Journal of Computer Science 6 (1): 87-91, 2010
ISSN 1549-3636
© 2010 Science Publications

A Free Educational Java Framework for Graph Algorithms

Gianluca Costa, Claudia D'Ambrosio and Silvano Martello
Dipartimento di Elettronica, Informatica e Sistemistica, University of Bologna,
Viale Risorgimento 2, 40136 Bologna, Italy

Abstract: Problem statement: In the teaching of graph theory algorithms it is important that students can experiment them on numerical instances in order to fully understand their logical meaning and to learn how they can be implemented. **Approach:** We developed an open source Java framework to help students in their approach to the study of graph algorithms. The framework was implemented so as to be portable and easy to use. In addition, we included a library that anyone can easily use to develop custom algorithms. **Results:** The framework, which is currently in use at the University of Bologna, and is available on-line, presently includes four basic algorithms in graph theory, for the solution the following problems: shortest spanning tree, shortest paths, maximum flow, and critical path. It includes an intuitive graphical user interface, and gives the users the possibility of performing a "step-by-step" execution. **Conclusions:** The presented Java framework constitutes a first step towards the implementation of didactical instruments for the teaching of graph theory. Future developments will also include a new major release and an implementation targeting the Microsoft. NET framework.

Key words: Computer-aided education, visualization, graph theory, algorithms

INTRODUCTION

The official birth date of graph theory is 1736, when Leonard Euler (Euler, 1736) solved the famous problem of the seven bridges of Königsberg through a graph model. For many years the use of this discipline was mainly limited to recreational mathematics like, e.g., the Icosian Game, invented in 1857 by the Irish mathematician William Rowan Hamilton: This puzzle required finding a particular path, later known as the Hamiltonian circuit, in a graph defined by the edges of a dodecahedron. In the mid Nineteenth century however, graph theory also began being used for the solution of practical problems: In 1845 physicist Gustav Kirchhoff (1845) employed it to calculate the currents in electrical networks. Later, Arthur Cayley, James J. Sylvester and George Polya used graph theory concepts to enumerate chemical molecules. In the Twentieth century, the development of many algorithms for a huge number of problems arising on graphs greatly increased its use in the solution of optimization problems arising in many fields. Today graph theory is a recognized powerful tool and the great majority of courses in operations research include it.

In the teaching of graph theory algorithms it is important that students can experiment them on numerical instances in order to fully understand their logical meaning and to learn how they can be implemented. GraphsJ, the Java framework described in this study, was developed to help students in their approach to the study of graph algorithms. Several graph-oriented applications exist nowadays(Lau, 2006; Sedgewick, 2003; Wu, 2005), but it is rare to find a number of important general features in one program. Indeed, in our opinion, a useful educational application should be:

- Portable: It should run on most operating systems
- Easily extendable: Even non-experienced programmers should be able to quickly create and run a new algorithm
- Easy to use: It should enable students to enjoy their learning experience
- Open source

GraphsJ provides a standalone application and a library that anyone can use to develop custom algorithms: A new algorithm can be created by just inheriting from few abstract classes. Its graphical user interface, the possibility of performing a "step-by-step" execution and the detailed output information it provides proved to help the students in their learning of basic graph algorithms. At the home page http://www.or.deis.unibo.it/staff_pages/martello/GraphsJ/GraphsJ.html the user can either execute GraphsJ without installing it via Java Web Start (JWS) or

Corresponding Author: Silvano Martello, DEIS, University of Bologna, Viale Risorgimento 2, 40136 Bologna, Italy

download its Software Development Kit (SDK).

