SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2001; **31**:615–635 (DOI: 10.1002/spe.379)

JGAP: a Java-based graph algorithms platform

Ding-Yi Chen[1], Tyng-Ruey Chuang[1] and Shi-Chun Tsai[2,*,†]

[1]*Institute of Information Science, Academia Sinica, Taipei 115, Taiwan*
[2]*Department of Information Management, National Chi-Nan University, Nantou 545, Taiwan*

SUMMARY

We describe JGAP, a web-based platform for designing and implementing Java-coded graph algorithms. The platform contains a library of common data structures for implementing graph algorithms, features a 'plug-and-play' modular design for adding new algorithm modules, and includes a performance meter to measure the execution time of implemented algorithms. JGAP is also equipped with a graph editor to generate and modify graphs to have specific properties. JGAP's graphic user interface further allows users to compose, in a functional way, computation sequences from existing algorithm modules so that output from an algorithm is used as input for another algorithm. Hence, JGAP can be viewed as a visual graph calculator for helping experiment with and teach graph algorithm design. Copyright © 2001 John Wiley & Sons, Ltd.

KEY WORDS: graph algorithms; Java; software visualization; web-based software systems

1. MOTIVATION

Building a software system for experimenting and teaching graph algorithms is not an easy task for several reasons. First, graph algorithms often need to represent graphs by some complex data structures for efficiency reasons. Different algorithms often use different graph representations. However, to ease program development, the software system will need a common representation of graphs. Not only must the common representation satisfy various needs of different algorithms, it must do so in an efficient way. Secondly, for experimentation and teaching purposes, the system must provide a graphic user interface (GUI) so that graphs can be concisely displayed and conveniently edited. Last but not least, the system should allow end users, not just the system designers, to add new algorithms, or new

*Correspondence to: Shi-Chun Tsai, Department of Information Management, National Chi-Nan University, 1 University Road, Puli, Nantou 545, Taiwan.
†E-mail: tsai@im.ncnu.edu.tw

Contract/grant sponsor: National Science Council, Taiwan; contract/grant number: NSC 88-2213-E-260-002 and NSC 88-2213-E-001-007

Received 11 May 2000
Revised 6 September 2000
Accepted 20 November 2000

implementations of existing algorithms, to the system with ease. That is, the system architecture must be modular, with the ability to add and update algorithm modules even after the system has been put into use. In the following, we further elaborate the above three requirements.

The system needs a common representation of graphs, or at least a common application program interface (API) for graph operations. A uniform interface helps program graph algorithms. It also helps in the exchange of graphs among the algorithms. However, it is important that basic graph operations (such as edge addition and deletion, node enumeration, etc.) on the common representation are as efficient as those on customized graph representations favored by individual graph algorithms; otherwise, they will unduly affect the overall performance of the algorithms. Intrinsic properties of the graphs (e.g., whether the graphs are dense, regular, or multi-edged, etc.) must also be taken into consideration when selecting a common graph representation.

For users, it is necessary to have a good GUI for generating, editing and displaying graphs. Moreover, users' actions and the produced graphs should be kept in a 'history list' so that they can be re-used. They will be used to re-run experiments with new graphs or new algorithms. The system should also provide a performance meter to measure the performance of algorithms, and to present the results in a visual way. Ideally, the software system, including its GUI, should be readily portable to multiple computing systems. This often requires the system being implemented in a standard programming language, with a portable library and execution environment. Java, with its rich class library and ubiquitous virtual machine, is very suitable for such a task.

As mentioned above, a common graph representation as well as a good visualization library helps users to program new algorithms. To allow users to incorporate new algorithms into the system, however, one needs additional effort in storing and managing the newly developed code. A popular choice is to use a web-based system, where a web server acts as code repository and new code can be submitted to the web server via a web browser. After the system is updated, a user then re-loads the front-end Java applet to bring all the currently available algorithm modules to the local browser for execution.

We describe in this paper JGAP, a Java-coded web-based platform for experimenting and teaching graph algorithms designs. The platform contains a library of common data structures for implementing graph algorithms, is equipped with a visual graph editor for generating and modifying graphs of specific properties, and features a 'plug-and-play' modular design for adding new algorithms.

This paper is organized as follows. We first survey related works in Section 2. Section 3 outlines the modular architecture of JGAP. Section 4 describes in detail JGAP's common data structures, including their interfaces, for graph algorithms. We also consider the advantage and disadvantage of using a fixed set of common data structures for graphs. Section 5 shows how JGAP is used as a visual graph calculator and how it helps test and teach graph algorithm designs. Section 6 concludes the paper.

2. RELATED WORKS

We classify related tools and systems for graph algorithms into three categories: Java-coded algorithm animation programs, library and datasets for graph algorithm designs and experimentations, and web-based systems for collaborative research. In the following, we list several representative systems in each of the three categories and compare them with JGAP.

Java-coded algorithm animation. Systems in this category use script languages to control animation actions and include a Java applet to interpret user scripts for display in a web browser. These systems can be used to animate general algorithms and data structures, not just graph algorithms. Representative systems include *JSamba* [1], a Java version of the *Samba* animation system [2] and *JAWAA* [3,4]. For general techniques and various systems for visualization of software systems, see [5].

Graph library and datasets. There are many libraries for drawing graphs and implementing graph algorithms. There are also graph datasets for teaching or benchmarking purposes. We name just a few of the most well-known. *GraphViz* is a set of graph drawing tools developed at AT&T Labs [6,7]. It has a graph description language called *dot* and it includes *Grappa*, a Java-coded applet that interprets and draws graphs that are described in dot. *Stanford GraphBase* contains a set of graph data, a graph library, and a collection of C programs coded in a literate programming style [8]. *LEDA* is a C++ library of efficient data structures and algorithms for combinatorial and geometric computing [9].

Web-based collaborative systems. The web infrastructure, with Java-enabled code mobility, has been advocated as a general platform to deploy distributed software systems. In a web-based collaborative system, a web server is used to store and exchange research results among the researchers, who can use browsers to retrieve and update research information in a dynamic manner coordinated by the web server [10]. For the geometric computing domain, a system called *GeoJava* has been built to allow user to execute, even develop, geometric algorithms in a distributed environment [11]. Geometric algorithms are compiled and executed using the LEDA library at the server site, while the results are visualized at the client sites using Java applets. The system also coordinates multiple users to work on an algorithm at the same time.

