An Object-Oriented Design for Graph Visualization

M.S. Marshall, I. Herman, G. Melancon

Information Systems (INS)

INS-R0001 January 31, 2000

Report INS-R0001
ISSN 1386-3681

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

An Object-Oriented Design for Graph Visualization

M.S. Marshall, I. Herman, G. Melançon

CWI

P.O. Box 94079, 1090 GB Amsterdam

Email: {M.S.Marshall, I.Herman, G.Melancon}@cwi.nl

ABSTRACT

Many applications, from everyday file system browsers to visual programming tools, require the display of network and graph structures. The Graph Visualization Framework[1] (GVF) is an architecture that supports the tasks common to most graph browsers and editors. This article gives a brief overview of the design of the GVF and focuses on the core classes that are used to represent and manipulate graphs. The design of the core classes is justified by the requirements for navigation and visualization.

1998 Computing Reviews Classification System: D.2.2, D.2.11, H.5.2, I.3.6, I.3.8

Keywords: Graph Visualization, Networks, Data Structure, Framework, Software Libraries.

Note: This paper has been submitted as a journal publication. At CWI, the work was carried out under the project INS3.2 “Information Visualization”.

1. INTRODUCTION

Graph visualization refers to the display of a set of data that has relations between elements and to the interactive facilities that make navigation through the visual representation of that data possible. Graph visualization is an increasingly important area of research, both because of its everyday applications and because of the need in many fields to view large and complex networks. A few areas that have tasks involving graph visualization are web administration, network administration, computer science, biology, chemistry, medicine, and financial analysis. Although many different graph visualization systems have been built for both research and specialized applications, none of these systems has been able to meet all the requirements necessary for the exploration of large graphs.

When faced with the task of analyzing a large set of data with relations we are immediately faced with some of the challenges in graph visualization. For instance, we must choose a type of layout before we can draw the graph. If the graph is too large to fit on the screen, we must choose an abstract view of the graph that exposes certain types of information about the graph, yet reduces the amount of information rendered. Both choices influence what we can discover about the graph, because they determine which information is presented and how it is presented. However, our needs don’t stop here; we would also like to interact with the graph, changing the view in order to gain insight into the data. All these features require a system that can easily adapt to our needs and quickly change the way a graph is presented. Obviously, graph visualization has a set of special requirements, all of which are necessary to support the full exploration of a graph.

In order to render a large graph, it is sometimes necessary to reduce the data set in some way. Simple filtering techniques based on attributes don’t always suffice, so more sophisticated techniques such as clustering (i.e. classification) are often used. The process of clustering involves discovering groups in the data. A fundamental technique in graph visualization displays the groups, or clusters, of a graph using a special type of node called a *meta-node* to represent clusters or subgraphs in the graph. This technique makes it possible to represent a large graph by displaying fewer elements, allowing the user to control the level of detail by “opening” and “closing” meta-nodes. Such an approach requires a way to store and manipulate graphs whose nodes may represent subgraphs.

[1] Available at http://www.cwi.nl/InfoVisu/GVF

Users wishing to explore a graph must be able to manipulate it, in addition to navigation. A user should be able to interactively apply criteria that result in different sets of clusters within the data. In fact, the user should also be able to apply clustering proccesses to subgraphs or clusters resulting from previous activities — a process called *hierarchical clustering*. The process of clustering creates additional structural information about the graph that the user will want to compare with other processes. These types of activities demand a system that can store multiple levels of complex information about the graph, such as the auxiliary structures resulting from clustering, yet keep the original structural information about the graph intact. We will call a graph with such cluster information a *multigraph*[2], or a graph where a node may be shared among multiple graphs. In such a clustered graph, each node belongs to the original graph and may also belong to certain subgraphs that result from clustering. In the case of hierarchical clustering, the node may belong to a series of nested subgraphs. If information about several clustering operations is stored, a node may belong to subgraphs or clusters, each resulting from a different clustering proccess. As we shall see in Section 2.1, multigraphs require a different kind of design than simple graphs.

Other tasks besides clustering make support for constantly changing graphs necessary. For example, the ability to edit a graph can be an important part of a graph visualization system. A user experimenting with layout may want to add or delete parts of the graph to see the effect it has on a particular layout, or simply edit the properties of a given element to see the resulting effect. Systems that update the graph with real-time information also require a way to handle constantly changing structures. All of these tasks make support for dynamic graphs an important requirement for graph visualization systems.

