Algorithmic Patterns for Orthogonal Graph Drawing*

Natasha Gelfand *Roberto Tamassia*

Department of Computer Science
Brown University
Providence, Rhode Island 02912–1910
{ng,rt}@cs.brown.edu

Abstract. In this paper, we present an object-oriented design and implementation of the core steps of the GIOTTO algorithm for orthogonal graph drawing. Our design is motivated by the goals of making the algorithm modular and extensible and of providing many reusable components of the algorithm. This work is part of the JDSL project aimed at creating a library of data structures and algorithms in Java.

1 Introduction

In the last few years, there has been an increasing interest in applications of software engineering concepts, such as object-oriented programming and design patterns, to the area of design and implementation of data structures and algorithms (see, e.g. [19, 18]).

Traditionally, algorithms have been implemented in a way that would maximize their efficiency, which frequently meant sacrificing generality and extensibility. The area of *algorithm engineering* is concerned with finding ways to implement algorithms so that they are generic and extensible. One of the proposed concepts is that of an *algorithmic pattern* [18]. Similar to the design patterns [9] in software engineering, algorithmic patterns abstract and generalize the algorithms and data structures of common use. They allow the programmer to implement new algorithms by extending already existing ones, thus reducing the time spent coding and debugging. In many cases, algorithmic patterns abstract out the core of several similar algorithms, and the programmer only has to implement the extensions of the core that are needed for a specific algorithm. The object-oriented approach to algorithm implementation incurs some overhead due to the indirection, and yields implementations that are slower than the ad-hoc ones. For applications where fast running time is essential, the object-oriented approach is intended to be used for rapid prototyping of new algorithms, which, once fully designed, can be efficiently implemented using traditional methods.

In this paper, we describe an object-oriented design and implementation of the core steps of GIOTTO drawing algorithm [17], *orthogonalization* and *compaction*, which construct a planar orthogonal drawing of an embedded planar

* Research supported in part by the U.S. Army Research Office under grant DAAH04-96-1-0013 and by the National Science Foundation under grants CCR-9732327 and CDA-9703080

graph with the minimum number of bends [16]. We present a new algorithmic pattern called *algorithmic reduction*, which provides conversion among different types of data structures. Reductions are often used to convert data structures into the form required by certain algorithms. GIOTTO is particularly suitable as a case study for the use of algorithmic reductions because it consists of many steps where the original graph is converted into various other structures (e.g., a flow network).

Our implementation of GIOTTO is part of the GEOMLIB project [18]. GEOMLIB is a reliable and open library of robust and efficient geometric algorithms written in Java. It is part of a larger project, named JDSL [10, 11], aimed at constructing a library of algorithms and data structures using the Java programming language.

The rest of this paper is organized as follows. In Section 2, we present a brief description of the JDSL objects used in our implementation of GIOTTO, especially the concept of *decorations*. In Section 3, we describe how algorithms are implemented in JDSL and introduce the concept of algorithmic reductions. The object-oriented design of GIOTTO is described in Section 4. In Section 5, we compare our implementation with two other implementations of GIOTTO.

2 JDSL Structures

Data Structures for Graphs JDSL contains three different graph structures represented by the interfaces Graph, OrderedGraph, and EmbeddedPlanarGraph. All graphs are "mixed graphs," that is, they can contain both directed and undirected edges at the same time. The interfaces provide methods for dealing with the different types of edges both together and separately.

The Graph interface models a graph as a combinatorial object, that is a container of vertices and edges. The Graph interface provides methods for adding and removing vertices and edges, as well as methods for examining the graph structure.

The OrderedGraph interface inherits from the Graph interface and describes a graph as a topological structure by adding information about the ordering of the edges around each vertex. OrderedGraph provides additional methods that manipulate the topological information.

The EmbeddedPlanarGraph interface describes an ordered graph whose ordering of the edges around each vertex is induced by a planar embedding of the graph (hence it extends OrderedGraph). In addition to storing vertices and edges, EmbeddedPlanarGraph also stores the faces of the embedding. It provides additional methods that access various information about the faces of the embedding.

Decorations Frequently, in implementing algorithms, it is convenient to associate extra information, either permanent or temporary, with the elements of the algorithm's input. In graph algorithms, the elements that we want to augment with extra information are vertices, edges, and faces of the input graph. This extra information, called *decorations* or *attributes*, can be used as temporary scratch data (for example, to mark vertices of a graph as visited during

a traversal algorithm), or to represent the output (for example, to store geometric information in a graph drawing algorithm). In writing the pseudocode of an algorithm, decorations are used implicitly (we often say "mark vertex v as visited"), and it is left to the programmer to determine exactly how they should be implemented (for sample implementations see, e.g [15, 6]).