MATERIALS AND METHODS

Algorithms: The following algorithms were implemented in GraphsJ, for the solution of four basic graph theory problems arising in different graph families:

- Shortest spanning tree: Given an undirected graph $G = (V,E)$, with vertex set $V =\{v_1, ..., v_n\}$, edge set $E\subseteq\{(v_i ,v_j): v_i, vj\in V \text{ and } i<j\}$ and integer weight $w(v_i ,v_j)$ associated with each edge $(v_i ,v_j)\in E$, find a spanning tree $G' = (V, E')$ of G such that $\sum_{(vi\ ,vj)\in E'} w(v_i ,v_j)$ is a minimum. Implemented algorithm (Prim, 1957)
- Shortest paths: Given a directed graph $G = (V,A)$, with vertex set $V = \{v_1, ...,v_n\}$, arc set $A\subseteq\{(v_i,v_j): v_i , v_j\in V\}$ and non-negative integer length $w(v_i,v_j)$ associated with each arc $(v_i ,v_j)\in A$, find the shortest paths from a specified vertex $s \in V$ to all vertices of V. Implemented algorithm (Dijkstra, 1959)
- Maximum flow: Given a network $N = (V,A)$, with vertex set $V = \{v_1, ..., v_n\}$, arc set $A\subseteq\{(v_i,v_j): v_i , v_j\in V\}$ and non-negative integer capacity q_{ij} associated with each arc $(v_i,v_j) \in A$, find the maximum flow from a specified vertex $s\in V$ to a specified vertex $t\in V$. Implemented algorithm (Ford and Fulkerson, 1962)
- Critical path: We are given a directed acyclic activity graph $G = (V,A)$, with vertex set $V = \{v_1, ..., v_n\}$, arc set $A\subseteq\{(v_i,v_j): v_i, v_j\in V\}$ and non-negative integer duration $d(v_i,v_j)$ associated with each arc $(v_i,v_j)\in A$. Arcs represent activities, vertices represent the start and end of activities and the graph itself represents precedence relations among activities. The problem is to find the starting time of each activity so that the makespan between a specified starting vertex $s \in V$ and a specified ending vertex $t\in V$ is a minimum. Implemented algorithm: Critical Path Method (CPM). (CPM was developed in the mid Fifties at the DuPont Company at the same time that the consultant firm of Booz, Allen and Hamilton developed PERT for Lockheed Aircraft Corporation (Siemens, 1971)

A detailed pseudo-code statement of the algorithms can be downloaded from GraphsJ home page. A common characteristic of these algorithms is that they all operate by associating labels to the vertices. In the part devoted to the Java Implementation we detail how they were handled.

Architecture: The choice of Java. The target platform chosen for GraphsJ was the Java Virtual Machine, because of several considerations:

- Unlike other excellent frameworks like Microsoft .NET, Java can run almost flawlessly on several operating systems and portability was one major requirement for GraphsJ
- Java is object oriented: OOP (Object Oriented Programming) is one of the most elegant and productive development models nowadays, as it allows, for instance, to naturally represent interactions between the entities in the application domain
- Java is a worldwide language, therefore a large number of people already know it and are able to develop their algorithms without having to learn a new programming language
- Java is fairly easy, mature and with few subtleties: Scientists who didn't previously study Java can benefit of the gradual learning curve offered by the language

RESULTS AND DISCUSSION

Framework architecture: Similarly to most well-designed object-oriented frameworks, GraphsJ adopts the MVC (Model/View/Controller) pattern (Fig. 1): This means that the model (the data) and the view (what is shown to the user) are kept separated, in our case by using different classes and packages; the controller is the subsystem which fills the gap between them. Strong layer (and sub-layer) separation is a key factor of software robustness and, even if not always perfectly applicable, ensures a substantial benefit to the overall architecture.

