VisualGraph – A Graph Class Designed for Both Undergraduate Students and Educators

Jeff Lucas
U Wis. Oshkosh, USA
lucasj13@uwosh.edu

Thomas L. Naps
U Wis. Oshkosh, USA
naps@uwosh.edu

Guido Rößling
Darmstadt U Techn., Germany
roessling@acm.org

Abstract

Graphs and graph algorithms play an important role in undergraduate data structures and algorithms courses. However, they often also represent the first case where both the correctness and the underlying concepts of the algorithms are not evident. Both students and educators can therefore benefit from a simple yet expressive tool for coding graph algorithms and then conveniently visualizing them. We present such a tool, derived from a set of instructional requirements, and give an example application.

Categories & Subject Descriptors

K.3.2 *Computers & Education*: Computer & Information Science Education - Computer Science Education

General Terms

Algorithms

Keywords

Visualization, Animation, Pedagogy, CS 1, Graphs

1 Introduction

Graphs and graph algorithms are one of the more advanced topics typically covered in an undergraduate course on data structures and algorithms. Early in such a course, students will have been exposed to programming with pointers and recursion, usually by coding lists and trees. Thus, the focus of teaching graph algorithms often lies less on learning new programming techniques than on solving given problems.

The additional challenge of graph algorithms is that the typical algorithms are based on concepts that are deeper than students have previously studied. Even subtle aspects of advanced algorithms that students have encountered before – such as Quicksort partitioning – usually have a concept which can be explained within one or two sentences and is still relatively easy to understand. However, this is not true for many graph algorithms. Indeed, graph algorithms may be the first type of algorithms encountered by students that is not "obviously correct". Consider for example the order of the loops for Floyd's all-pairs shortest path algorithm, where the outer loop concerns the *intermediate* path vertex rather than the starting vertex.

Programming assignments that accompany the graph component of the course should allow the students to focus their energies upon understanding the more conceptual techniques rather than struggling with low-level implementation details of a graph data type. Typically, such an implementation is carried out using adjacency lists or adjacency matrices, neither of which will expose the student to anything substantively new from a coding perspective. Nor should the student have to climb an exceedingly complicated learning curve to understand the interface for a graph class library that is oriented to industrial-strength applications. Certainly such libraries have their place in courses that want to emphasize rigorous software engineering methodologies. But, in the course goals emphasized here, having students struggle more than necessary with a class interface only detracts their attention from the problem-solving and analysis techniques upon which they should be focusing.

Based on these premises, Section 2 presents a detailed breakdown of the requirements for the "ideal" graph class library to use in an undergraduate algorithms and data structure course. A set of existing libraries and software packages is evaluated in Section 3. As the regarded packages did not fully match our requirements, we present our own class library in Section 4. Section 5 illustrates how easy using the class library is for students and educators. Finally, Section 6 outlines future directions for research and development.

Permission to make digital or hand copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, require prior specific permission and/or a fee.
SIGCSE 2003, February 19-23, 2003, Reno, Nevada, USA.
Copyright 2003 ACM 1-58113-648-X/03/0002....$5.00

2 Requirements for an Educational Graph Library

2.1 Student Requirements

Basic graph operations

These operations let students implement a wide variety of graph algorithms using operations analogous to what they would find in textbook-style pseudo-code.

Set/query global graph attributes. Users should be able to set and then query two global graph attributes – directedness and weightedness. In a directed graph all edges are directed. Thus, a bidirectional edge is really two edges – one in each direction. If the graph is weighted, all edges must have a numerical weight.

Graph-editing operations. Four graph-changing operations are essential for all graphs – add or remove a vertex, and

add or remove an edge. For weighted graphs, users must also be able to set the weight of an edge to a particular value.

Vertex- and edge-querying operations. Users require an operation that, given two vertices, returns a boolean indicating whether or not an edge exists from one to the other. For weighted graphs, students must also be able to query the weight of an edge. Our focus on making it easy for undergraduates to code algorithms means that the vertex data type can be a simple enumeration such as characters.

Graph generation operations

Once a student has implemented the algorithm, inputting data to both test and experiment with the algorithm is vital. As graphs are a visually oriented data structure, requiring students to input a graph in textual fashion is counterintuitive and awkward. We propose three basic modes for graph generation.