In the JDSL framework, each vertex, edge, and face contains a hash table for storing its decorations. Each decoration has a key (which is its key in the hash table) and a value, both represented by a Java Object. The interfaces Vertex, Edge, and Face extend the interface Decorable, which provides methods for creating, deleting, and setting the values of decorations, as well as accessing all decorations of the object at once. The use of decorations is intuitive, since for each action that may potentially be performed on a decoration, the Decorable interface provides a corresponding method. Thus, most manipulations of decorations require only a single method call.

3 Algorithms and Reductions

Algorithms as Objects In the JDSL framework, algorithms are modeled by objects instead of just procedures (an approach also discussed in [7]). This approach presents the advantage of being able to store within the algorithm its input and output, as well as other information such as auxiliary variables and data structures used by the algorithm. In other words, this approach gives a state to the algorithm. An algorithm object is created with an instance of its input, provides a way to compute its output (frequently right in the constructor), and a variety of accessor methods that can be used to examine its state once the computation is finished.

As an example, consider again an algorithm that performs a depth-first traversal starting at a given vertex. The main output of this algorithm is a depth-first tree corresponding to the traversal. Thus, the algorithm object will provide a method that can be used to examine the tree once it is computed. However, in the course of computing the tree, the algorithm uses internal marking to indicate that a given vertex has been visited. Usually, this marking is only temporary (the data structure that stores it is local to the algorithm procedure). If an algorithm is modeled as an object, the marking can be stored inside the object and will remain valid even after the computation is finished. The algorithm object will provide a method to access this internal data, which can be used, for example, to determine whether a given vertex belongs to the connected component of the start vertex. The algorithm's state persists as long as the algorithm object does (until it is deleted or garbage collected), and can be examined any number of times without having to be recomputed.

Modeling algorithms as objects and storing the results of their computation as state allows the same algorithm to compute several different results. In the course of computing a DFS tree, the depth-first traversal algorithm can also classify edges as discovery and back, compute the start and finish times for each vertex, etc. This information is computed as a side-effect of computing a DFS tree, but if it is stored inside the algorithm object, it allows the object to be used for purposes other than just traversal. In fact, the user who is only interested in

whether a given edge is discovery or back need not be aware of the fact that the algorithm object computes other information as well. The user can instantiate the algorithm object with the desired input, and use the accessor method that returns the type of each edge, ignoring any other accessors the object may have.

Another advantage to modeling algorithms as objects is the ability to define relationships among different algorithms. An algorithm can extend another and specialize some of its parts, several algorithms can implement the same interface or extend the same base class, or an algorithm can use another as a subcomponent.

Reductions In this section, we describe the pattern of *algorithmic reduction*. Often an algorithm can be implemented by making some alterations to its input, using another algorithm on the modified input, and then undoing the modifications and interpreting the algorithm's results to provide the answer to the original problem. When this is the case, all too often the transformations and the algorithms are clumped together, preventing reuse of any of the components. A reduction is an object responsible for the first and the last step of the computation described above, that is for transformation of the input to fit an algorithm's requirements and for undoing the alterations and interpreting results.

A reduction, just as an algorithm, is modeled by an object. A reduction object is created with an instance of its input and proceeds in two directions, forward and reverse. In the forward direction the reduction alters its input in a specified way, often to make it fit the input requirements of some algorithm. Once an algorithm has been run on the modified input, the reduction undoes the changes to its input and possibly transforms the output of the algorithm to provide an answer to the original problem that used that reduction. Thus, a generic Reduction object has two methods forward() and reverse() and its functionality is to report an error when reverse() is called before forward() (since in that case the reduction does not make sense). The subclasses of this class define these methods to carry out the specific transformations.

Reductions and algorithm objects are frequently combined together as components of other algorithms. By dividing the components of an algorithm's implementation into reductions and algorithms, we clearly separate the transformation and computation steps (which usually alternate). In this scheme, algorithms should never modify the structure of their input, all modifications are done by the reductions so that they can be undone later. When we look at an implementation of an algorithm, we can easily identify which steps modify the input and which just perform computations, since they are modeled by fundamentally different objects.