JGAP differs from JSamba and JAWAA in that it does not use a separate script language to perform animation. Rather, in addition to providing a common data structure for general graph operations, JGAP includes a Java library for on-line editing and displaying graphs. It also offers a window-based GUI to help user manipulate graphs. Because the graph library in JGAP is coded in Java, graph algorithms developed with JGAP have better code mobility than algorithms developed with GraphViz, GraphBase and LEDA, which are coded in C/C++. Both JGAP and GeoJava need a web server to distribute the developed graph programs and their documentations. However, JGAP is a lightweight system in which both graph computation and visualization occur at the user site inside a web browser. This is contrary to GeoJava, where computation is performed and coordinated at the web server, while visualization occurs at the user site. On the other hand, when compared to GeoJava, right now JGAP is not a fully collaborative system. For example, addition of new functionalities to JGAP itself still needs manual coordination.

There is also a difference between JGAP—whose problem domain is graph algorithms—and other Java software tools that use graph representations just to manipulate objects in other problem domains. In this category of software tools, often the emphasis is on a user-friendly graphic interface for manipulating domain-specific objects on screen. The graphic interface may contain little algorithmic content, and the objects being manipulated are not necessarily graphs. A UML editor that models software processes in a visual way, for example, may be considered to be in this category.

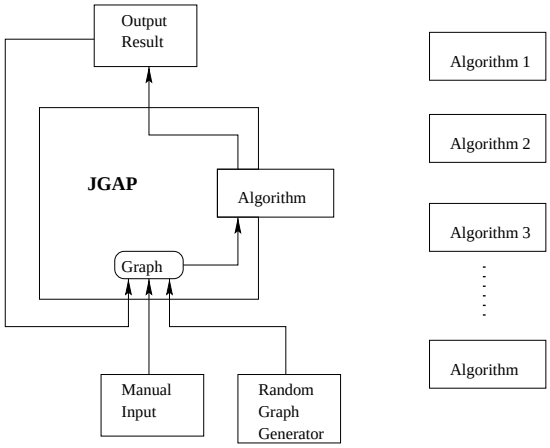

Figure 1. An overview of the JGAP software architecture.

3. SOFTWARE ARCHITECTURE OF JGAP

JGAP consists mainly of two components: a graph editing component and a graph algorithm component. The software architecture of JGAP is illustrated in Figure 1. It shows a *Graph* editing component that edits and visualizes graphs from manual input or from a random graph generator. The graphs are passed to an *Algorithm* component to produce results, which are also graphs and can be fed back to the graph editing component for further modification and visualization. The graph algorithm component can apply one of several algorithm modules to a graph. The result can be passed to yet another algorithm module, and so on. All these are achieved via direct manipulation on JGAP's graphic user interface. This results in a highly interactive user experience. It is like using a calculator, except that the objects of calculation are graphs instead of numbers.

The graph editing component implements the following functionalities.

- Visualization and on-screen modification of graphs. Users can add/delete both nodes and edges to/from graphs, and visualize the results.
- Graph input/output from/to external storage.
- Random-graph generation. Parameters to the generator include number of nodes, edge density, edge weight distribution, whether the graph is directed or undirected, and whether self-loop cycles are allowed.
- History list maintenance. The history list keeps a list of all graphs that have been generated or computed so far. Graphs in the list can be retrieved for future computation and comparison.

The graph algorithm component provides a library of common graph representations, upon which a variety of graph algorithms are implemented. Details of the common graph representations and their interfaces are described in Section 4. This component also includes several standard graph algorithms,

so that JGAP can be readily for use as a calculator. An important part of the graph algorithm component is the performance meter. When applied to an algorithm module, the performance meter measures the algorithm's execution time on a series of random graphs of increasing sizes, and plots its asymptotic performance. Section 5 describes how to use the performance meter as well as other JGAP features.

4. COMMON REPRESENTATIONS OF GRAPHS IN JGAP

Graph representations play a central role in the design and implementation of JGAP. Various representations of graphs (e.g. adjacent matrix, edge list, etc.) have been used in graph algorithms for efficiency reasons. In JGAP, we use a uniform representation for graphs, where the representation can be used as a matrix and a list. Thus it is easy in JGAP to pass around a graph object among different algorithms, where each may demand a different graph representation. Our graph representation is also designed to maintain the time efficiency of both adjacent matrix and edge list, at the cost of extra space overhead and initialization time. If a graph algorithm uses the common representation only as a list, the algorithm's asymptotic performance will be as good as a list algorithm. This is also true for algorithms that use the common representation only as a matrix.

In the following subsections, we provide details of the data types used in JGAP. The definitions of Java classes *Vertex* and *Edge* are self-evident. The definition of *Graph* uses classes *ListArray* and *ListArray2*, which are described in separate subsections. For each class, we list both its data fields and applicable methods. We also present the time and space complexity of the implementations.

4.1. Data types for graphs

A graph $G = (V, E)$ consists of a vertex set V and an edge set E. G can either be directed or not. Graphs, vertices, and edges are all objects. We list in Figure 2 the class declarations for vertices, edges, and graphs. Table I lists the methods that are used to access variables of types *Vertex*, *Edge*, and *Graph*.

4.2. ListArray

In a Graph object, a ListArray object is used to store vertices and a ListArray2 object is used to store edges. ListArray implements both the functionality of doubly linked list and array. ListArray2 is a two-dimensional version of ListArray. For an illustration of ListArray, see Figure 3.

Data stored in a ListArray object can be accessed either as a linked list or as an array. We call the linked list part of ListArray *ListItem*. Each ListItem stores an item in the list. The array part of ListArray is simply an array of ListItem. Array access methods of ListArray are supported via this part of the data structure. The class declarations of ListItem and ListArray are shown in Figure 4. Each item in a ListArray object can be accessed by a specific index, as in an array. It can be accessed like a doubly linked list as well, which is useful, for example, when enumerating all items. Applicable methods of ListArray are described in the next subsection.

