LayoutShow: A Signed Applet/Application for Graph Drawing and Experimentation
System Demonstration

Lila Behzadi

Department of Computer Science
York University
4700 Keele Street, North York
Ontario M3J 1P3, Canada
lila@cs.yorku.ca

Abstract. LayoutShow is a Java-based multi-threaded applet/application for experimentation with graph drawing algorithms, particularly, force-directed algorithms. The motivation behind the development of this software is the lack of features that would help to experiment, and as a result, understand the behavior of force-directed algorithms in the existing graph drawing software. Some of these features include smooth node-based and iteration-based animations, display of running-time and iteration counts, and variety of initial layout algorithms. LayoutShow supports a number of force-directed graph drawing algorithms as well as layouts based on eigenvectors. Node-based and iteration-based animations have been implemented. In addition, the software provides some algorithms for producing non-random initial layouts for force-directed algorithms. File I/O using GML file format has been implemented. Furthermore, users of LayoutShow applet can choose to perform local file I/O since LayoutShow is a signed applet. To our knowledge, LayoutShow is the first graph drawing software with this feature.

1 Introduction

At the start of our research on improving the existing force-directed graph layout algorithms which resulted in CostSpring layout algorithm [1], we realized that the existing graph embedding software do not accommodate all the features that are useful in experimenting with and better understanding the force directed algorithms. As a result, LayoutShow, an application/applet for graph drawing, was developed.

In this Paper, we first discuss LayoutShow's features. Then, we describe the way that different components of the system interact with each other. This is followed by two snapshots of LayoutShow. Finally, we present a list of known bug and limitations.

LayoutShow has been implemented using JDK 1.1.6 [13] under Solaris 2.5.1 (SunOS 5.5.1). It has also been tested in Linux 2.0.36, Windows 98, and Windows NT 4.0 as an application as well as an applet. The source, and bytecode of

LayoutShow classes for down-load, and the LayoutShow applet can currently be found at "http://www.cs.yorku.cs/˜lila/work.html".

2 Features

2.1 Outline

LayoutShow provides functionality to

– Generate a variety of graphs: complete graphs, rectangular , hexagonal, and triangular grids, complete binary trees, random graphs, hypercubes, and circular graphs.
– Draw graphs using different force-directed Spring algorithms: CostSpring[1] [1], GEM [4], FR [5], KK [8], and a combination of eigenvectors layout [11] and CostSpring [1].
– Generate initial layouts for Spring algorithms using: CostChosen [2] [1], Insert [4], and EigenLayout [11].
– Randomize a graph.
– Obtain graph quality measurements[3]: number of edge crossings, longest edge to shortest edge ratio, edge length deviation, and cost value: a graph layout quality measurement described in [1].
– Perform iteration-based and node-based animations. Some of the Spring algorithms find the forces acting on a node and reposition this node. In this case, it is possible to update the layout after a node is repositioned, node-based animation, or after all the nodes are repositioned once, iteration-based animation. LayoutShow gives the user the opportunity to choose one of these animation types, or no animation at all. For Spring algorithms such as FR [5] which move the nodes at once, only iteration-based animation is applicable.
– Configure the algorithms.
– Label the nodes.
– Reading/Writing a graph from/to disk using the GML [6] file format.

2.2 The LayoutShow Applet

We mentioned that LayoutShow is an application and also an applet. When running LayoutShow as an applet, all the classes that LayoutShow needs are down-loaded in one HTTP transaction. This has been achieved through the use of a JAR, Java Archive file [3]. This may slightly slow down the speed at which the applet is loaded, but it will certainly speeds up the execution once the applet

[1] CostSpring is a variation of spring-based graph drawing algorithms that has been developed as a part of the author's Master Thesis.

[2] CostChosen is an algorithm to generate initial layouts for force-directed algorithms that has been developed as a part of the author's Master Thesis.

[3] These quality measurements are mainly relevant to the layouts produced by a force-directed spring-based algorithm which LayoutShow focuses on.

is loaded and is running. A JAR file can also be compressed, as the JAR file for LayoutShow is. The current size of this file is about 0.5 megabytes.