4 GIOTTO Implementation

In this section, we show how using the JDSL components and algorithmic patterns described above, we can create a simple, modular, and extensible implementation of the core steps of the GIOTTO algorithm: *orthogonalization* and *compaction*.

Our implementation consists of two main components each modeled by an algorithm object. The first component is the *orthogonalization* algorithm, which

constructs an orthogonal representation of a given embedded planar graph. In the current implementation, this algorithm accepts only embedded 4-planar graphs (embedded planar graphs whose vertices have degree at most four), however we plan to add a reduction from general embedded planar graphs to embedded 4-planar graphs, which would allow this algorithm to operate on any embedded planar graph. The second part of the implementation is the *compaction* algorithm, which accepts as input a 4-planar graph and its orthogonal representation and produces a planar orthogonal drawing where each vertex is represented by a point, and each edge by a chain of vertical and horizontal segments. Although these algorithm objects were developed as parts of the overall GIOTTO implementation, their modular design allows them to be used independently, which means they can be reused as parts of other algorithms.

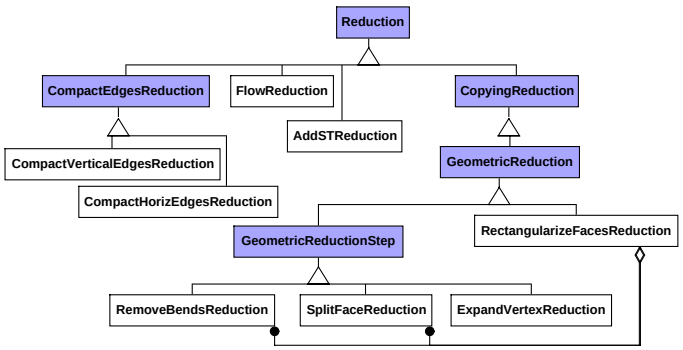

Fig. 1. Inheritance hierarchy of the reductions used by our implementation of the GIOTTO algorithm. Abstract classes are represented by shaded boxes.

Orthogonalization Algorithm The orthogonalization algorithm [16] accepts as its input an embedded 4-planar graph with a fixed external face and produces an orthogonal representation of the graph with the minimal number of bends. We briefly review the algorithm following the description in [4]. For each (undirected) edge with endpoints u and v, call the two possible orientations (u, v) and (v, u) *darts*. The *orthogonal representation* of a graph is defined by assigning to each dart values α and β defined as follows:

- $\alpha(u, v) \cdot \pi/2$ is the angle at vertex u formed by the first segment of this dart and the next dart counterclockwise around u;
- $\beta(u, v)$ is the number of left turns of value $\pi/2$ that are made when traversing the dart from origin to destination.

The implementation of the algorithm is broken down into several objects that model the computational steps above.

Orthogonalize This object models the orthogonalization algorithm itself. Its input is an instance of EmbeddedPlanarGraph and a designated external face which are provided to the object's constructor (where the computation of the orthogonal representation is performed). Orthogonal representation is modeled by associating with each edge an array of two Dart objects (with

computed α and β values) using a decoration. The key to this decoration as well as all intermediate data generated by the computation (e.g. a flow network and the flow information) can be retrieved from the object using the accessor methods.

FlowReduction (see Figure 1) This subclass of Reduction is responsible for converting an EmbeddedPlanarGraph into a flow network (modeled by an object of type Graph) and then interpreting the results of the minimum cost flow algorithm to compute an orthogonal representation. In the forward() method of the reduction, the flow network is constructed piecewise by each dart. The reverse() method, called once a flow algorithm has decorated each edge of the network with the amount of flow in it, computes the orthogonal representation by calling an appropriate method of each dart that interprets the flow and computes α and β values.