4.2.1. Methods of ListArray

Table II shows the interfaces of the methods for accessing a ListArray object. Each array access operation takes a constant time. Also, as in doubly linked list, it takes only $O(1)$ time to perform


```
class Vertex {
  int identity;    // Identity of this vertex.
  int predecessorNode;    // The predecessor of this vertex.
  int x;           // The x position of this vertex.
  int y;           // The y position of this vertex.
  Color color;     // Status of this vertex. Use color representation.
  int hopDist;     // Hop distance to this vertex.
  int inDegree;    // In-degree of the vertex.
  int outDegree;   // Out-degree of the vertex.
  double distance; // Distance to this vertex.

class Edge {
  int fromNode;    // Indicate the node that this edge begins with.
  int toNode;      // Indicate the node that this edge ends at.
  double weight;   // The weight or the capacity of the edge.
  double flow;     // The flow on this edge.
  Color color;     // Status of this edge. Use color representation.

class Graph {
  protected int vertexSerialNo;   // Indicate the number of vertices.
  ListArray vertexList;   // Store all vertices.
  ListArray2 edgeList;    // Store all edges.
  boolean directed;       // Indicate whether the graph is directed.
}
```

Figure 2. Class declarations for vertices, edges, and graphs.

an insert or delete operation in ListArray. To insert an item to a ListArray object, first we prepare a ListItem object and place it at the specified index of ListArray. We then link it up to its neighbors in the ListArray object. Thus, we can add one element in $O(1)$ time. To delete an item from a ListArray object, we just remove the corresponding links as in the case of doubly-linked list. We then remove the item at the corresponding index in the ListArray object. The time for a delete operation is also constant.

4.3. ListArray2

ListArray2 is ListArray with a two-dimensional support. Figure 5 shows an example of ListArray2. There are three pairs of (*prev*, *next*) pointers for each item in ListArray2. Horizontal pointers (*prevX*, *nextX*) can be used to enumerate all elements in a row. It is useful when retrieving all out-going edges of a vertex. The vertical pointers (*prevY*, *nextY*) can be used to enumerate all elements in a column, which, again, is useful for retrieving all in-coming edges of a vertex. The third pair of

Table I. Methods in classes Vertex, Edge, and Graph: variables v, e, and G are of classes Vertex, Edge and Graph, respectively.

v.getInDegree()	get in-degree of vertex v.
v.getOutDegreee()	get out-degree of vertex v.
v.getColor()	get color of vertex v.
v.setColor()	set color of vertex v.
e.getFromNode()	get the starting node of edge e.
e.getToNode()	get the ending node of edge e.
e.getColor()	get color of edge e.
e.setColor()	set color of edge e.
G.getNumOfVertices()	get the number of vertices in graph G.
G.getNumOfEdges()	get the number of edges in graph G.
G.verticesElements()	enumerate all vertices in graph G.
G.allEdgesElements()	enumerate all edges in graph G.
G.addVertex(v)	add vertex v to graph G.
G.deleteVertex(v)	delete vertex v from graph G.
G.addEdge(e)	add edge e to graph G.
G.deleteEdge(e)	delete edge e of graph G.
G.directedAdjVerticesElements(v)	enumerate vertices that are adjacent to vertex v in G.
G.undirectedAdjVerticesElements(v)	enumerate vertices that are adjacent to vertex v in G, ignoring the direction of edges.
G.isEdge(u,v)	whether an edge exists at specified ends in G.
G.inEdgesElements(v)	enumerate all incoming edges to vertex v in G.
G.outEdgesElements(v)	enumerate all outgoing edges from vertex v in G.

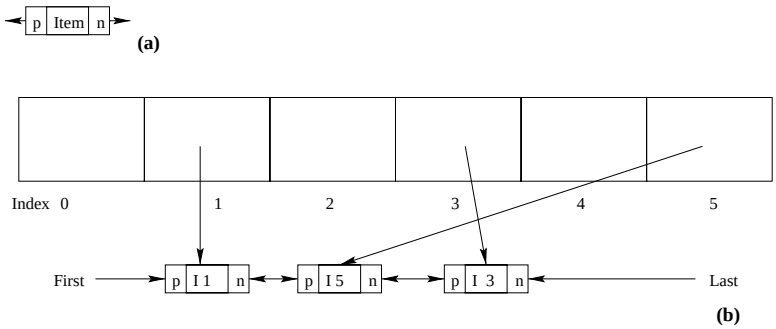

Figure 3. A ListArray structure with three list items and an array of five entries. All non-null array elements appear in the list.

```
public class ListItem implements Cloneable, Serializable {
  public static final int NUL = -1;  // Used to indicate a null item.
  private Object obj;      // The object is stored here.
  private int dimension;   // The dimension of the ListArray.
  private int coord[];     // Coordinate of this item.
  private ListItem prev[]; // Previous items.
  private ListItem next[]; // Next items.
  boolean link;            // Indicate whether this item is a link.
}

public class ListArray implements Cloneable, Serializable{
  protected int capacity;   // The maximum capacity of the ListArray.
  protected int numOfItem;  // The number of items in the ListArray.
  private int maxItemIndex; // Maximum index that is currently used.
  protected ListItem itemArray[]; // An array of items.
  protected ListItem first=null;  // The first item of this ListArray.
  protected ListItem last=null;   // The last item of this ListArray.
 }
```

Figure 4. Class declarations of ListItem and ListArray.

Table II. Methods to access a ListArray object. The first three methods treat the object as an array, and the rest treat it as a list.

Object itemAt(x)	Get the item at the specified index x.
void setItemAt(o, x)	Store an item o at index x.
boolean itemExistAt(x)	Check whether an item exists at index x.
ListItem getPrev(ListItem *li*)	Get the previous item of *li*.
int getPrevIndex(int i)	Get the index of previous item.
ListItem getNext(ListItem *li*)	Get the next item of *li*.
int getNextIndex(int i)	Get the index of next item.
ListItem getFirst(); int getFirstIndex()	Get the first item; get its index.
ListItem getLast(); int getLastIndex()	Get the last item; get its index.
Object elements(), nextElement(), hasMoreElement()	Extract items one by one, via the Java java.lang.Enumeration interface [12].

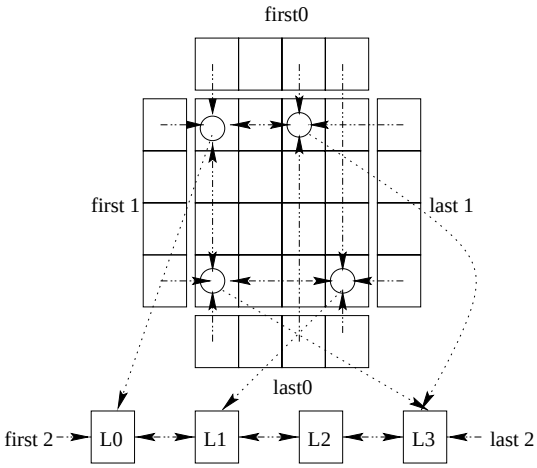

Figure 5. A ListArray2 structure with four list items and a 4×4 array. All non-null array elements appear in the list and are interlinked in both dimensions.

pointers (*prevZ*, *nextZ*) is used to enumerate all edges in a graph. Methods and their costs of inserting and deleting items in ListArray2 are similar to those in ListArray.