Interactive input. When called, this operation pops up a simple GUI that allows students to establish the two global graph attributes established above (directed/undirected and weighted/unweighted) and then point-and-click their way through the entry of vertices and edges. Because data in vertices are characters, vertices are automatically labeled with the next available character when the student picks a location for the vertex.

Random graph generation. Even with interactive input based on a GUI, entering a graph suitable for a particular problem can be time-consuming. It is therefore advantageous if the graph library itself can also generate random graphs that can then be used to test student-written algorithms. Of course, such generation cannot be completely random but must be subject to constraints determined by the problem being solved. Among these constraints are three shared by all graphs – weighted/unweighted, directed/undirected, and number of vertices or edges. Four additional constraints are then related to the specific type of graph to be generated – connected graphs, directed acyclic graphs, graphs with Hamiltonian cycles, and complete graphs.

Input from and output to a file. Once generated using random constraints or via the GUI, the graph must be writable to a file format that can later be read. This allows both students and educators to conveniently build up a suite of graphs to use as test data for a particular algorithm and to compare two algorithms solving the same problem on identical data sets.

Graph animation operations

Because graph algorithms are inherently visual in nature, it is natural for students to want to view the output of their work as an animation of the algorithm they have implemented. Recent research [1, 10] indicates that the most effective instructional use of algorithm animation may occur when students are responsible for developing code that produces the animation. There should be two modes that enable students to create their own animated algorithms. The first of these should make it nearly effortless to produce an "unadorned" animation in which no use of color or highlighting is used to explain the executing algorithm. In this unadorned mode, the graph class should produce a picture of the current state of the graph each time a "print graph" operation is called.

Creating an unadorned animation should thus be as easy for the student as the process of inserting tracer output to debug a program. The student merely needs to identify key points in the execution of the algorithm and add a call to the "print graph" operation. This operation is a sophisticated one from the perspective of the implementor of the class. This is because the resulting visual display of the graph must be laid out in an aesthetically pleasing way that minimizes unnecessary edge crossings and effectively displays the edge weights for a weighted graph. We discuss some of the details of the layout algorithm used by the *VisualGraph* class in Section 4.

At a level above that of an unadorned animation, we might want to ask students to develop a more visually effective animation that could become part of a larger project for a course. Here the graph class should provide a set of visually oriented operations that again make it relatively painless to produce such an animation. Among these operations are (1) set the color of a vertex or edge, (2) highlight a vertex or edge to distinguish it from other vertices, and (3) add explanatory text at a particular location on the graphics screen.

2.2 Educator Requirements

Since many educators may want to create instructional animations of graph algorithms for their students to explore, it is important that a graph class designed for instructional use offers the potential to develop such "expert-created" animations. What separates such animations from those created by students? Recent research on engagement techniques in algorithm animation [7] indicates that augmentation of the animation with "engagement objects" may serve to direct a student's exploration of the algorithm in a more effective way. Such objects include the use of additional graphic displays to augment the graph itself, hypertext documents that supplement the algorithm's animation with explanations of what the viewer is seeing, and the use of interactive questions that force the student to predict what future states of the animation will reveal. Although the graph library need not include specific engagement objects such as these, the library should encourage and make it relatively easy for educators to stream such materials into the animation.

3 Existing Libraries and Software Packages

Until we developed *VisualGraph*, no graph class libraries combined every aspect described in Section 2. Nonetheless, the *VisualGraph* class was not developed in a vacuum and owes much to a variety of earlier work. A selection of this work includes research in graph generation algorithms [2], graph layout algorithms [3, 4], object-oriented graph libraries [5], and instructional algorithm visualization [6, 9].

The work of Johnsonbaugh and Kalin [2] was presented in the context of a C program that generated random graphs subject to a variety of constraints. The criteria they used to define the graphs produced by the program are unsurpassed in terms of producing data sets suitable to test student-written algorithms and served as the basis by which we built random graph generation into the *VisualGraph* class.

The graph library providing the richest set of operations is certainly LEDA [5]. Apart from providing basic graph operations, it provides the implementations of many classic graph algorithms and a set of calls to graphic functions that are layered on the X Window system. However, although it sets the standard for industrial-strength graph libraries, LEDA entails a very steep learning curve. Its overall complexity is comparable to that of the Standard Template Library in C++. Consequently, unless students are well-versed in LEDA's use from their work with simpler data structures, having them use it for the first time when they begin to study graphs will most certainly shift their focus to language-oriented details instead of the more conceptual algorithm analysis and experimentation that we are trying to encourage.