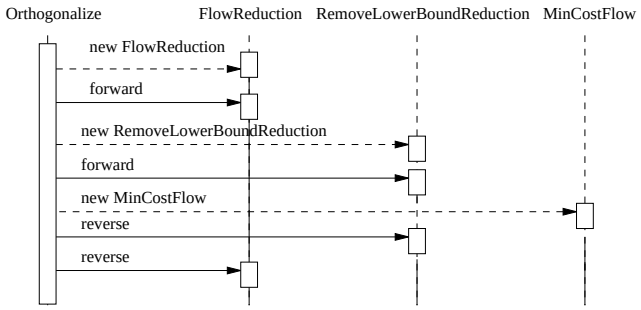

Fig. 2. Interaction diagram of the orthogonalization algorithm

The interaction diagram of the orthogonalization algorithm is shown in Figure 2. The computation, which is done in a separate protected method called from the constructor, proceeds by creating an instance of FlowReduction and calling its forward() method to build the flow network. The flow computation in the resulting network is done in a separate method. This provides the flexibility of allowing the user to redefine the minimum cost flow algorithm that is used. We provide a default implementation, but using another implementation is easy, all that is necessary is subclassing the Orthogonalize class and redefining the computeFlow method. All information necessary for the computation is passed to this method so that the programmer need not know the internal state of the algorithm object. This is an example of the *template method pattern* [9, 8] that often comes up in the object-oriented design of algorithms.

In order to make Orthogonalize fully functional, we provide an implementation of computeFlow method using the cycle-annealing algorithm described in [1]. The algorithm assumes that the edges of the network do not have a lower bound, so a reduction is used to transform the flow network into the form required by the algorithm. The forward() method adjusts the production of vertices and capacities of edges so that the lower bound can be removed. The reverse() method

adjusts the computed flow to correspond to the correct flow in the original network.

Once the lower bound is removed, we can use the cycle-annealing algorithm to compute the minimum cost flow. This algorithms uses several other objects as subcomponents. The initial flow in the network is computed by an object modeling the Ford and Fulkerson augmentation algorithm. The flow is stored as a decoration of the edges of the network. This flow algorithm operates on a single-source, single-sink flow network, so a reduction described in Section 3 is used to convert the network produced by the FlowReduction into the required form. The negative cost cycles are computed by a subclass of a Bellman-Ford algorithm object.

Compaction Algorithm The compaction algorithm [16] takes as its input an embedded planar graph and its orthogonal representation and produces a drawing of the graph by associating a point with each vertex and a chain of vertical and horizontal segments with each edge. We briefly review the algorithm following the description in [4].

The algorithm first transforms the input graph into a graph whose faces all have rectangular shape by adding fictitious vertices at edge bends and decomposing the non-rectangular faces into rectangles by means of fictitious edges. The algorithm then computes the length of the edges in the resulting graph. There are several ways to compute edge lengths, so the algorithm's design again uses a template method to allow the user to specialize the computation. The default implementation computes the lengths of the vertical and horizontal edges separately using an optimal weighted topological numbering algorithm. To compute the lengths of the vertical edges, the algorithm orients each vertical edge from top to bottom, condenses maximal runs of horizontal edges into vertices and computes an optimal weighted topological numbering of the resulting planar *st*-graph with respect to unit edge lengths. The length of a vertical edge is set to be the difference between the topological numbers of its two endpoints. The lengths of the horizontal edges are computed analogously. The final step of the algorithm generates a geometric object for each vertex and edge of the input graph.

The compaction algorithm is modeled by the CompactOrthogonal algorithm object. The computation of the drawing can be broken up into three stages: the refinement of faces into rectangles, performed in the forward() method of the RectangularizeFacesReduction, the length computation, performed in the computeLengths method of CompactOrthogonal class, and the assignment of geometric objects, performed in the reverse() method of the reduction. The interaction diagram of this algorithm is shown in Figure 3.

We will first describe the objects used in the computation of edge lengths.

CompactEdgesReduction (see Figure 1) This abstract subclass of the generic Reduction takes as its input an EmbeddedPlanarGraph and its orthogonal representation in the form of Dart objects attached to the edges. The forward() method of the reduction produces a planar *st*-graph by condensing some edges into vertices, and orienting other edges in a given direction. The