4.4. Adjacency ListArray

There are many ways to represent a graph. For example, a graph can be represented by an edge list, an adjacency list, or an adjacency matrix [13,14]. The edge list and the adjacency list representations have the advantage of efficient enumeration of adjacency edges with economical usage of storage space. But for both of the representations, it takes more time to find out whether or not there exists an edge adjacent at a specific vertex. The adjacency matrix representation can perform this operation in $O(1)$ time, but it is less efficient in edge enumeration. The adjacency matrix also consumes more space than the edge list and the adjacency list.

We have designed a data structure called *Adjacency ListArray* to represent a graph. It provides the functionality of both adjacency list and adjacency matrix. For each operation it supports and maintains the minimum time complexity as required by adjacency list and adjacency matrix. However, it incurs extra space overhead compared to that required for the adjacency list and adjacency matrix. The usage of a common graph representation like Adjacency ListArray is crucial in the design of JGAP, as it eliminates the need of representation conversion between different graph algorithms, while still maintaining the asymptotic performance of each algorithm.

Figure 6 shows an illustration of an Adjacency ListArray representation for a graph of five vertices and seven edges. Part (a) shows the directed graph. Part (b) shows the data structure for a vertex, where *in* points to the incoming edges to the vertex, and *out* points to the outgoing edges from the vertex.

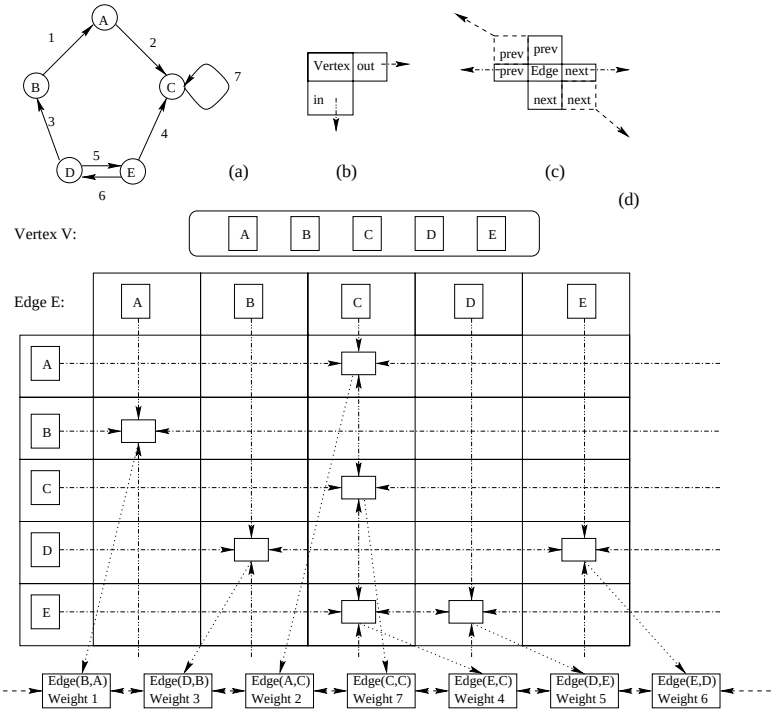

Figure 6. An Adjacency ListArray structure for a graph of 5 vertices and 7 edges.

Part (c) shows the data structure for an edge, where the horizontal *prev/next* pointers link up edges of the same source vertex, the vertical *prev/next* pointers link up edges of the same destination vertex, and the diagonal *prev/next* pointers link up all edges in the graph. Part (d) is the Adjacency ListArray representation of the graph. As is clearly shown in the illustration, the representation makes use of the ListArray2 to maintain two collections of (orthogonal) list structures in a two dimensional array. The third list structure, the edge list, is also maintained by the edge items in the ListArray2 data structure via their diagonal *prev/next* pointers.

Table III summarizes the time complexity of graph methods for various graph representations. The graph methods are already described in Table I. In Table III, *ideg* indicates the in-degree of a vertex, *odeg* indicates the out-degree of a vertex, and *deg* can be ideg or odeg. As shown in the table, for each graph method, the cost of Adjacency ListArray is the minimum of all graph representations. On the other hand, the space complexity of Adjacency ListArray is $O(n^2)$.

Table III. Space and time complexity of four graph representations: n is the number of vertices and m the number of edges.

Graph methods	Graph representations			
	Edge List	Adj. List	Adj. Mat	Adj. ListArray
getNumOfVertices	$O(1)$	$O(1)$	$O(1)$	$O(1)$
getNumOfEdges	$O(1)$	$O(1)$	$O(1)$	$O(1)$
verticesElements	$O(n)$	$O(n)$	$O(n)$	$O(n)$
allEdgesElements	$O(m)$	$O(m)$	$O(n^2)$	$O(m)$
addVertex	$O(1)$	$O(1)$	$O(1)$	$O(1)$
deleteVertex	$O(m)$	$O(odeg(v) + ideg(v))$	$O(n)$	$O(odeg(v) + ideg(v))$
addEdge	$O(1)$	$O(1)$	$O(1)$	$O(1)$
deleteEdge	$O(1)$	$O(1)$	$O(1)$	$O(1)$
getInDegree	$O(1)$	$O(1)$	$O(1)$	$O(1)$
getOutDegree	$O(1)$	$O(1)$	$O(1)$	$O(1)$
undirected-AdjVertices-Elements(v)	$O(m)$	$O(deg(v))$	$O(n)$	$O(deg(v))$
directed-AdjVertices-Elements(v)	$O(m)$	$O(odeg(v) + ideg(v))$	$O(n)$	$O(odeg(v) + ideg(v))$
isEdge(u,v)	$O(m)$	$O(min(odeg(u), ideg(v)))$	$O(1)$	$O(1)$
inEdgesElements	$O(m)$	$O(ideg(v))$	$O(n)$	$O(ideg(v))$
outEdgesElements	$O(m)$	$O(odeg(v))$	$O(n)$	$O(odeg(v))$
getFromNode	$O(1)$	$O(1)$	$O(1)$	$O(1)$
getToNode	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Space required	$O(n + m)$	$O(n + m)$	$O(n^2)$	$O(n^2)$

5. JGAP: A VISUAL GRAPH CALCULATOR

For arithmetic, people frequently use calculators to assist numerical computation. For graph problems, it is very useful to have a similar tool to help users to deal with graph computation in a visual way. JGAP provides tools to help users to generate graph instances that serve as input to graph algorithms. JGAP also has a GUI to visualize and directly manipulate graphs. JGAP currently provides graph algorithms for breadth-first search, depth-first search, spanning tree, shortest paths and maximum flow. Users can also implement and add new graph algorithms to JGAP. Consequently, one can view JGAP as a visual graph calculator.