In 1999, our research group at CWI started looking for an environment in which it was possible to experiment with various graph visualization techniques. The experimentation would include the interactive definition of nested clusters and the use of visual elements to represent graphs and their properties[3]. An essential feature was interaction, in which the user could select and manipulate parts of the graph and use user-defined properties to influence how the graph was displayed. Another feature that we looked for was the ability to easily extend the existing functionality, such as extending the available layout algorithms or even replacing the graphics renderer. Although we have looked at other class libraries for graphs[3, 4] in addition to many complete systems[5], none fulfilled our needs and so we decided to develop our own. The ultimate goal was to create a system that is general enough that it can be embedded in information visualization applications. Because our goal was also portability and ease of maintenance, we decided to implement it in Java. It is impossible to design a system that has everything needed for all applications, so flexibility and ease of extension were the guiding heuristics.

One of the first tasks in developing such a system is to define a suitable set of data structures. In the case of Java, this means defining a class hierarchy that enables the representation and manipulation of graphs. The challenges described so far hopefully make it clear that the specification of such a class library is not a simple matter. This paper explains the rationale of the design, describing how the core classes are used within the Graph Visualization Framework (GVF). Although the development was done in Java, we believe that the solutions we have found are of a general interest for any object-oriented environment.

2. THE FRAMEWORK: AN OVERVIEW

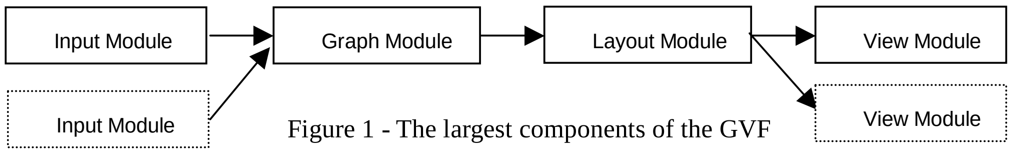

Figure 1 - The largest components of the GVF

[2] We do not use the term as it is used in graph theory.

[3] Some earlier reports give a good illustration of the kinds of developments that we are interested in [1, 2].

Although the goal of this paper is to describe the core graph class library rather than the entire GVF, it is necessary to put these classes into the right context to understand some of the requirements that directed our choices. The graph class libraries are used in the context of the GVF. The largest software components in the GVF are called *modules*. Each module fulfills a specific function and has its own thread of control that cooperates with other threads through a JavaBeans-style notification mechanism[6]. These modules correspond roughly to the functional modules described in AVS[7]. The modules can be assembled into a graph visualization application in which each module corresponds to one stage of processing in a pipeline of data flow as in Figure 1. They communicate with one another by sending messages through logical pipelines. It is possible, of course, to use the modules independently. For example, the Input Module, Graph Module, and the Layout Module have been used to perform layout measurements on sets of randomly generated graphs, in a process that didn't involve rendering (rendering takes place in the View module). Furthermore, it is reasonably simple to add new functionality such as a new layout algorithm, parser, or even an entire new View Module (e.g., if the underlying graphics engine is changed). In this paper, we will focus mostly on what takes place in the Graph Module and how it relates to the tasks in the other modules.

In the GVF, most objects can have properties and several mechanisms using properties have been put into place. Properties allow the association of a *value* with a *key* (in what is sometimes called a "dictionary"). Although properties are not unique to the GVF and numerous other systems use them, a description may lend valuable insight into the application of these mechanisms and aid in understanding how the core classes fit into the GVF.

One of the challenges faced by graph system designers is the graph classification problem. How can we design an object hierarchy for the many types of possible graphs? Unfortunately, graph types do not fit neatly into a hierarchy; they are classified according to the various combinations of properties that they have. Multiple inheritance solves part of the problem but doesn't prevent the proliferation of classes. For example, a graph with the property "directed" and "acyclic" would be classified as a DirectedAcyclicGraph. If the same graph is also "connected", then it becomes a ConnectedDirectedAcyclicGraph. Creating objects for each combination of properties in this manner causes the number of classes to increase exponentially with the number of properties. Even if it were possible to design, such a class type hierarchy would be impractical for dynamic graphs: a single edit of an edge may change the graph's type (e.g. from acyclic to non-acyclic). Such a change would necessitate a dynamic change of the class type of a graph instance, which is impossible in Java or C++. Some researchers have suggested delegation-based systems to overcome these types of limitations[8, 9]. However, our solution to this problem is to use properties to categorize graphs. Properties add a level of dynamism to objects that isn't otherwise possible in Java or C++. Adding a new property to a class instance could be considered functionally equivalent to adding a new field to the class specification. In the case of graphs, the use of properties makes it unnecessary to design a special type hierarchy for the many different types of possible graphs. Rather than instantiate a new type of graph object and transfer information when a graph property changes, we can simply change the value of the affected properties.