To be more precise, GraphsJ:

- Fully implements the model, by introducing classes and utilities related to the domain of graph theory. They are mainly enclosed by package graphsj.model and its subpackages, among which the most important is graphsj.model.graphkeeper
- Realizes the view mainly via frames, dialogs and, notably, via GraphCanvas, the component on which users can draw their graphs. GraphCanvas extends the JGraph component provided by JGraph, a third-party library referenced by the program (Alder, 2000). JGraph is a useful Swing component which enables the program to present the user with a graph canvas; GraphsJ implements this feature by extending JGraph and adding custom editing logic

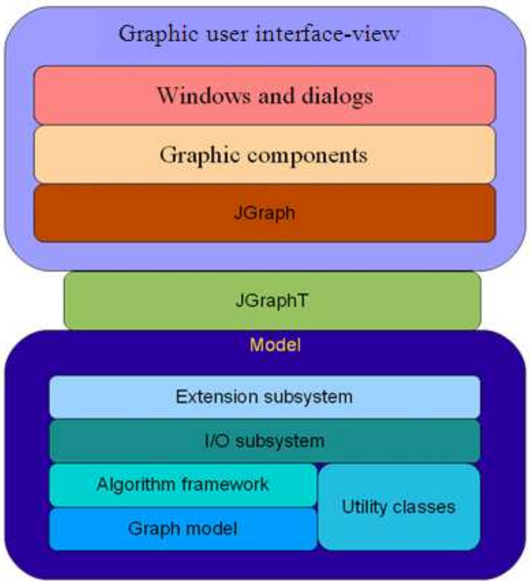

Fig. 1: GraphsJ general architecture

- Implements the controller by introducing the SafeAdapter class, which inherits from JGraphModelAdapter, the controller exposed by JGraphT (Sichi, 2003), which allows easy integration of custom data with the JGraph library

In the model layer, or between it and the controller layer, lies the Extension subsystem, which is able to read custom algorithm classes from external jars (that could be not only local, but also remote files) and to transparently execute them just like the algorithms deployed with the program.

Java implementation:
Model: The model relies on Java generics so as to obtain a coherent type system: Most classes have generic type parameters, starting from the basic GraphObject<V, E>, which models a generic graph attachment and which is subclassed by both Vertex<V, E> and Edge<V, E>. The mutual dependency between Vertex and Edge imposed by their generic parameters (<V, E>), although not mandatory from a theoretical point of view, further enforces type coherence: One cannot insert a wrong kind of vertex into a graph, nor even associate it with an unsuitable kind of edge.

Graph: GraphsJ defines a GraphKeeper<V, E> class, that stores a private instance of the DefaultListenableGraph class provided by JGraphT, which, in turn, could be seen as a graph with events; GraphKeeper exposes the private graph's functionality via public methods such as vertexSet(), edgeSet() and getEdge(): This strong encapsulation guarantees a correct use of the features provided by the graph itself, because every public access to the graph is strictly checked by GraphKeeper.

Vertices and edges: The graph constituents are modelled by the abstract classes Vertex<V, E> and Edge<V, E>: Their behavior is similar, because they both inherit from GraphObject<V, E> and share a common set of public methods to attach or detach them from a graph. In addition, each class exposes public methods that further expose the data of the internal graph. For example:

- Vertex provides-among many others-getIncomingEdges() and getOutcomingEdges()
- Edge provides getSource() and getTarget()

It should be noted that the information provided by these methods sums up global, graph-scoped data, which would not be available to GraphObject instances if the GraphKeeper didn't provide package-level access to its private graph instance, by defining the protected getter:

```
protected DefaultListenableGraph<V, E> getGraph() {
        return graph;
}
```

which, according to Java's visibility rules, can also be called by classes in the same package (in particular, Vertex and Edge).

Algorithm: The algorithm logic is implemented by the Algorithm<V, E> class, whose abstract methods are called by the RunController<V, E> class during algorithm execution. Therefore, from the software engineering viewpoint, Algorithm behaves as a strategy for a RunController instance. Algorithm is a class with very few facilities, useful only if one wants to develop an unusual algorithm. For most cases, StandardAlgorithm or StartStopAlgorithm are a far better choice. The former supports the runtime label mechanism we will discuss below. The latter inherits from StandardAlgorithm and simplifies development by asking both a source vertex and a target vertex at the very beginning of the algorithm execution. It then performs several checks on them. For example, the source vertex must not have incoming edges, whereas the target vertex must not have outcoming edges.