The following outlines a typical scenario of using JGAP as a visual graph calculator. A graph is first generated, manually or automatically, and used as input to an algorithm. The result is then displayed in a separate window, and can be used as input for yet another algorithm. A history of results is kept by JGAP so that they can be referenced later on. Graphs can also be directly edited between computations. The supported editing functions include graph union, graph difference, and addition and deletion of vertex and edge. All editing operations are initiated by the user's direct input, such as mouse-button

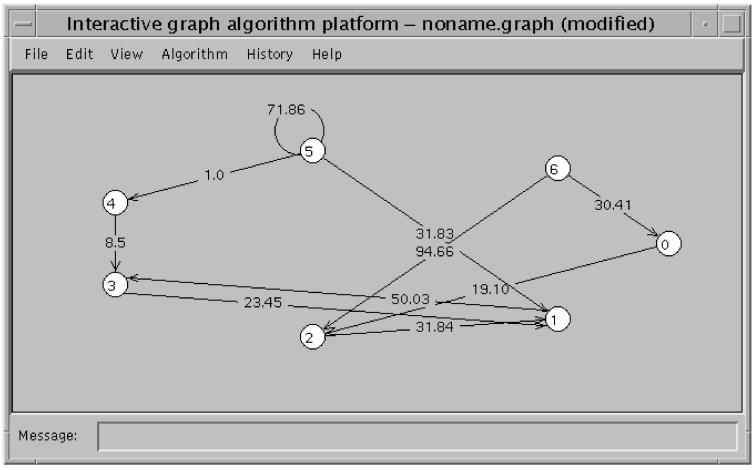

Figure 7. The JGAP main window.

clicking and keyboard input. With JGAP, one can test and evaluate a graph algorithm quickly; it can also be used to verify properties of input graphs. Since JGAP is coded in Java, it can be loaded from the Web and executed inside a browser. It therefore provides users with a friendly platform for designing and learning graph algorithms.

The rest of this section gives an overview of the JGAP graphic user interface, and briefly describes how to add a new algorithm module to JGAP.

5.1. The JGAP main window

JGAP starts when its applet is executed inside a web browser. A main window will pop up with a title: *Interactive graph algorithm platform*. Below the title is the menu bar with the following menu items: *File*, *Edit*, *View*, *Algorithm*, *History* and *Help*. Each of these items contains additional menu items, a some of which can be further extended by users by the addition of new algorithm modules. For example, Figure 7 shows the JGAP main window which displays a small graph ready for calculation.

File menu. Users can select the *New* item under the menu to manually create a graph or to generate a random graph. One can also select the *Open* item to load an existing graph saved from a previous session. The graphs can be directed or undirected, weighted or unit-weighted. They are generated by selecting appropriate sub-menu items. Users can select the *Save* item to store a graph. Because of the security restriction imposed on earlier versions of Java applets and browsers, the *Open* and *Save* functions are not available in the current applet version. A future extension of JGAP is to use Java RMI (Remote Method Invocation) or HTTP (Hypertext Transfer Protocol) to save/restore

graphs to/from the web server, from which the applet is loaded. This may incur additional system overhead for user administration.

Edit menu. This menu contains menu items to edit a graph. There are four groups of commands. The first group is for editing edges, from which one can add edges, delete edges and change edge weight. The second group is used to delete and rename vertices. Items in the third group are for deleting 'special edges', which are generated after executing a graph algorithm. For example, when we select DFS (depth first search) under the *Algorithm* menu for a graph, the output can produce *discovery, back, cross* and *forward* edges to indicate how the search is done [13]. We specify special edges in the output window with different colors and we may use the output as input for the next algorithm. During the selection, we can keep or discard the special edges, since these edges are mostly for visual purposes. Once the special edges are kept, we can edit them as regular edges. The last group of items are used to make a union or a difference of two graphs. We can prepare the first operand from *File* menu or *History* menu and select one of these two operations. Then a dialogue box will pop up and ask for the second input graph, which can be selected from the history menu only. The results of these two operations are shown in the main window and cloned in the history list. These two graph operations are very useful for testing and developing graph algorithms.

View menu. This menu allows one to view different representations of the same graph, (i.e., visual representation, adjacent matrix representation, adjacent list) by selecting the corresponding menu items.

Algorithm menu. JGAP includes many commonly used graph algorithms: depth-first search, breadth-first search, Prim's algorithm for minimum spanning tree, Kruskal's algorithm for minimum spanning tree, Dijkstra's algorithm for shortest path, Bellman–Ford's algorithm for shortest path, Floyd–Warshall's algorithm for all pair shortest path, and Ford–Fulkerson's maximum flow algorithm. The listed algorithms may not be the most efficient ones for the problems they solve; they are included so that JGAP can be readily used as a platform for graph computation. Users can also execute their own algorithms by selecting the *Custom* item, which will load their Java class library into JGAP. Again, because of security issues, this function is only available in the application version of JGAP, but not in the applet version.

The algorithms included in JGAP are implemented faithfully so that their asymptotic performance is fully reflected, and can be observed by a performance meter for evaluational and educational purposes. Graph output from an algorithm execution is shown on a new window, which can be used as input to another algorithm. All outputs are kept in a list, and can be accessed from the *History* menu.

History menu. Under this menu, JGAP keeps a list of graphs that have been generated or computed so far. If one needs to access a graph from this list, one just goes through the history list and selects it. This way, we can compose the effect of several graph algorithms. The graphs are stored in a vector (of type *java.util.Vector*), and can be easily identified by its index in the vector. For the current version, each graph is about 100 kB, and the history list can handle up to 50 graphs without difficulty. If users need to handle more graphs at a time, the virtual memory of the host operating system will take care of the space problem.