Because some properties depend on others, it is sometimes necessary to update several properties when a particular property's value changes. To aid in such interactions, we applied the Observer Pattern[10]: class instances can express their "interest" in the change of a specific property by registering themselves. If the property changes, the registered observers (called listeners in Java) are notified and can act appropriately. The property change listener facilitates the update of property dependency chains. For instance, if the property "acyclic" changes, a registered listener should check to see if the property "tree" now applies.[4] .

Another use of properties within the GVF is commonly found in many applications that have user preferences. Application behaviors such as how a node is rendered or which type of layout to use can be influenced by the property value that is stored at various levels throughout the application. A simple example is the color of a node. The preferred color for a node could be set through the property at the `Node` object, in the containing `Graph` object, in the containing View, or in the application itself.

[4] Trees require the "acyclic" property to be "true".

The most important graph properties used by the GVF are boolean properties, like “directed” and “acyclic”. These properties can be used in a simplified version of a constraint-based negotiation mechanism[11] to implement dynamic menus. For example, some layout algorithms will only work with particular types of graphs, i.e. graphs with certain properties. The negotiation mechanism allows us to decide at runtime which layout algorithms are made available on the menu for a particular graph view, depending on the graph’s properties. The same mechanism aids in deciding which layout to use when a graph is initially viewed. Such a mechanism makes it possible to add a layout algorithm to the application by defining a single class. In another application of properties, we can use the Factory Pattern[10] together with the negotiation mechanism to generate objects, such as specific types of graphs. The negotiation mechanism ensures that the objects are produced with the requested properties but according to user preferences and within system constraints. In this way, it is possible to precisely determine the properties of the objects that such a factory produces at runtime.

2.1. The graph as a node

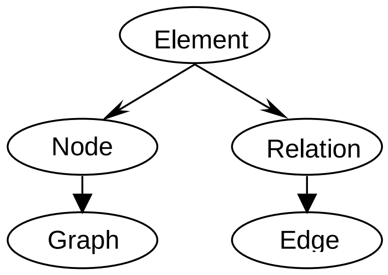

Figure 2

Figure 2 shows a simplified class diagram of some essential classes in the GVF. The `Element` object is the most fundamental object of the core classes. It is meant to capture the features common to the two types of visual elements in a graph: nodes and edges. An `Element` represents data before any relational information has been added to it and can therefore represent a generic data element in a data visualization application. All `Element` objects can have properties assigned to them. For the purposes of layout, coordinates can be assigned to an `Element`. An `Element` can also have data (i.e. the data that the graph is representing) associated with it, along with any information necessary to view it. For instance, a node (that is also an `Element`) may represent a web page that can be rendered by a specified browser. Elements can be grouped together, as in a graph or a selection, and we can ask an `Element` about the groups to which it belongs. All of these features are available to nodes, edges, and graphs through inheritance.

Some features of the hierarchy in Fig. 2 might be surprising at first glance. For example, a designer might be initially inclined to define a `Graph` object as a direct translation of the definition of a graph: a collection of nodes and edges. A `Graph` object, in this case, would store a list of all the nodes and edges that belong to it. Similarly, one might be tempted to define nodes in a graph as a direct container of references to its neighbors, in the form of object references directly represented as fields in the `Node` class. However, this simple approach does not work in view of the requirements that we discussed in the introduction. The use of multigraphs and their dynamic nature requires a somewhat more sophisticated approach that will be outlined in what follows.

In dynamic graphs, where clusters are constantly created and disbanded, and where edits or updates may change a graph, flexibility is of the utmost importance. Not only support for dynamic graphs but also support for multigraphs is necessary. Multigraph support is especially important for clustering. During clustering, we often introduce new relations into the data, both between elements of the original graph and between the clusters themselves. We would like to build clusters and store the induced relations, yet retain information about the graph that contains them.

