeps track of the history of each graph, i.e. what transformations were applied to create it. This is done by attaching a history file to each graph file that lists its ancestors and the appropriate transformations.

4.4 Applying transformations

The transformation[2] programs are meant to be totally independent from the main application. The user should be able to write a transformation and, without recompiling or restarting GraphView, that transformation should become visible and available for use immediately. This implies that the main application must have some means of executing the transformation, passing it the graph in question and reading back any intermediate and final results. Furthermore it is desirable to have a separate communication channel along which the transformation program can pass auxiliary feedback or progress information back to the user. There are several ways to do this.

---

1 Actually two other classes are also used. One is a subclass of the built-in `ScrollView` class that adds scrolling capabilities to a `GraphView` class, and the other is a class that reads in an interface file, for an instance of a `GraphView`, and initializes its parameters.

2 By transformations we also mean generators, which are actually a special case of transformations operating on an empty graph, and displayers, which will compute the geometric locations of vertices.

The method that GraphView currently implements, is to simply write the graph to be transformed to a file, then create a process which executes the transformation program and wait for that process to terminate. The executed program starts out by reading in the graph file (its name is supplied as a parameter) then it applies the transformation and writes an output file which the application can then read in and display. Intermediate results are handled the same way: by writing a file and passing a message along the communication channel (a Unix pipe) to display the file. Figure 2 illustrates this arrangement. Currently the transformation programs are written in Objective C using the same classes as described above to store and access the graphs and compiled as stand-alone programs. The user is not restricted to using only Objective C. Any programming language capable of reading and writing files (and of course manipulating the graphs) can be used.

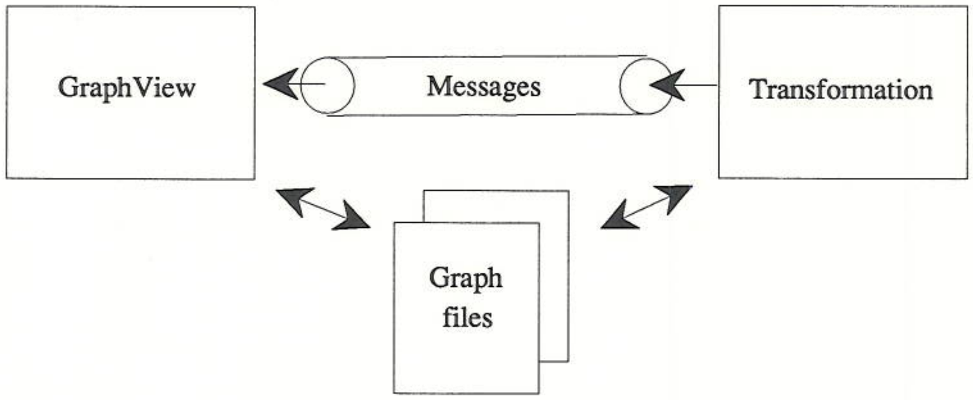

Figure 2. Applying a transformation.

The advantage of this method is its simplicity. The transformations are completely independent programs and can easily be run as such, given the proper parameters. They could even be run on a remote (possibly non-NeXT) machine for transformations requiring large amounts of storage or CPU time.

The disadvantages of using files for the graph transfers are obvious: it is inherently slow for large graphs, having to do at least four read/write operations on the entire graph for one transformation.

Another way of communicating with the transformation programs is to use shared memory between the two processes. The easiest way to do this is probably to load the transformation program into the address space of the application and execute it there. However, this requires relocating the transformation program before loading, since there is no way of knowing where it will be loaded at compile- or link-time.