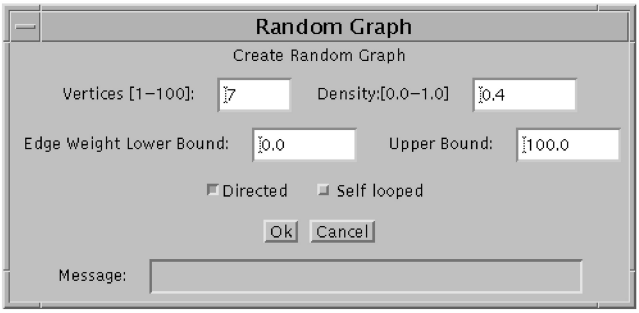

Figure 8. The dialog box to generate a random graph with specified edge density and weight range.

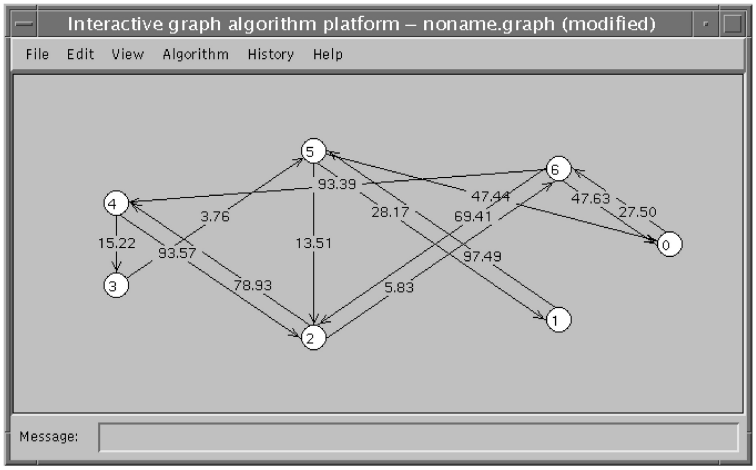

Figure 9. The generated random graph.

Help menu. This menu provides basic guidelines on how to use the system. When selected, a window pops up showing the content of a readme file.

We use the following sequence of screen shots to show how JGAP is used to perform, visually, a sequence of graph calculations. Figure 8 shows the dialog box which one uses to generate a random graph for input. As shown, the dialog box is configured to generate a directed graph of seven vertices whose edge density is 0.4 and whose edge weight is in the range of 0 to 100. The graph is instructed to contain no self-loop. The generated graph is shown in Figure 9.

Figure 10. The dialog box for entering the starting vertex of a breadth-first search.

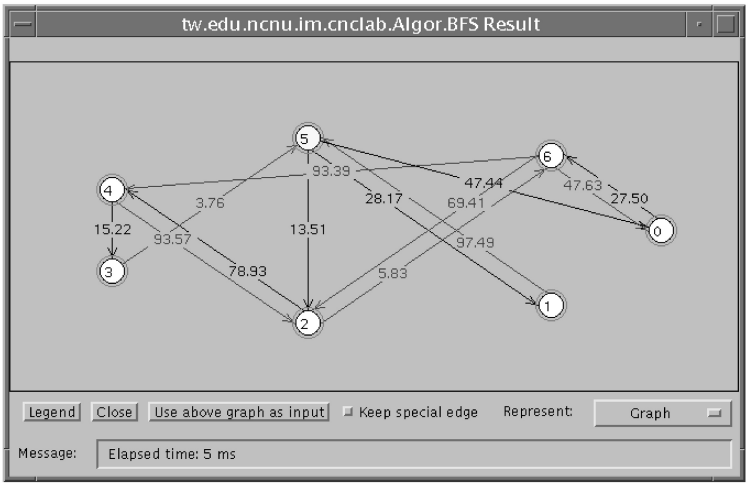

Figure 11. Result of the breadth-first search. The traversed edges are (5, 2), (2, 4), (4, 3), (5, 1), (5, 0), (0, 6). They are highlighted on the screen.

To perform a breadth-first search on the generated graph, one just selects BFS (breadth first search) under the Algorithm menu of the main window. A dialog box, as in Figure 10, will pop up asking for the starting vertex of the search. The result of the search is shown in Figure 11, where the breadth-first search tree is shown with other edges. One can use the result as input to another graph algorithm by clicking the button *Use the above graph as input*. Figure 12 shows the result, which contains only the breadth-first search tree. Then, as an example, one can supply this graph as an input to the Ford–Fulkerson maximum flow algorithm, by selecting the *Ford–Fulkerson* item under the Algorithm menu. Again, a dialog box pops up, as shown in Figure 13, asking for input from users. The result is shown in Figure 14.

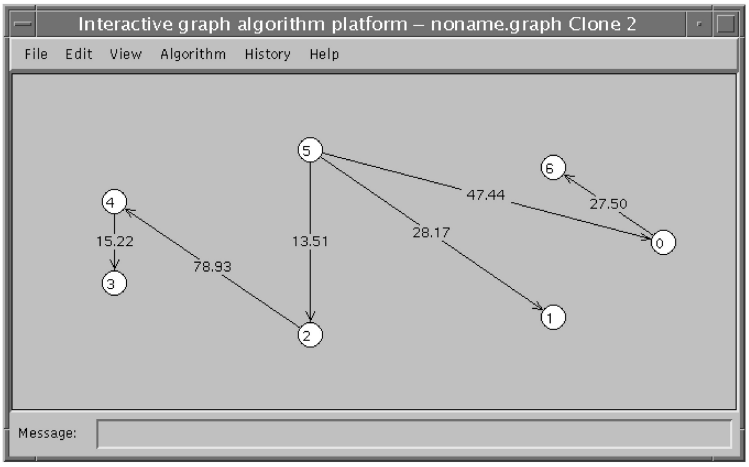

Figure 12. Result of the breadth-first search, only the traversed edges are shown.

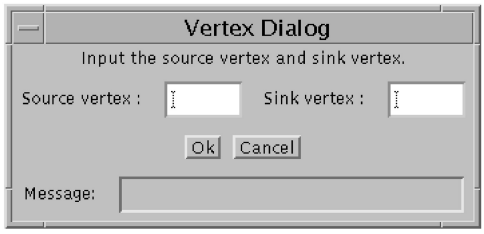

Figure 13. Entering the source and sink vertices for the maximum flow algorithm.