Another place where multigraphs are needed is during layout. Often, layout algorithms extract a subgraph (for example, by temporarily disregarding certain edges) and perform layout on the extracted elements, replacing the missing elements in the view. In this case, it is advantageous to regard the elements as belonging to both the original graph and the extracted graph, i.e. multiple graphs.

In all these examples, there is one thing in common: whereas the set of nodes and edges may be constant, a graph could be considered as a special “view” of these entities, and a system might need several such “views” concurrently on the same set of entities.

In the GVF, we store the structural information about a graph in the `Node` object. In other words, the `Node` object can be queried about the edges that it has *in a particular graph context.* This makes it possible for a `Node` to belong to multiple graphs while retaining a unique identity. For example, the following lines could be used to find out what edges `NodeA` has in `Graph1` and then `Graph2`:

```
edgeSet1 = NodeA.getEdges(Graph1);

edgeSet2 = NodeA.getEdges(Graph2);
```

Notice that the `Node` object is queried with the graph context as a parameter. More importantly, the node can change graph membership by simply changing what is stored in the `Node` object; there is no need to find and edit an entry for the `Node` in lists maintained in the containing graphs. The node-centric model of the GVF makes this possible.

Another motivation of our project besides the ability to handle multigraphs is the need to treat a `Graph` object as a node when it is being used to represent a subgraph or cluster. Because many of our `Graph` objects will be treated as meta-nodes, the `Graph` object is defined as a subclass of the `Node` object. This definition makes several things possible. In this way, a `Graph` can be handled as a `Node` when it represents a subgraph, which simplifies the implementation of nested graphs. Furthermore, a `Graph` can have a relation with another `Graph` or `Node`, making it possible to express one-to-many relations and relations between clusters.

```
cluster1 = new Graph();

cluster1.add(Node1);

bigGraph.add((Node)cluster1); // cluster1 is treated as a node in bigGraph

Edge edge = new Edge((Node)cluster1, (Node)cluster2);

bigGraph.addEdge(edge);
```

Several features of `Element` can also be used by the `Graph` object. Of course, a `Graph` object can have properties just like any other element. Additionally, a graph can have data associated with it.

The discussion until now has only used the term “edge” to refer to the relations between nodes. The word “edge” implies a binary relation. However, *hypergraphs* are also important in practice: they involve relations between *sets* of nodes that can be best represented by tuples. The `Relation` object in the GVF stores such a tuple (and degenerates in the simplest case to a binary relation or edge). The `Relation` object makes GVF potentially useful for systems that use hypergraphs such as VLSI and PCB design[12] while remaining inconspicuous to implementers concerned only with binary relations.[5]

2.2. Traversals

During the manipulation of graphs, it is often necessary to visit a set of nodes and/or edges. The order of the visitation may be important for a particular application, such as a traversal that describes a path from one node to another. In the GVF, a `Traversal` is an object that can produce a set of `Elements` in an order

[5] Hypergraph relations have not yet been exploited in our visualization environment, and are the subject of future work.

that describes the visitation of a graph. The intended method of access for a `Traversal` is through the Iterator Pattern[10], familiar to many readers through the STL library of C++ and collections in Java 1.2. The `Iterator` allows the user to iterate over a collection of objects without requiring special knowledge about the contents of the collection. Because different circumstances require different implementations of the `Traversal`, the factory pattern is applied to produce the type of traversal that best suits the situation and the resulting iterator is returned to the user.

It is important to note that `TraversalIterators` return `Element objects`. In this way, visits of both nodes and edges, and even graphs, can be included in a single traversal. This provides us with a unified and comprehensive approach to graph traversal. This is especially convenient during the layout of several connected clusters, where we would like to treat each cluster as an `Element` along a traversal:

```
Iterator clusters =
          theTraversalIteratorFactory.create("BreadthFirstSearch", graph);
while (clusters.hasNext()) {
  Node cluster = (Node)clusters.next();
  ...
}
```

In general, the `Iterator` should make the implementation details of most traversals irrelevant to the user. However, the user may occasionally want a special type of traversal that isn't available from the factory and find it necessary to write her own. An explanation of how the Factory Pattern is used for Traversals should clarify its value and help the reader understand what is involved in implementing a traversal.