Our assumption that a vertex can be labeled by a single character means that a vertex will be geometrically small and

not vary in size due to the amount of labeling data. Under such conditions the graph layout algorithm described by Kamada in [3] provides an excellent means of drawing a graph using straight lines in such a way that edge intersections are minimized. This minimization of edge intersections is particularly important in weighted graphs because we must have space for displaying the textual weight along an edge in addition to the line that actually composes the edge. Providing all this information in a way that remains legible requires exceedingly judicious use of screen space. Prior to the development of our *VisualGraph* class, graph display systems employing such sophisticated algorithms had only been available in research-oriented tools such as COOL [4]. The combination of providing such a geometric layout algorithm with the ability to randomly generate meaningful graphs for assignment problems provides students with a rich visual playground in which to watch their algorithms work on many varied data sets.

We chose AnimalScript [8] for implementing the graphics in the *VisualGraph* class. AnimalScript is a scripting language in which textual animation commands are written to a file that is then parsed and rendered by the Animal animation system [8]. Animal is one of several animation engines supported by *JHAVÉ* [6], a platform that offers such engines the ability to intersperse instructional interaction objects into the stream of rendering information.

4 Implementation of the VisualGraph class

As outlined above, all graph vertices in *VisualGraph* are identified by a character. To support large graphs, characters go from upper to lower case, followed by digits. Accordingly, edges are identified by the character IDs of the vertices they connect, with the first character representing the starting vertex for directed graphs.

By default, *VisualGraph* generates AnimalScript output (optimized for size) to a Java *PrintWriter*. The resulting stream can then be used by *JHAVÉ*. Using the latest extensions of both AnimalScript and *JHAVÉ*, the stream may also contain interaction elements for asking questions about the display [6]. Because *VisualGraph*, AnimalScript, and *JHAVÉ* are all written in Java and use text to describe rendering information, algorithms developed in *VisualGraph* and their resulting animations are portable to virtually all environments.

Four methods support adding or removing vertices and edges from a graph. The position of a vertex is given by Cartesian coordinates, while the position of an edge is determined by the vertices it joins. The color and highlighting for added elements can be initialized. A set of auxiliary methods allows (un-)highlighting or coloring edges or vertices, toggling the highlight mode, and changing the weight of an edge. Apart from querying the existence of vertices and edges and their highlight state, the weight of a given edge can be retrieved for weighted graphs. Graphs can be tested for being empty, directed, or weighted.

VisualGraph can generate random graphs with certain properties, such as connected, directed acyclic, Hamiltonian, complete, or random. Each generation method (except for complete graph generation) requires the number of vertices and edges. Self-loops and weights are optional, as well as whether the graph is directed (apart from the directed acyclic random graph). The user can also specify minimum and maximum weights. A given graph can be saved and loaded to and from a file.

The *organizeGraph* method is used to geometrically layout the graph vertices in an aesthetically pleasing way, based on the Kamada algorithm [3]. Vertices are first arranged around a half-unit circle. The vertices with the highest "energy" are then moved until their energy reaches a (local) minimum, with the energy determined by the proximity to other nodes. The process is repeated until the sum energy of the graph reaches a (local) minimum. Vertices are then exchanged to test for lower-energy arrangements. If a lower energy state is achieved, the entire process is repeated. After thus determining the position of all vertices, the graph is suitable for Animal display. The overall efficiency of this process is $O(V^3)$, where V is the number of vertices.

As illustrated in Figure 1, while the display of the graph is usually quite good, it is not necessarily "optimal" with regard to the number of crossing edges. However, when considering that optimal graph layout is NP-complete, this is an acceptable compromise.