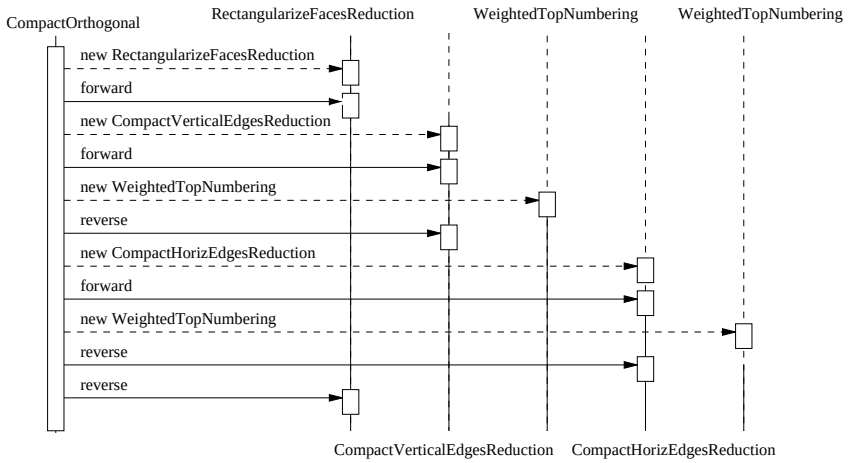

Fig. 3. Interaction diagram of the compaction algorithm

discriminators for identifying which edges should be condensed and which oriented are provided by the subclasses of this class. The reverse() method anticipates that the vertices of the *st*-graph have been decorated with their topological numbering and uses the numbering to compute the lengths of the edges of the original graph. The lengths are stored as decorations of the edges.

UnitWeightedTopNumbering This objects models an algorithm for computing optimal weighted topological numbering of a digraph. The algorithm stores the numbering of each vertex as a decoration and provides a method to access the decoration's key.

The computeLnegths method proceeds by creating an instance of CompactVerticalEdgesReduction and calling its forward() method to obtain a graph corresponding to the input graph with contracted runs of vertical edges. In the next step, an instance of UnitWeightedTopNumbering is constructed using the graph obtained from the previous step as its input, the execution of this algorithm decorates the vertices of the graph with their topological numbers. The reverse() method of the reduction is called next, it uses the just computed numbering decoration to compute lengths of the horizontal edges and store them using another decoration. Analogously, vertical edge lengths are computed using a CompactHorizEdgesReduction and another instance of UnitWeightedTopNumbering.

We now turn our attention to the refinement procedure. In order to transform the faces of the graph so that they all have rectangular shape, it is necessary to add fictitious vertices at edge bends and split non-rectangular faces into rectangular components with fictitious edges. These operations present a book-keeping complication. In the JDSL implementation of graph structures, when an edge is split with a vertex, two new edges are generated, while the original edge becomes invalid. A similar situation occurs when a face is split with an edge. We would like to implement a reduction that would modify the original graph by splitting some of its components, but once reversed give back the unaltered graph. In order to do this, however, we may have to keep track of the edges that were

split and insert them back when the reduction is reversed. Such book-keeping is rather cumbersome to implement. Instead, we propose an alternative strategy. Whenever a reduction *destructively* modifies its input graph, it creates an exact copy of that graph first and makes the alterations on the copy. Mapping between the elements of the original graph and the elements of the copy can be done through decorations. Each vertex, edge, and face of the original graph is decorated with the corresponding element of the copy, and vice versa. When the reduction is reversed, instead of trying to patch up the modified graph, it uses the information computed for the copy (which usually takes form of decorations) to determine the correct output for the original graph. Once all necessary information is transfered from the copy to the original, the copy graph can be thrown away (deleted or garbage collected).

Operations that slightly alter the structure of a graph to make it satisfy a given criterion are quite common in graph drawing algorithms. In fact, the need to model such operations generically was the main motivation for developing the pattern of algorithmic reductions. We have developed a hierarchy of graph-modifying reductions (see Figure 1) that reflects the different situations where these reductions can be used.

CopyingReduction This subclass of the Reduction class is the parent of all reductions that act on instances of Graph interface and destructively modify their input. The class provides a method which creates an exact copy of the reduction's input graph. The reduction also provides keys to two decorations: mapping from the elements of the original graph to the elements of the copy and vice versa.

GeometricReduction This reduction is the superclass of structure-altering reductions used by graph drawing algorithms. The graph objects that it operates on are of type EmbeddedPlanarGraph. Subclasses of this reduction are usually structured as follows: in the forward() method the input graph is copied and altered. The reverse() method computes the geometric shapes for the elements of the input graph based on the information computed by an algorithm acting on the copy of the original graph. Geometric information is stored as decorations of the elements of the input graph.