The `Traversal` is implemented in several different ways. A `Traversal` can be statically defined or stored, or can be dynamically defined by the logic that implements the iterator. The choice among implementations is made by the factory and can be determined by the tradeoff between speed and storage. If the factory has previously built and stored the requested traversal, it simply returns it. However, if the user requests a traversal that hasn't been built yet, the factory may return an iterator whose next element is dynamically defined. Rather than throw away the traversal information, this iterator stores the returned elements in a traversal located at the factory while it is being used. The next time that particular traversal is requested for that particular graph, the cached version can be supplied. Of course, more sophisticated logic may be used by the factory. For example, the factory could consider several factors such as available memory, probability of a particular type of traversal being used again, and so forth. In some situations, it may be useful for the user to indicate to the factory that the traversal will only be used once and this information could be used to help decide what sort of object to return. As with most user interactions with the factory, the preference can be contained in a property.

One stored implementation of `Traversal` is called `LinkedTraversal`. The design model for the `LinkedTraversal` is very similar to the model for storing graph information at the `Node` object except that the storage for `LinkedTraversal` is in the `Element` object. In the `LinkedTraversal`, the elements can be considered part of a "switching network" in which each element points the way to the next element. In such a traversal, we may also decide to change to a second traversal when we are part way through the first one (this requires access to the actual `Traversal` object rather than only the `Iterator`). However, the implementation of `LinkedTraversal` is constrained because it cannot store a path that crosses through the same element more than once. Another stored type of `Traversal`, called `StoredTraversal`, is less dynamic but faster and more storage efficient. The `StoredTraversal` is implemented as an array of `Elements,` which makes iteration faster.

In order to build most types of traversals, we must be able to visit all the nodes of a graph. When a graph is being initially built, a special `LinkedTraversal` called a `nodeTraversal` is simultaneously built that visits all the nodes. It is easy to demonstrate that if such a traversal were implemented with an array, changes to the graph would require more work to update the `nodeTraversal`. This can become a crucial issue when dealing with large graphs.

The `LinkedTraversal` was designed with dynamic graphs in mind. One of the advantages of the `LinkedTraversal` is that during deletes, we can ask an element which traversals it belongs to and then repair those traversals. This way, a `LinkedTraversal` can be updated when changes are made to the graph. In a list model, each traversal would have to be checked after each deletion to see if it contains the element being deleted.[6] Another potential advantage of the distributed model is the ability to use parts of existing traversals to create new ones. Changing a given traversal is also quite simple. However, for certain temporary tasks the `StoredTraversal` object is better because it stores the traversal in a local object that can then be thrown away after use. The `StoredTraversal` access times are also faster than that of `LinkedTraversal`.

The dynamically defined implementation of `Traversal` is called `CodedTraversal`. The `nextElement()` returned by the iterator of a `CodedTraversal` is defined by logic in a particular implementation rather than a stored order. Obviously, `CodedTraversal` is more storage efficient than fast. In a situation where traversals are used on a very large graph, it may actually be necessary to use the `CodedTraversal`. For a very large graph, it may be impossible to store the order of several traversals without running out of memory. However, not all traversals can be implemented as a `CodedTraversal` because the definition of some types of traversals requires storage anyway.

The last class in the toolkit of traversals that we will discuss is the `MultiTraversal`. A `MultiTraversal` can be used to build a traversal from several other traversals. The `MultiTraversal` can transparently manage iteration across all the traversals that it contains. In cases where the user has collected several paths that, together, span the graph, a `MultiTraversal` may be used to follow each path successively until all elements have been visited.

2.3. Layout

Layout is an important part of viewing a graph. Different layouts reveal different information about the data so it is important to have a variety of layouts for different tasks. Although layout uses information about the graph structure, it is not part of the graph itself but, rather, it is part of a view of the graph. A designer might be initially inclined to simply store the coordinates for a given layout in a class variable in the `Node` object. However, such a design doesn't support the storage of the coordinates from a layout for re-use. Nor does it support concurrent layout, where a graph is laid out with different layout methods in parallel. Neither does it support the storage of the same type of layout for different graphs in which the node is a member. For instance, this would necessitate running a new layout procedure every time the user selects a different layout or a different graph.