StandardAlgorithm and its protocol: Standard

Algorithm not only inherits from Algorithm, but also narrows its generic constraints; its declaration is: public abstract class Standard Algorithm<V extends Standard Vertex<V, E>, E extends Edge<V, E> and Standard Edge<V, E> extends Algorithm<V, E>> Standard Vertex is declared as an abstract class, whereas Standard Edge is an interface: This is due to the fact that many Edge subclasses exist in the library and can be chosen by the developer (Java does not support multiple inheritance). On the other hand, Standard Vertex directly inherits from the basic Vertex class and is supposed to be the vertex base class of choice. Anyway, they must be implemented by the concrete Vertex and Edge subclasses (respectively) that one wants to associate with a Standard Algorithm subclass. A common trait is that they both require the implementation of method void set Algorithm(Standard Algorithm<V, E> algorithm), which is called by Standard Algorithm at the beginning of every algorithm execution so that every vertex and edge always has a reference to the running algorithm. The advantage of this choice becomes clear if we consider the following code extract, from class Prim Vertex (which models a vertex for Prim's shortest spanning tree algorithm):

```
if (getAlgorithm().getRunController().isVerboseRun())
{
    return String.format("%s {%s, %s}", getName(),
    bestVertexName,
    weightFromBestVertex.toString());
} else {
    return String.format("%s {%s}", getName(),
    bestVertexName);
}
```

As we can see, getAlgorithm() is used to retrieve the RunController instance used to manage the current algorithm execution and, consequently, the current run options; in this case, isVerboseRun() returns true if the user has chosen 'Verbose run' in the Run menu.

Algorithm labels: As we stated before, all the implemented algorithms rely on labels not only to process their data, but also to show their current state to the user. The combination of JGraph and JGraphT shows, for both vertices and edges, the return value of the toString() method. During the development of GraphsJ, a problem arose: When designing the graph, vertices must only show their name (for example, 'V1'), whereas at run-time they must show their full label (that is, the name with additional algorithm-related data). The chosen solution was to override toString() in StandardVertex as follows:

```
@Override
public String toString() {
    if (algorithm != null) {
            return getRunningLabel();
    } else {
            return super.toString();
    }
}
```

where getRunningLabel() is an abstract methods whose signature is:

```
Protected abstract String getRunningLabel();
```

In other words, at design-time just the vertex name-provided by the base class-is shown to the user, whereas at run-time the label text is left to the concrete subclass implementation. For what concerns edges, no particular run-time behavior has been provided - but it should easily be implemented by custom subclasses through using the setAlgorithm() hook method, always called when the execution of a Standard Algorithm begins. It should also be noted that there is no method called when algorithm execution ends. The reason is quite simple. When the user runs an algorithm, the input graph, drawn on the main Graph Canvas, is copied and the new graph is assigned to another Graph Canvas, which is the one shown during algorithm execution. This means that the algorithm works on a copy of the original graph, leaving the input untouched. When the algorithm ends, the graph used during its execution is no more used and can be removed by the garbage collector.

Controller: The controller is mainly realized by the SafeAdapter<V, E> class, which inherits from JGraphModelAdapter, a class provided by JGraphT whose duty is basically to watch the graph, draw it on a JGraph component and react to changes in the model by updating the view. SafeAdapter adds some useful methods to ease the management of fonts and colors: for example, setVertexFontSize() and setEdgeFontSize() enable the developer to programmatically set the font size for vertices and edges; makeUpEdge() can change both the line width and the color of an edge.

View: The most important feature of the view is the GraphCanvas component, which enables users to draw graphs in a simple and intuitive way. It inherits from the third-party component JGraph and adds significant custom logic, in particular object editing support: when the user double clicks a vertex or an edge, the underlying GraphObject's edit() method is called; if the algorithm creator overrides the related methods exposed by Vertex and Edge, it is possible to customize the editing process in order to ask for custom data. Many more classes compose the view, starting from MessageProvider, which provides a unified, singleton source of predefined message dialogs: It can be used, for example, to show a 'warning box' or an 'error box' to the user without having to deal with the details of Swing's JOptionPane class.