RectangularizeFacesReduction, which performs the refinement procedure in the CompactOrthogonal algorithm is a subclass of GeometricReduction since in its forward step it makes alterations to the graph structure, and in the reverse step it computes geometric information for the graph. The reduction proceeds by making many elementary changes to the graph structure, such as splitting an edge or a face. Each of these changes can in turn be modeled as a separate object, with several of such objects combined to form the complete reduction. Thus, another subclass of GeometricReduction is GeometricReductionStep, which models an alteration of one element (vertex, edge, or face) of a graph structure. Subclasses of this class operate on a copy of the original graph (since the changes they may make destructive changes), but they may either use a copy provided to their constructor, or create one of their own. Each GeometricReductionStep is created with a graph element that it will modify in its forward() method and pos-

sibly decorate with its geometric representation in the reverse() method. GIOTTO implementation uses the following subclasses of the GeometricReductionStep

RemoveBendsReduction This reduction models removal of bends from a single edge of a graph. In the forward() method of the reduction the corresponding copy edge it split by inserting a "dummy" vertex for each bend of the edge. The reverse() method of the reduction decorates the input edge with its geometric representation — a chain of vertical and horizontal segments. The reduction uses the information about the length of each edge, which was computed between the calls to forward() and reverse() to compute the geometric representation.

SplitFaceReduction The input of this reduction is a face of the graph which is not rectangular. The forward() method of the reduction decomposes the corresponding face of the copy into its rectangular components by splitting it with several edges. The reduction determines the locations for the splitting edges by traversing the face counterclockwise performing a split every time a right turn is encountered. Since a face does not have a geometric representation in the drawing reverse() method of this reduction is empty.

ExpandVertexReduction This object will be used as a subcomponent of the reduction object to convert a general embedded planar graph into 4-planar.

In its forward() method RectangularizeFacesReduction creates an instance of RemoveBendsReduction for each edge that has bends, and an instance of SplitFaceReduction for each non-rectangular face. The reduction steps are constructed with the copy of the original graph created by the RectangularizeFacesReduction, since there is no need to copy the graph for each little alteration. Executing forward() methods of all reduction steps refines all faces of the copy graph into rectangles. In the reverse() method, the reduction creates geometric representations of the vertices of the graph (since no vertex refinement was done), undoes the step reductions, which creates geometric representations for the edges with bends, and finally creates geometric representations for the remaining edges. Since all alterations were made on a copy, there is not need to make any modifications to the input graph.

5 Design Evaluation and Comparison

Graph Hierarchy JDSL does not make a detailed classification of graphs and their drawings based on their specific properties. All types of graphs are modeled by the Graph interface if they contain only combinatorial information, and by EmbeddedPlanarGraph interface if they also contain the topological information. Thus, in the JDSL framework there are no special classes for a DAG, a biconnected graph, a 4-planar graph, etc. A DAG, for example, is just a Graph which happens to have only directed edges and no directed cycles.

A design which models each graph type which has some special properties with a separate class is also possible. This approach is used in the design of the GDToolkit [12] graph drawing package (the successor to Diagram Server [2, 5]). GDToolkit provides a hierarchy of graph classes, where addition of spe-

cial properties or structure is modeled through inheritance. For example, an orthogonal planar undirected graph adds to a planar undirected graph information about the number and angles of bends, and is, therefore, modeled as its subclass.

In the JDSL framework, the conversion between different types of graphs does not require creating new graph objects, just modifying existing ones. For example, to create a directed graph from an undirected one, all that is required is to set the direction of the edges using, e.g., the setDirectionFrom(Edge, Vertex) method of the Graph interface. If a directed graph is modeled by a separate class, as it is done in GDTOOLKIT, a new object has to be created, using the original graph as a parameter in the conversion constructor. The drawback of creating a new object is the fact that a *given* graph cannot be made directed, a conversion constructor will create a graph which is a directed *copy* of the original. This presents a difficulty since any objects referencing the vertices and edges of the original graph will have to be updated to reference the elements of the new one or a mapping between the new and old elements will have to be created.

On the other hand, not having separate classes for the special types of graphs complicates error checking. In JDSL, since there is no type for a digraph, an algorithm that operates on digraphs takes just a Graph as its input. It then has to check at *runtime* that its input is correct, that is that the graph only has directed edges, and report an error if that is not the case. If a digraph is implemented as its own type, however, any algorithm that accepts an object of that type can assume that it is inherently correct, and thus does not have to do any error checking of its input.

Decorations As described in Section 2, decorations are very useful in implementing various graph algorithms. There is a number of ways to implement decorations, but to provide a good implementation the following guidelines should be adhered to: (*i.*) Decorations should be easy to use. A procedure for identifying and manipulating a decoration that belongs to a vertex, edge or face of a graph should be easy and intuitive. (*ii.*)The number of decorations that can be added to a graph element (vertex, edge or face) should not be limited, otherwise that element will not be extensible.

JDSL implements decorations by associating with each element of the graph a hash table for storing its decorations. This scheme satisfies both conditions, since any number of decorations can be stored in an element's hash table, and accessing decorations requires just a method invocation.