In the GVF, the coordinates of a `Node` or `Edge` are stored at the `Element`. These coordinates are associated with a particular layout and can only be retrieved by providing the relevant `Layout` and `Graph` objects as parameters. With the dual mapping of coordinates using both graph and layout as keys, we accomplish two things: it is possible to store a particular type of layout for several different graphs and it is possible to store multiple layouts for a particular graph. This allows us to have multiple graphs, each with a different layout and seamlessly switch between the layouts without waiting each time for a new layout process to finish. Furthermore, layout is often a costly process that we would like to have done in the background, if possible. Because the layout process is independent of the view, we can perform a different layout in the background on the same graph that is being viewed.

One of the advantages of the way the coordinates of a layout are accessed can be illustrated by a unique application. While building a view, we may take advantage of the fact that the coordinates from different layouts are stored at the `Element` and show a mixture of several layouts. The mixture can be of both the scaling and type of layout. For instance, when encountering a meta-node during a traversal of a clustered graph, we may choose to show the nodes in the subgraph (depending, of course, on the preferences for that view). The nodes in the subgraph may have the same layout type as in the super-graph or a different type.

[6] However, an addition will invalidate most of the traversals associated with a particular graph in either model except for nodeTraversal.

As an example, we may have a tree laid out using the traditional hierarchical Reingold-Tilford algorithm, but a subtree laid out using a radial algorithm. The negotiation mechanism that we mentioned previously may be used to determine the type of layout based on the layouts that are possible for the given graph type and the user preferences.

2.4. Metrics

As with any form of analysis, we need a set of measures that we can apply to graphs. Many graph theoretic properties are well known measures to those familiar with graph theory. The degree of a node, which is equal to the number of edges connected to that node, is a simple example. The degree of a node is a type of a metric that can be called a *local metric*, or a metric that only uses information local to the `Node`. However, a metric may involve more complex processes that use information from other parts of the graph to calculate a measure associated with a node. We call such a metric a *global metric*. A metric that uses its neighbor's metrics to determine its value is an example of a global metric. One of the challenges in the design of Metrics was to provide a uniform approach to defining a metric, which could be either local or global.

Certain measures or metrics are useful for providing visual cues when rendering a graph[1, 2]. One technique renders an edge with continuously shaded color that reflects the metric values of the nodes at its endpoints. The overall effect is the emphasis of certain edges in the graph. In another technique, the metric value for a node determines whether the node and its corresponding edges are displayed. When applying these techniques, different metrics create different effects on the view of a graph and emphasize different types of information.

Visual cues aren't the only way to use metrics. For instance, some types of layout algorithms use metric values to aid in positioning[13]. Metrics are also essential to clustering operations, during which some type of value must be assigned to an `Element` in order to decide its cluster membership. Each different area of application may make use of metrics and these metrics take on many forms. Therefore, in the GVF, we have made it possible to define a `Metric` object and associate it with an `Element`. There may also be several different metrics for any given `Element`, each one associated with a particular graph that contains the `Element`. In order to support such associations, we must again provide a dual mapping based on the metric type and the graph. We may retrieve the `Metric` object for a given `Element` by providing both the name of the metric and a graph context. Metrics may also be associated with edges.

A simpler approach might make a class variable of type `double` to store the metric value at the `Node`. However, in order to support multiple metrics with unknown algebraic properties, we must have a set of mappings to objects that implement the necessary operations. In an attempt to keep the possible applications open, we have made `Metric` an interface that defines several operators for manipulation (such as addition, subtraction, multiplication, and division) and comparison. Although we have an implementation of `Metric` that uses `double` as the internal representation of the value, it is also possible to implement the Metric interface with a different internal representation. For instance, a `Metric` based on vectors could be used without any changes to the GVF. Such a metric could store the values of several submetrics as the parameters of the vector.

3. COMPARISON WITH OTHER SYSTEMS

Although many graph visualization systems exist, most of them have been developed for specific tasks rather than as libraries for general application. However, a few general libraries have been designed and we will compare some of them to the core classes of the GVF here. Our comparison focuses on the functionality afforded by the design rather than the feature set of the applications that have been built using them. It is important to note that all of the systems in the comparison are works-in-progress and that our comparisons have been made based on the available documentation for these systems. Indeed, because of a lack of publicly available documentation, we were unable to examine certain other systems in the detail necessary for a comparison with the GVF. A system from Bell Laboratories[14] and a proprietary system from Tom Sawyer Software[15] are a few examples of systems that we might have otherwise included in this comparison.