How to develop a custom algorithm: What follows is a very simple task list useful to briefly introduce the steps required in order to create a custom algorithm pluggable in GraphsJ. For a more complete and detailed description, the reader is referred to Chapter 9 in (Costa, 2009):

- Create the source files to translate the concepts and logic of the problem into software. In particular, one must:
  - Extend the abstract class graphsj.model.graphkeeper.Vertex<V, E> to model the algorithm vertices
  - Extend the abstract class graphsj.model.graphkeeper.Edge<V, E> to model the algorithm edges
  - Inherit from the abstract class graphsj.model.graphkeeper.GraphKeeper<V, E> to join the two classes above in one wrapper containing the graph instance itself
  - Subclass the abstract class graphsj.model.graphkeeper.Algorithm<V, E> to implement the algorithm logic
- Compile the source files with the Java compiler, remembering to reference both the GraphsJ library (GraphsJ.jar) and the two Java libraries on which it depends, namely lib/jgraph.jar and lib/jgrapht-jdk1.6.jar
- Pack all the compiled files in one jar. (This and the steps above can be easily accomplished by using an IDE)
- Run GraphsJ, click on Graph/New... and choose to run a custom algorithm. Then specify the jar's URL (which could be any jar file URL, not necessarily a local file URL) and the fully-qualified class name
- Click OK and begin drawing the graph using the vertices and edges provided by the algorithm

CONCLUSION

We have described GraphsJ, an open source educational Java framework that also provides a library anyone can easily use to develop custom algorithms. We are currently working on a new major release of GraphsJ and on an implementation targeting the Microsoft. NET framework.

REFERENCES

Alder, G., 2000. JGraph. http://www.jgraph.com

Costa, G., 2009. Didactic java software for graph algorithms. Bachelor’s Thesis, University of Bologna.

Dijkstra, E.W., 1959. A note on two problems in connection with graphs. Numerische Mathematik, 1: 269-271. http://shortestpath.wordpress.com/2009/02/23/a-note-on-two-problems-in-connection-with-graphs-dijkstra-1959/

Euler, L., 1736. Solution of a problem relating to the geometry of position. Commentarii academiae scientiarum imperialis. Petropolitanae, 8: 128-140. http://math.dartmouth.edu/~euler/docs/originals/E053.pdf

Ford, L.R. and D.L. Fulkerson, 1962. Flows in Networks. Princeton University Press, New Jersey, USA., ISBN: 10: 0691079625, pp: 198.

Kirchhoff V.S., 1845. Ueber den durchgang eines elektrischen stromes durch eine ebene, insbesonere durch eine kreisförmige (About the electric current running through a plane, especially through a circular form). Annalen der Physik und Chemie, 140: 497-514. DOI: 10.1002/andp.18451400402.

Lau, H.T., 2006. A Java Library of Graph Algorithms and Optimization (Discrete Mathematics and its Applications). Chapman and Hall/CRC, Boca Raton, FL., ISBN: 978-1-58488-718-8, pp: 386.

Prim, R.C., 1957. Shortest connection networks and some generalizations. Bell. Syst. Tech. J., 36: 1389-1401. http://bibnetwiki.org/wiki/Shortest_connection_networks_and_some_generalizations.

Sichi, J.V., 2003. JGraphT. http://jgrapht.sourceforge.net

Sedgewick, R., 2003. Algorithms in Java, Part 5: Graph Algorithms. 3rd Edn., Addison Wesley, Boston, MA., ISBN: 10: 0201361213, pp: 528.

Siemens, N., 1971. A simple CPM time-cost tradeoff algorithm. Manage. Sci., 17: B354-B363. DOI: 10.1287/mnsc.17.6.B354

Wu, M., 2005. Teaching graph algorithms using online java package IAPPGA. ACM SIGCSE Bull., 37: 64-68. http://portal.acm.org/citation.cfm?id=1113879