An alternative implementation is used in the LEDA library [15], which provides decorations for graphs and planar maps through *node-*, *edge-*, and *face-arrays*. A node-array stores the decorations for the vertices of a graph in a C++ vector and uses the internal numbering of the vertices to access decorations of a given vertex. The edge and face arrays are implemented in similar fashion. This scheme also satisfies both conditions above since creating new decorations just requires making more vectors, and the mapping between elements and their decorations is done internally, so that the user can easily access the necessary decorations.

JDSL's implementation has one advantage over LEDA's — the decorations and the decorated objects are tightly coupled, which makes it easier to implement several actions. In the LEDA implementation, it is difficult to access all decorations of a given object, since they may be spread out through the entire implementation of the algorithm. In the JDSL implementation, the Decorable interface provides the method attributes, which returns an enumeration of all the decorations of that object. Coupling decorations and the decorated objects also provides more fine grained control over the access to the decorations. In LEDA implementation, any object that has access to, e.g., a node-array can modify a decoration of any vertex. In JDSL implementation in order to access an object's decoration, one needs a reference to that object first.

The JDSL system of decorations is not limited to just graph data structures. Nodes of sequences and trees can also be decorated, which simplifies implementation of such data structures as red-black trees, where a node's color can be stored in a decoration.

GIOTTO Implementation There are currently several other implementations of GIOTTO, provided as parts of graph drawing packages (see, e.g., [2, 3, 13, 14]). In this section we compare some of the features of two such implementations, provided in GRAPH DRAWING SERVER [3] and GDTOOLKIT [12], with our implementation described in Section 4. We will examine each implementation with respect to the following criteria. *Modularity:* How the algorithm is broken up into subcomponents. *Extensibility:* How easy it is to make modifications to the algorithm and extend its functionality. *Ease of use:* How clearly defined is the interface through which the algorithm should be used. *Use of graph structures:* Which graph structures the implementation uses and how it uses them. *Alterations to the input:* Several steps of GIOTTO make modifications to the input graph such as adding edges and vertices which need to be removed in the final output. We examine how each implementation adds the new elements, keeps track of them throughout the execution, and removes them.

The GRAPH DRAWING SERVER [3] is a web-based graph drawing and translation facility. The user submits a graph in one of a number of formats, selects a drawing algorithm to be used and specifies an output format with a request to the server. The implementation of GIOTTO that is part of the server is not meant for use outside the server, and therefore has several application-specific features. *Modularity:* The component breakdown is limited to placing major algorithms into separate procedures. *Extensibility:* This particular implementation is hard to extend, since many of its components were developed specifically for use by the GIOTTO algorithm and were not meant to be reused for other purposes. *Ease of use:* Using the GIOTTO implementation requires calling several functions corresponding to the steps of the algorithm in order. A unifying function for GIOTTO algorithm is not provided. *Use of graph structures:* This implementation of GIOTTO uses a special data structure to keep track of the graph and other information. The data structure is highly specific to the implementation. *Alterations to the input:* Fictitious vertices and edges are added directly to the main data structure and labeled as such. In the stage when the algorithm computes its

final output, the labels are examined and elements that are labeled as fictitious simply do not get included in the output.

GDTOOLKIT [12] provides an implementation of several graph drawing algorithms. As a consequence, it has a supporting library of graph data structures, and also uses several LEDA components. Although the main purpose of the package is for displaying graphs, it is developed in such a way as to make it possible to add new algorithms. *Modularity:* The library provides several graph structures, each modeled by a class. The implementation of GIOTTO, however, is not separated from the graph class that it is applied to. Also, algorithmic subcomponents (such as the minimum cost flow computation) are not separated from the main GIOTTO implementation. *Extensibility:* The implementation of GIOTTO is a private function of the embedded planar graph structure, which means that extending its functionality, even through inheritance, is not possible. Therefore it is difficult to use the algorithm for anything other than computing a drawing and displaying it on the screen. *Ease of use:* Using the GIOTTO algorithm in GDTOOLKIT is very easy. To create an orthogonal representation for a given graph instance, that instance needs to be assigned to an object of type orthogonal planar graph. The assignment automatically performs the conversion and runs the orthogonalization algorithm. *Use of graph structures:* GDTOOLKIT provides several graph structures to be used by graph drawing algorithms. One of the classes is used to model a drawing as a graph which contains information about the coordinates of its elements and can be drawn in a window. GIOTTO proceeds by first constructing a drawing object with an unspecified layout, which is a copy of its input graph and then creating an orthogonal representation for the drawing. *Alterations to the input:* This implementation of GIOTTO uses a marking system similar to the one used in the GRAPH DRAWING SERVER to indicate fictitious elements. The object that models a drawing provides a method to remove the fictitious elements that can be called before displaying a graph.