Some libraries, such as AGD[16] and GDT[17] have impressive collections of algorithms but focus on tasks such as layout rather than interaction. However, we will discuss the GDT in the context of the solution that they provide to the graph classification problem presented in Section 2. The GDT is based on a set of libraries for combinatorial computing called LEDA[18, 19] and is written in C++. In [9], Pizzonia and Battista propose extensions to C++ called ECO C++ (Extender and Classer Oriented C++) that implement a delegation-based system. The design, which is applied within the GDT, overcomes the problem of creating a class hierarchy for dynamic graphs but requires a pre-compiler for the language extensions. Furthermore, the implementation of graph algorithms using the GDT requires familiarity with a large and specialized graph class hierarchy.

LINK[20] was developed as a portable tool for interactive graph manipulation and computation. It has an object-oriented Scheme command-line interface, with direct access to the Tk graphical user interface and an underlying set of C++ libraries. Functionality for types of manipulation common in discrete mathematics is built on Collections and Iterators. *LINK* is successful at providing a high degree of flexibility and interactivity. One of the biggest drawbacks to *LINK* is its speed, which makes it inappropriate for viewing massive data sets according to its authors.

The Graph Foundation Classes (GFC)[3] for Java became available from IBM's alphaWorks site in March 1999. The GFC contains several ideas in common with the GVF and is implemented in Java, although the current implementation is limited to JDK 1.1 features, which lacks, for example, the collections library of JDK 1.2. As with the GVF, layout and drawing are separate from the core classes. There are classes called a Walk, Trail, and Path that can be used to traverse a graph, although they are limited to traversals of alternating nodes and edges. It is also possible for nodes and edges to be shared among multiple graphs. However, different method signatures must be used in the base classes depending on whether there are multiple graphs or not.

The Graph Template Library (GTL)[4] was developed at the University of Passau in C++ based on the Standard Template Library (STL). The model for the API is based on LEDA[18, 19]. The GTL is used in order to implement a toolkit for graphs called Graphlet and can be used to implement other systems. The GTL consists of a set of data structures for describing graphs and a set of algorithms for manipulating them, including several useful algorithms such as planarity testing. It is possible to handle multiple graphs in the GTL using a feature called "hiding". It is possible to manipulate a subgraph by hiding all nodes and edges that don't belong to it and performing operations on the graph. To use the entire graph again, you are required to "unhide" nodes and edges. Although this approach makes it possible to manipulate subgraphs without building entirely new graphs, it doesn't effectively solve the problem of manipulating nested multiple graphs.

The systems that we investigated had several features in common with the GVF. All the systems are written in an object-oriented language and implement graph traversal with a storage object and some form of iteration. Most systems have a way to associate data or properties with either a node or an edge, although explicit support for metrics was generally absent. Most systems are also capable of specifying the visual attributes of elements and provide several choices for layout.

Perhaps the most noticeable difference between the GVF and the systems discussed above is the GVF's use of properties to distinguish between graph types such as trees and directed graphs. All the other systems except for the Graphics Template Library employ a hierarchy of graph classes. We have already pointed out the disadvantages of such an arrangement for dynamic graphs (see Section 2). Although there was some support for multigraphs in the GFC, other systems did not appear as capable of handling the nested multigraph structures that the GVF was designed to handle. This was generally due to the fact that graph objects have their own separate branch in the object hierarchy and keep lists of the nodes and edges that they contain.

The node-centric design of the GVF provides all the features needed to visualize nested multigraphs in a flexible way. Although nested multigraph structures aren't necessary for all tasks, they are essential for the representation of multiple clusterings. We feel that this simple but genuine idea, together with our application of properties to overcome the problems of a graph type hierarchy, make the GVF a unique foundation for the visualization of graphs.

4. Conclusions

The special requirements of fully interactive graph visualization have been used to create a framework that is uniquely equipped to handle such things as dynamic and nested multigraphs. Although the libraries discussed here are still fairly new, they have proved to be useful in implementing a graph visualization system that is capable of meeting the requirements that we have discussed. The flexibility and extensibility of the GVF has been tested by a small group of researchers in the field.