5.2. Performance meter

JGAP includes a performance meter, the concept of which is illustrated in Figure 15. When applied to an algorithm module, JGAP performance meter will estimate its asymptotic performance. The *Performance Meter* item is in the Algorithm menu. When selected, a dialog box pops up as in Figure 16, where users can specify the algorithm to be measured, the graph size range, the edge weight range, the edge density, and other graph properties. More precisely, the first two fields after the *Vertices Start* label specify the range of graph size, and the third field specifies the increment of graph size in each iteration. Graphs of the same number of vertices can have different edge density. With each specific vertex number and edge density, JGAP repeatedly and randomly generates graphs, measures

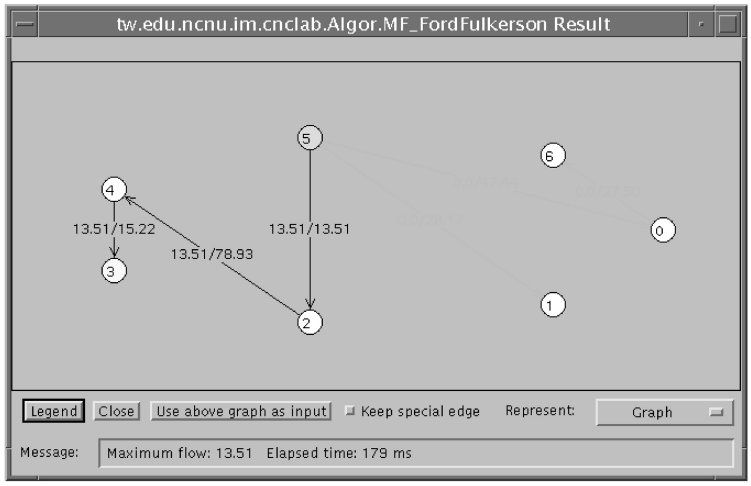

Figure 14. Result of the Ford–Fulkerson maximum flow algorithm. The graph from Figure 12 serves as input, where vertex 5 and vertex 3 are the source and the sink.

the execution time as many times as indicated in the *Retry* field, and calculates the average execution time for this graph size. Edge density starts with the number specified in the field of *Density step*. The density increases by the same number for subsequent iterations until it reaches 1. By sampling the execution time for graphs of different sizes and edge density, JGAP gives out realistic information on the efficiency of the tested algorithm. The result is shown in a window, where users can select to show the result in line chart or raw data format. In Figure 16, the meter is set up to measure the performance of breadth-first search, and in Figure 17 the performance results are displayed using a line chart by the meter. Different algorithms for solving the same graph problem can be metered, and their performance compared. One can also use the performance meter to evaluate an algorithm on different types of graphs, and use it to help design specialized but more efficient algorithms for those graphs. Also, for any user-defined graph, the execution time for the graph is already available in the message field of JGAP's main window.

5.3. Adding algorithm modules to JGAP

New algorithm modules can be added to JGAP. Not only can algorithm designers use JGAP to develop new algorithms, they can also combine and extend existing algorithm modules in JGAP with new modules. An algorithm module is developed by inheriting the *Algorithm* class. It is provided in JGAP to encapsulate routine and tedious implementation details, so that algorithm designers can spend most of their time on the intrinsics of their algorithms, instead of on GUI and other system issues.

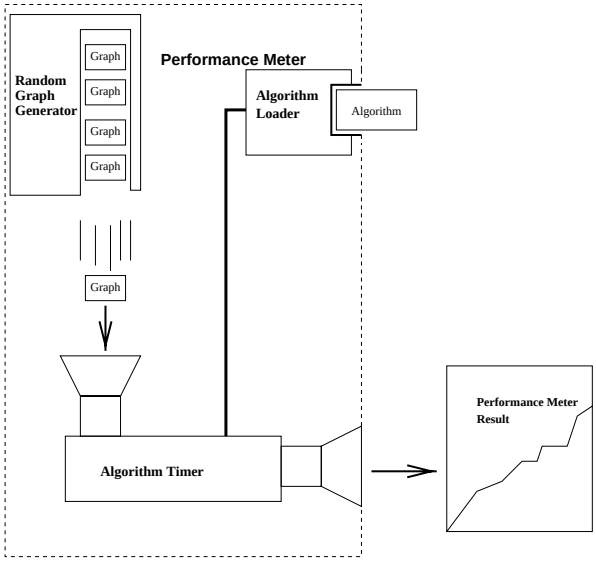

Figure 15. Concept of the JGAP performance meter.

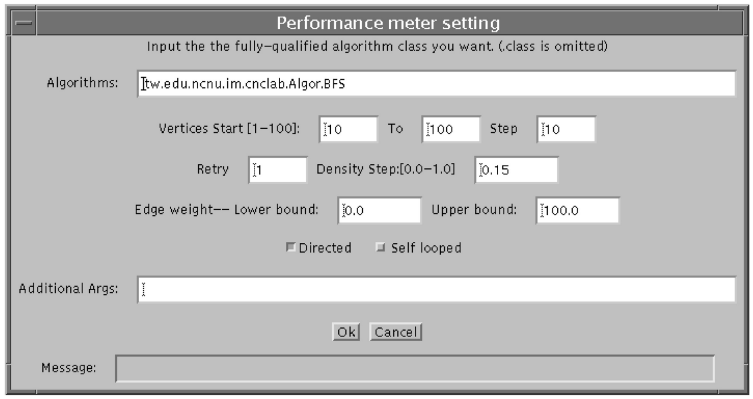

Figure 16. Set-up of the performance meter for breadth-first search.

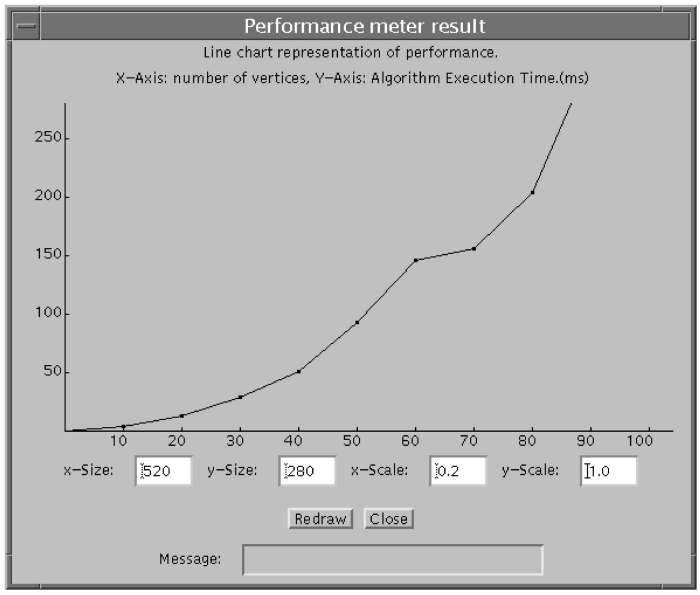

Figure 17. Output of the performance meter for breadth-first search.