Due to Java security restrictions a regular applet cannot access files on the local disk[4]. For this reason, VGJ [10], another graph drawing tool that is available as an applet, does not allow load and save operations. Although this restriction exists for applets by default, but Java has provided ways for permitting an applet to access a local disk and have all the privileges that a Java application has. This is done using *signing and verifying* JAR files. We have signed the JAR file of LayoutShow, providing the option of using the LayoutShow applet with all the privileges of an application including File I/O. This is particularly important since traditionally graph drawing applets such as VGJ did not allow file I/O that is an important part of graph visualization[5]. GDS [2] requires the user to send his/her data files to their server, and they return the URL of the file that contains the layout graph (produced on the server side) to the user. This approach is slow specially if the graph is large. In addition, with the increasing speed of the processors that average users currently have, there is no need for relying on a server to do the computations when the computation can be done locally and reasonably fast. The details of the concepts and procedures of signing and verifying an applet can be found in [3].

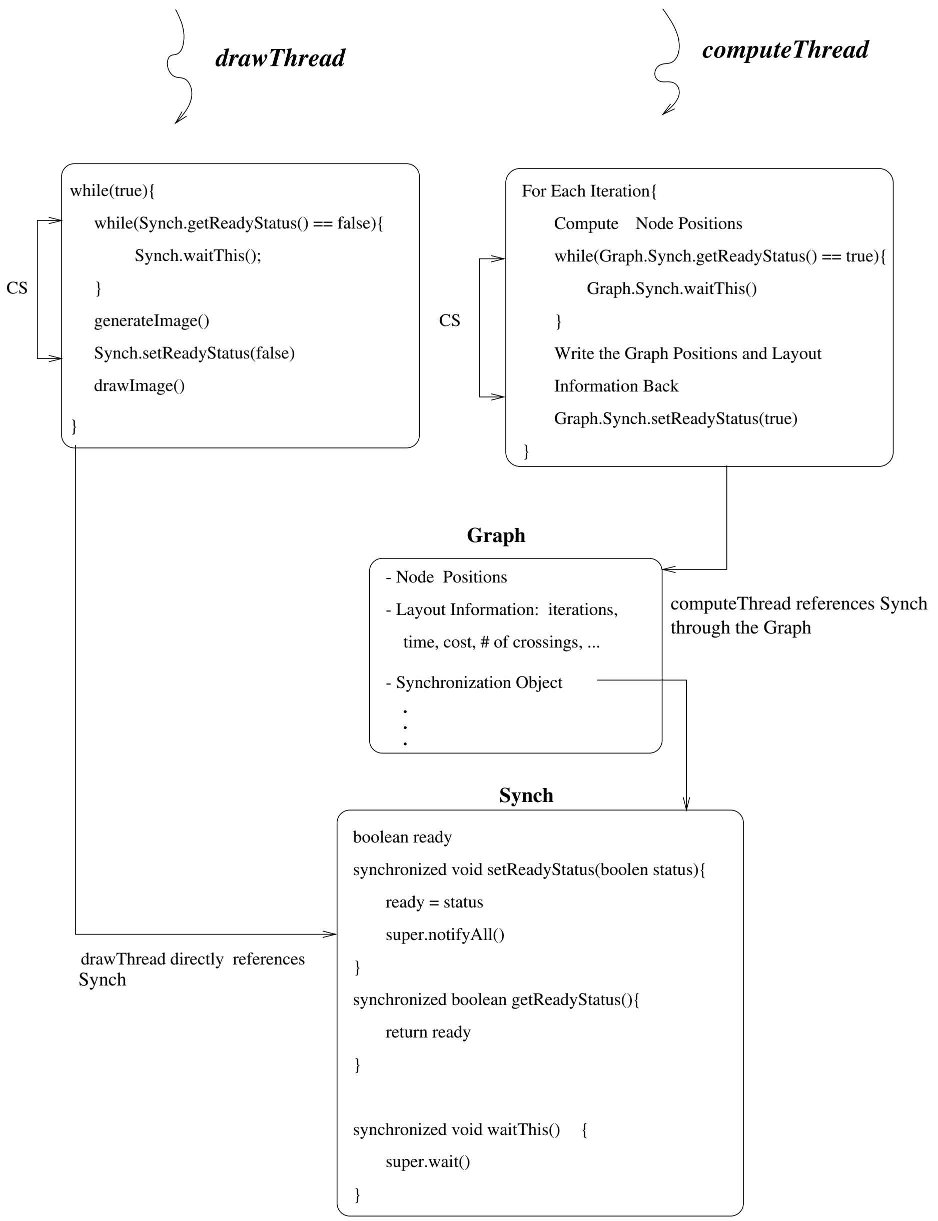
3 The Overall Design of the System

In this Section we elaborate on the way that various components of the LayoutShow software interact with each other to support the computations for finding the new node positions, drawing the graph, and animation. The computation of node positions for a graph (computation module), and its actual drawing on the screen (display module) make up the two main modules of any graph drawing software. Traditionally in software such as Graphlet [7] and VGJ [10], first the new positions of the nodes are computed, and then the graph is (re)drawn. These two actions happen in a sequential order. We have used the multi-threading capabilities of Java for simultaneous position computations and drawing of a graph. To the best of our knowledge on the Java-based graph drawing software, this is a new design on the way these two modules cooperate in such tools. Figure 1 show the relationship between the computation and the display modules.