5 Example Application: Dijkstra's Shortest Path

Consider the graph portrayed in Figure 1. This represents *VisualGraph*'s depiction of a connected graph that was randomly generated and then sent an *organizeGraph* message so that it would be subjected to the Kamada layout algorithm. Given appropriate input values for the variables *nodes (8)*, *edges (19)*, *selfLoops (false)*, *weighted (true)*, *directed (false)*, *minWeight (2)*, and *maxWeight (8)*, only three Java instructions are required to generate the graph and then produce its picture:

```
g.randomConnectedGraph(nodes, edges,
    selfLoops, weighted, directed,
    minWeight, maxWeight);      // generate
g.organizeGraph();              // layout
g.printGraph();                 // display
```

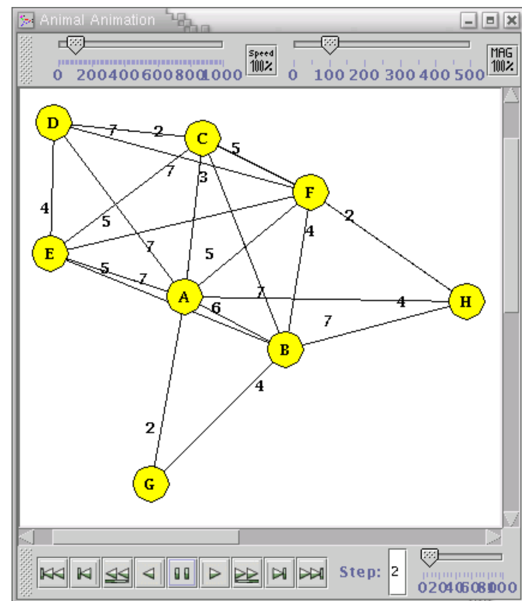

Figure 1: Unadorned graph snapshot

Because the graph is displayed in the Animal rendering engine, the user interface for walking through snapshots of the graph is automatically equipped with a vast array of tools, including VCR-like controls, snapshot magnification controls, speed controls for playing the snapshots in automated slide-show mode, and stepping controls for rewinding the algorithm's snapshots to any point in time.

The following Java code shows how a student may implement Dijkstra's shortest path algorithm on such a graph:

```
void leastSearch(VisualGraph g,char strt,
     char goal) throws IOException {
 char nextNode, n;
 initOpenList(g, strt);
 do {
  nextNode = findOpenNodeToExpand(g);
  g.setNodeColor(nextNode, "red");
  if (nextNode != goal) {
   nodes = g.allAdjacentNodes(nextNode);
   while (nodes.hasMoreElements()) {
    n = ((Character)nodes.
           nextElement()).charValue();
    if (!nodeClosed(n, g))  {
      updateStatus(nextNode, n, g);
      g.setNodeColor(n, "green");
    }
   }
  }
  g.printGraph(X_SIZE, Y_SIZE);
  closeNode(nextNode, g);
  g.setNodeColor(nextNode, "black");
  g.printGraph(X_SIZE, Y_SIZE);
 } while(nodesLeftToExpand(g)
        && (nextNode != goal));
}
```

Here the methods *initOpenList, findOpenNodeToExpand, nodeClosed, updateStatus, nodesLeftToExpand* and *closeNode* represent functions that implement the classic priority queue operations for Dijkstra's algorithm. The italicized messages are sent to the *VisualGraph* object *g*.

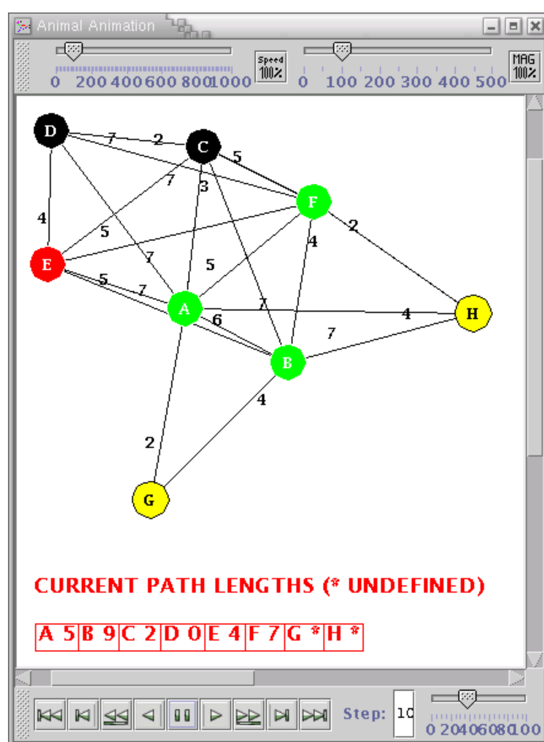

Figure 2: Graph snapshot during Dijkstra's algorithm

The *allAdjacentNodes* message sent to *g* returns a Java Enumeration that easily allows the student to walk through all the vertices adjacent to the vertex called *nextNode*. The *setNodeColor* messages are sent to *g* after *nextNode* is removed from the open list and again after node *n* has had its status on the open list updated. These messages color the node removed from the open list and its adjacent nodes red and green, respectively. Consequently, the *printGraph* message to *g* after leaving the loop that iterates through the Enumeration produces a snapshot that, by the use of colors, reflects the altered state of the algorithm. The *printGraph* message sent to the graph *g* after *nextNode* is closed produces the next snapshot in which closed nodes are colored black. One of the resulting snapshots from this adornment is shown in Figure 2.