Sharing memory between processes is also possible by other means on the NeXT computer. The Mach operating system allows a process to specify that another process is allowed to inherit a part of its virtual memory image. This is not straightforward to implement using the object representation mentioned above for the graphs. All objects are dynamically allocated memory at run time and there is no easy way for the application process to determine what part of its virtual image is allocated to a particular graph. We tried using this scheme by first writing a representation of the graph to a ''file'' in memory and allowing the transformation process to inherit the memory space occupied by the file, using Objective C classes (Speaker/Listener) supplied for message-passing between processes. This still requires both processes to perform the read/write operations for the graphs, although from a new memory file. The results showed that this was not worth the extra complexity; running times of transformations were almost the same as when using regular files for the transfer, indicating that increased paging due to the memory files probably results in similar disk activity as just using files explicitly.

The third way of implementing transformations would be to integrate an interpreter for a specialized graph manipulation language into the application. The user could then type in a program or load it from a file and have it executed under the control of the main application. One could imagine providing constructs such as:

```
for all vertices v with v.color == blue do ..... od
```

Just recently a Common Lisp environment that provides a full interface to the Application Kit's windowing environment has become available for the NeXT computer. This suggests that a conversion to a Lisp implementation would make the above problems disappear trivially while retaining all the benefits of the NeXT graphical user interface. A transformation program written in Lisp (augmented with a graph package) could simply be loaded or typed into a window and applied to the current graph. Objective C could still be supplied as a language for writing transformations, since Lisp also supports loading of foreign language functions. The transformation programs would probably be more straightforward to implement in Lisp than in Objective C, relieving the user from taking care of issues such as allocating an freeing memory.

5. Future Directions

GraphView's current limitations include no capacity for color (because it works only on the NeXT) and portability to other systems such as Suns, Macs, or even high-end graphics workstations. We are working in conjunction with the Visualization Lab at Indiana University's Center for Innovative Computer Applications (CICA) to write GraphView in terms graphics/interface description toolkit which they have ported to four computing platforms already, with more planned for the spring of 1990. This toolkit from CICA will make GraphView very portable.

In 1990, there are three major goals for GraphView. First is to get user feedback and feedback from applications where GraphView is a platform. To this end, GraphView will be integrated into the "Analysis of Algorithms" course in the Computer Science Dept. at Indiana University. Also, GraphView is will be used in a number of applications listed at the end of the Introduction Section. Another goal is to develop significant libraries of algorithms for generating, transforming, and displaying graphs. A final goal is to enable interactive high-level programming in GraphView so that new transformations can be built on-line. Scheme, a Lisp-related language, will be used for this.

References

[Bask 85] D. Baskerville, *Graphic Presentation of Data Structures in the DBX Debugger*," Technical Report UCB/CSD 86/260, Computer Science Division, University of California at Berkeley, October 1985.

[BiSh 89] B. Birgisson and G. Shannon, *GraphView Documentation,* technical report #299, Computer Science Department, Indiana University, December 1988.

[Brow 87] M. Brown, *Algorithm Animation*, ACM Distinguished Dissertations, MIT Press, 1987.

[BrSe 84] M. Brown and R. Sedgewick, "A System for Algorithm Animation," *Computer Graphics,* 18(3):177–186, July 1984.

[BrSe 85] M. Brown and R. Sedgewick, "Techniques for Algorithm Animation," *IEEE Software*, 2(1):28–39, January 1985..

[JaGu 89] D. Jablonowski and V. Guarna, "GMB: A tool for manipulating and animating graph data structures," *Software–Practice and Experience*, 19(3):283–301, March 1989.

[Mac 84] *MacProject,* Apple Computer, Inc., Cupertino, CA, 1984.

[Robi 87] G. Robins, "The ISI grapher: a portable tool for displaying graphs pictorially," *Proceedings of Symboliikka,* 17–18, August, 1987.

[Stas 89] J. Stasko, *TANGO: A Framework and System for Algorithm Animation*, Ph.D. Thesis, Brown University, May 1989.

[TaEa 89] R. Tamassia and P. Eades, *Algorithms for Drawing Graphs: an Annotated Bibliography*, Technical Report CS-89-09, Dept. of Computer Science, Brown University, Feb. 1989.