There is an artificial division between different types of visualization systems because of the specialized knowledge and tools that are currently necessary to implement them. However, it is not difficult to imagine that graph visualization could be useful to view relations that have been induced in the data, such as the relations resulting from clustering. Because clustering is so frequently used in information visualization applications that are not necessarily graph-oriented, it could be argued that the core classes would be useful as a base class for representing data.

One of the goals for our system still hasn't been achieved. Graphs with millions of nodes still remain out of reach. Future work may involve methods that allow us to incrementally navigate such structures, for example, storing a graph in a database or dynamically retrieving graph information from a remote process.

References

[1] I. Herman, G. Melançon, and M. Delest, "Tree Visualisation and Navigation Clues for Information Visualisation", *Computer Graphics Forum*, vol. 17, pp. 153–165, 1998.

[2] I. Herman, M. S. Marshall, G. Melançon, D. Duke, M. Delest, and J.-P. Domenger, "Skeletal Images as Visual Cues in Graph Visualization", in *Data Visualization '99, Proceedings of the Joint Eurographics and IEEE TCVG Symposium on Visualization*, E. Gröller, H. Löffelmann, and W. Ribarsky, Eds. Wien: Springer–Verlag, 1999, pp. 13–22.

[3] C. L. Cesar, *Graph Foundation Classes for Java*, IBM, http://www.alphaWorks.ibm.com/tech/gfc, (1999).

[4] M. Forster, A. Pick, and M. Raitner, *Graph Template Library*, University of Passau, http://infosun.fmi.uni-passau.de/GTL/, (1999).

[5] I. Herman, M. S. Marshall, and G. Melançon, "Graph Visualization and Navigation in Information Visualization", *(to appear) IEEE Transactions on Visualization and Computer Graphics*, vol. 6, 2000.

[6] B. Eckel, *Thinking in Java*, Prentice Hall, 1998.

[7] C. Upson, J. Thomas Faulhaber, D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. v. Dam, "The Application Visualization System: A Computational Environment for Scientific Visualization", *IEEE Computer Graphics & Applications*, pp. 30-42, 1989.

[8] F. Arbab, I. Herman, and G. J. Reynolds, "An Object Model for Multimedia Programming", *Computer Graphics Forum*, vol. 12, pp. 101-113, 1993.

[9] M. Pizzonia and G. D. Battista, "Object-Oriented Design of Graph Oriented Data Structures", *ALENEX*, 1999.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*, Addison-Wesley Professional Computing Series, 1995.

[11] D. J. Duke, I. Herman, and M. S. Marshall, *PREMO: A Framework for Multimedia Middleware: Specification, Rationale, and Java Binding*, Springer, 1998.

[12] C. J. Alpert and A. B. Kahng, “Recent Developments in Netlist Partitioning: A Survey”, *Integration: the VLSI Journal*, vol. 19, pp. 1-81, 1995.

[13] R. M. Wilson and R. D. Bergeron, “Dynamic Hierarchy Specification and Visualization”, in Proceedings of Information Visualization, 1999.

[14] T. He, “Internet-Based Front-End to Network Simulator”, in Proceedings of Data Visualization '99, Vienna, Austria, 1999.

[15] *The Graph Editor Toolkit for Java White Paper*, Tom Sawyer Software, http://www.tomsawyer.com/get/paper-java.html, (1999).

[16] P. Mutzel, C. Gutwengwer, R. Brockenauer, S. Fialko, G. Klau, M. Kruger, T. Ziegler, S. Naher, D. Alberts, D. Ambras, G. Koch, M. Junger, C. Bucheim, and S. Leipert, “A Library of Algorithms for Graph Drawing”, in Proceedings of Symposium on Graph Drawing GD'98, Berlin, 1998.

[17] ALCOM-IT, *Graph Drawing Toolkit*, Third University of Rome, http://www.dia.uniroma3.it/~gdt/, (1999).

[18] K. Mehlhorn and S. Näher, *LEDA: A Platform for Combinatorial and Geometric Computing*, Cambridge University Press, 1999.

[19] *LEDA Research*, Algorithmic Solutions GmbH, http://www.mpi-sb.mpg.de/LEDA/, (1999).

[20] J. Berry, N. Dean, M. Goldberg, G. Shannon, and S. Skiena, “Graph Drawing and Manipulation with LINK”, in Proceedings of Graph Drawing, 1997.