In the following, we use an example to illustrate how to implement a new algorithm module for JGAP. Suppose we want to design a module *FindPath* to find a path from vertex u to vertex v in a graph. *FindPath* can be added to JGAP by the following steps.

Inheriting from class *Algorithm*. Make class *FindPath* inherit from class *Algorithm*. In class *FindPath*, one needs to define a constructor so that a problem instance can be created. One also needs to describe how the output graph will be shown. As the path is described by a sequence of connecting edges, one will need to define the colors of the edges so that the path will stand out clearly. The *Algorithm* class defines two methods, *show* and *repaint*, of painting the graph in a window before it is used by the *FindPath* algorithm, and after it has been altered by the *FindPath* algorithm. The graph is an data member in class *Algorithm*, and must be used in the abstract method *algorImpl* (which actually implements *FindPath*, see the next item) to reflect change to the graph as a result of applying the algorithm.

Implementing method *algorImpl*. The *algorImpl* method, which *FindPath* inherits from *Algorithm*, is where one puts the actual implementation of the algorithm. It is defined as an abstract method, and left undefined in *Algorithm*. Method *algorImpl* will return an instance of class *Object*—which stores an instance of graphs, vertices, edges, Boolean values, or of whatever class—that will be returned as the result of the algorithm. It is the user's responsibility to ensure that the returned object will meet the specifications of all the methods that will take it as an argument.

Note that the user need not define a visualization method for the returned object, as visualization of the input graph, as well as any change to it, has been provided by the *show* and *repaint* methods described in the previous item.

Setting the parameters. Method *algorImpl* does not accept parameters. Instead, method *setArg* must be explicitly defined to accept an array of parameters from the users. Since *FindPath* needs two parameters, the starting and ending vertices, method *setArg* must recognize that the two integers in the input parameter array are the indices of the two vertices. If one wants to use the *Additional argument* item under the *Custom Algorithm* menu, method *setArg* must be defined to recognize string parameters which will represent customized input. Method *setArg* must accept the null parameter array as well, as it indicates the default input to the algorithm.

One can also select predefined data structures and algorithms included in the JGAP package to help program one's algorithm. For instance, one may use binary heaps or pair heaps (which both are available in JGAP) to implement a priority queue. In order to use these predefined data structures, the user must implement method *setArg* so that they are explicitly selected and passed to the algorithm.

Selecting a preferred dialog. JGAP provides several classes for user/system dialogs, and one can choose the dialog box that best fits the need of the algorithm. For example, *FindPath* needs two vertices as input, so one can use class *JGAP.UI.Ask2VertexDialog*, which is designed for this purpose. Algorithm designers may override the *getPreferedDialog* method to use a customized dialog for their algorithms.

Adding modules to main menu. To add a new algorithm to JGAP, one needs to make sure that CLASSPATH is modified accordingly so that implementation of the algorithm can be located by the Java run-time system. To test a new algorithm locally, the developer can select the *Custom* item in the *Algorithm* menu to load the implementation, where one needs to specify the fully qualified class name. For example, if *FindPath* belongs to package *test*, then one needs to type *test.FindPath*.

To incorporate a new algorithm item into the JGAP main window menu so that all JGAP users can access it, however, will need more work. One must modify and recompile class *AlgorithmMenu*, which is in package *JGAP.UI*, to reflect the addition of new items in the algorithm menu. It is not necessary to download the entire JGAP source for the above modification and recompilation process, one just needs to import the packages required by class *AlgorithmMenu*. One can probably use Java's reflection mechanism to provide a more flexible way to add new program modules in a system like JGAP, though we have not used it in the current version.

After all the steps are completed, the *FindPath* module is added to JGAP and is ready for use under the *Algorithm* menu.

6. CONCLUSION

We have presented JGAP, a platform for graph operations and graph algorithm design. The platform contains a library of common data structures for implementing graph algorithms and features a modular

design for adding new algorithm modules. JGAP can be used as a visual graph calculator, hence it helps to experiment with and to teach graph algorithms. The current version of JGAP appears as a Java applet, and can be downloaded for execution in a Web browser. Source code and documentation of JGAP can be found at http://im.ncnu.edu.tw/~tsai/definite/JGAP/JGAP.html.

Future work on JGAP includes user-account management, which is necessary if JGAP is to be used in a multi-user environment for collaborative graph algorithm designs and experimentations. This will involve security and load-sharing issues. Another direction is to add interoperability to JGAP so that it can work with other graph library and tools, and enhance its usability.

REFERENCES

1. JSAMBA—Java version of the SAMBA Animation Program. http://www.cc.gatech.edu/gvu/softviz/algoanim/jsamba/.
2. Stasko JT. Using student-built algorithm animations as learning aids. *Technical Report GIT-GVU-96-19*, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, 1996.
3. JAWAA—Java and web based algorithm animation. http://www.cs.duke.edu/~wcp/JAWAA.html.
4. Pierson WC, Rodger SH. Web-based animation of data structures using JAWAA. *29th SIGCSE Technical Symposium on Computer Science Education*, 1998.
5. Stasko JT, Domingue JB, Brown MH, Price BA (eds). *Software Visualization*. MIT Press, 1998.
6. Graphviz — Graph Drawing Software. http://www.research.att.com/sw/tools/graphviz/.
7. Gansner E, North S. An open graph visualization system and its applications to software engineering. *Software—Practice and Experience* 2000; **30**(11):1203–1233.
8. Knuth DE. *The Stanford GraphBase: A Platform for Combinatorial Computing.* Addison-Wesley: Reading, MA, 1993.
9. Mehlhorn K, Naher S. *Leda: A Platform for Combinatorial and Geometric Computing.* Cambridge University Press, 2000.
10. Beca L, Cheng G, Fox G, Jurga T, Olszewski K, Podgorny M, Sokolowski P, Walczak K. Java enabling collaborative education, health care, and computing. *Concurrency: Practice and Experience* 1997; **9**(6):521–533.
11. Aoki KF, Lee DT. Towards Web-based computing. *International Journal of Computational Geometry and Applications*, to appear.
12. Sun Microsystems Inc. Java Platform 1.2 API Specification. http://java.sun.com/products/jdk/1.2/docs/api/.
13. Cormen TH, Leiserson CE, Rivest RL. *Introduction To Algorithms* (2nd edn). McGraw-Hill, 1990.
14. Goodrich MT, Tamassia R. *Data Structures and Algorithms in Java*. Wiley, 1998.