The main function of both the GRAPH DRAWING SERVER and of GDTOOLKIT is to apply a variety of drawing algorithms to graphs provided by the user, and to display the resulting drawing. These packages are not intended to be used as libraries for programmers who want to develop new algorithm implementations. As a consequence, many of their components are application-specific and thus are difficult to reuse outside the package.

The implementation of GIOTTO presented in this paper is intended to be used in a variety of applications. It is therefore aimed at providing components that are reusable and can be easily adapted to the required application. *Modularity:* Each object and reduction in the implementation of GIOTTO is modeled by a separate object with a well-defined interface. Algorithms are used as subcomponents of other algorithms by instantiating them, computing the output, and using the desired accessor methods. *Extensibility:* GIOTTO and its subcomponents are designed in such a way as to be easily extended for use in various applications. All algorithms are modeled as objects, which means they can be extended through inheritance. The template method pattern is used when there are alternative algorithms for a given action, or when there may be user-defined

actions taken at a given step. *Ease of use:* Since the GIOTTO algorithm is modeled by an object, using it just requires creating an instance of that object passing to it the desired input graph in the constructor. *Use of graph structures:* Our implementation of GIOTTO uses the Graph and EmbeddedPlanarGraph interfaces described in Section 2. *Alterations to the input:* Our implementation of GIOTTO does not make any changes to the input data structure, it only adds decorations. Any alterations that are done to the input are handled by the reduction objects which have the ability to undo any changes they make.

References

1. R. Ahuja, T. Magnanti, and J. Orlin. *Network Flows.* Prentice Hall, 1993.
2. P. Bertolazzi, G. Di Battista, and G. Liotta. Parametric graph drawing. *IEEE Trans. Softw. Eng.*, 21(8):662–673, 1995.
3. S. Bridgeman, A. Garg, and R. Tamassia. A graph drawing and translation service on the WWW. In S. C. North, editor, *Graph Drawing (Proc. GD '96)*, Lecture Notes Comput. Sci. Springer-Verlag, 1997.
4. G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing:*. Prentice-Hall, 1998.
5. G. Di Battista, G. Liotta, and F. Vargiu. Diagram Server. *J. Visual Lang. Comput.*, 6(3):275–298, 1995. (special issue on Graph Visualization, edited by I. F. Cruz and P. Eades).
6. ffgraph homepage. http://www.fmi.uni-passau.de/~friedric/ffgraph/main.shtml.
7. B. Flaming. *Practical Algorithms in C++.* Coriolis Group Book, 1995.
8. G. Gallo and M. G. Scutella. Towards a programming environment for combinatorial optimization: a case study oriented to max-flow computations. *ORSA J. Computing*, 5:120–133, 1994.
9. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley, 1995.
10. N. Gelfand, M. T. Goodrich, and R. Tamassia. Teaching data structure design patterns. In *Proc. 29th ACM SIGCSE Tech. Sympos.*, pages 331–335, 1998.
11. M. T. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java.* John Wiley, New York, NY, 1998.
12. GDToolkit homepage. http://www.inf.uniroma3.it/people/gdb/wp12/GDT.html.
13. M. Himsolt. The Graphlet system. *Lecture Notes in Computer Science*, 1190, 1997.
14. H. Lauer, M. Ettrich, and K. Soukup. GraVis — system demonstration. *Lecture Notes in Computer Science*, 1353, 1997.
15. LEDA homepage. http://www.mpi-sb.mpg.de/LEDA/.
16. R. Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM J. Comput.*, 16(3):421–444, 1987.
17. 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.
18. R. Tamassia, L. Vismara, and J. E. Baker. A case study in algorithm engineering for geometric computing. In G. F. Italiano and S. Orlando, editors, *Proc. Workshop on Algorithm Engineering*, 1997. http://www.dsi.unive.it/~wae97/proceedings/.
19. K. Weihe. Reuse of algorithms: Still a challenge to object-oriented programming. In *Proc. OOPSLA-97*, pages 34–48. ACM Press, 1997.