The legend printed out below the graph in Figure 2 is an example of how supplementary material above and beyond the graph itself can be streamed into the animation script. To produce this legend, the student (or educator) would have to write a function (here called *legend*) that produces a Vector of Strings representing the state of the total-cost array associated with graph vertices. The addition of the following lines of Java code each time through the outer loop then produces the legend in the ANIMAL renderer. Here, *animationScript* is the same *PrintWriter* to which the *VisualGraph* *g* is directing its rendering commands.

```
Vector v = legend(g);
for (int i = 0; i < v.size(); i++)
  animationScript.println("arrayPut \""
    + (String)v.elementAt(i)
    + "\" on \"legend\" position " + i);
```

6 Future Directions

This paper has focused on showing how easy it is for students using the *VisualGraph* class to generate test data for and produce high-quality animations of graph algorithms that they are implementing as part of a normal assignment for an undergraduate data structures and algorithms course.

The scope of the paper has prevented us from discussing the details of how one would insert complicated interaction objects, such as stop-and-think questions and hyper-document links, into an animation produced by the *VisualGraph* class. Presently doing this requires annotation of the algorithm in a fashion similar, albeit more complicated, than that used to produce the legend in Figure 2. Detailed knowledge of the rendering language [8] and the *JHAVÉ* instructional delivery environment are required. Hence, the addition of such embellishments are a time-consuming process for an instructor (or very ambitious student) who wants to augment an animation with these interaction objects. We hope in the future to use XML specifications for such interaction objects and thereby "abstract away" the details of incorporating them into an animation produced by *VisualGraph*.

References

[1] Hundhausen, C. D., and Douglas, S. Using Visualizations to Learn Algorithms: Should Students Construct Their Own, or View an Expert's? *IEEE Symposium on Visual Languages, Los Alamitos, California* (2000), 21–28.

[2] Johnsonbaugh, R., and Kalin, M. A Graph Generation Software Package. In *Papers of the Twenty-Second SIGCSE Technical Symposium on Computer Science Education* (1991), ACM Press, pp. 151–154.

[3] Kamada, T. *Visualizing Abstract Objects and Relations.* World Scientific Publishing, 1989.

[4] Kamada, T., and Kawai, S. A General Framework for Visualizing Abstract Objects and Relations. *ACM Transactions on Graphics (TOG) 10*, 1 (1991), 1–39.

[5] Mehlhorn, K., and Näher, S. LEDA: a Platform for Combinatorial and Geometric Computing. *Communications of the ACM 38*, 1 (1995), 96–102.

[6] Naps, T., Eagan, J., and Norton, L. JHAVÉ: An Environment to Actively Engage Students in Web-based Algorithm Visualizations. 31st *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2000), Austin, Texas* (Mar. 2000), 109–113.

[7] Naps, T. L., Rößling, G., Almstrum, V., Dann, W., Fleischer, R., Hundhausen, C., Korhonen, A., Malmi, L., McNally, M., Rodger, S., and Velázquez-Iturbide, J. Á. Exploring the Role of Visualization and Engagement in Computer Science Education. *To Appear in ACM SIGCSE Bulletin 35*, 1 (Mar. 2003).

[8] Rößling, G., and Freisleben, B. Program Visualization Using AnimalScript. *First International Program Visualization Workshop, Porvoo, Finland. University of Joensuu Press* (July 2001), 41–52.

[9] Rößling, G., Schüler, M., and Freisleben, B. The Animal Algorithm Animation Tool. 5th *Annual ACM SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education (ITiCSE 2000), Helsinki, Finland* (July 2000), 37–40.

[10] Stasko, J. Using Student-built Algorithm Animations as Learning Aids. 28th *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '97), San Jose, California* (Feb. 1997), 25–29.