The Two Threads The tasks of computation and display are managed by two threads: `drawThread` and `computeThread` that share the graph as a common data. As a result, the segments of the code in which these two threads read or modify the graph are considered as critical sections, and must not be executed

[4] Any discussion on Java security in this thesis refers to the Java 1.1.x security model [3], [13].

[5] This is because data may have been produced by another tool and saved in a file, and now the user needs to visualize this data.


drawThread
computeThread
CS
CS
Graph
Synch
computeThread references Synch through the Graph
drawThread directly references Synch

```
while(true){
    while(Synch.getReadyStatus() == false){
        Synch.waitThis();
    }
    generateImage()
    Synch.setReadyStatus(false)
    drawImage()
}
```

```
For Each Iteration{
    Compute   Node Positions
    while(Graph.Synch.getReadyStatus() == true){
        Graph.Synch.waitThis()
    }
    Write the Graph Positions and Layout
    Information Back
    Graph.Synch.setReadyStatus(true)
}
```

```
- Node  Positions
- Layout Information:  iterations,
   time, cost, # of crossings, ...
- Synchronization Object
  .
  .
  .
```

```
boolean ready
synchronized void setReadyStatus(boolen status){
    ready = status
    super.notifyAll()
}
synchronized boolean getReadyStatus(){
    return ready
}

synchronized void waitThis()    {
    super.wait()
}
```

Fig. 1. Synchronization scheme between `drawThread` and `computeThread` in LayoutShow.

simultaneously[6]. Furthermore, the order in which these sections are executed is crucial.

[6] We must note that by simultaneous execution of threads we do not mean that threads run on multiple processors. What we refer to is the processor sharing by multiple threads through pre-emption.

The Critical Sections The two critical sections are:

1. The writing back of the new node positions by the `computeThread`.
2. The reading of these new positions by the `drawThread`.

It is clear that item 1 above should be executed before item 2, and synchronization is required to manage this.

Synchronization The synchronization of the critical sections are managed by using the wait and notify mechanism of Java [9] through a synchronization object: `Synch` (see Figure 1). As we can see in this Figure, the `drawThread` loops infinitely, and in each iteration if the Synch object indicates that the graph is not ready to be drawn then the `drawThread` waits on the `Synch` object[7]. On the other hand, after writing the node positions, the `computeThread` calls `Synch.setReadyStatus(true)` which in effect calls the `notifyAll` function of `Synch`. This resumes `drawThread` which now can start generating the image. At this point, the `computeThread` can continue computing the node positions for the next iteration. However, it cannot write the new positions back to the graph unless the `drawThread` has already called `Synch.setReadyStatus(false)`. If the `drawThread` has not called this function the `computeThread` will wait on the `Synch`. The `drawThread` makes this call after it reads the graph and generates the image, however, the actual drawing of the image occurs after the call to this function. As a result, the code segments that are labeled with CS (for critical section) in Figure 1 cannot be executed simultaneously by the two threads. And, the node positions are computed before the image is drawn. We must also note that the methods of `Synch` object are synchronized, and therefore, only one thread at the time may exist in this object.

For algorithm that don't have multiple iteration, or for when the animation option is off in force-directed algorithms, the node positions are only written back once at the end of the algorithm by the `computeThread`. However, in case of iteration-based animation the node positions are all written back once at the end of each global iteration, and the image is redrawn. In case of node-based animation, the position of the node that has moved is written back and the image is redrawn. For any choice of animation, the computation of the node position(s) and generation/drawing of the graph can occur concurrently resulting in a smoother animation.

Snapshots of LayoutShow Window

Figure 2 shows a snapshot of LayoutShow window with a hexegonal grid drawn using the CostSpring algorithm [1].

Figure 3 displays a snapshot of LayoutShow window performing the Cost-Chosen initial layout algorithm [1]. The middle panel of main window of LayoutShow has `CardLayout` manager which allows for multiple `Component` objects to

[7] Object A waits on object B when object A calls the wait function of object B. A call to notify or notifyAll of B can resume A.

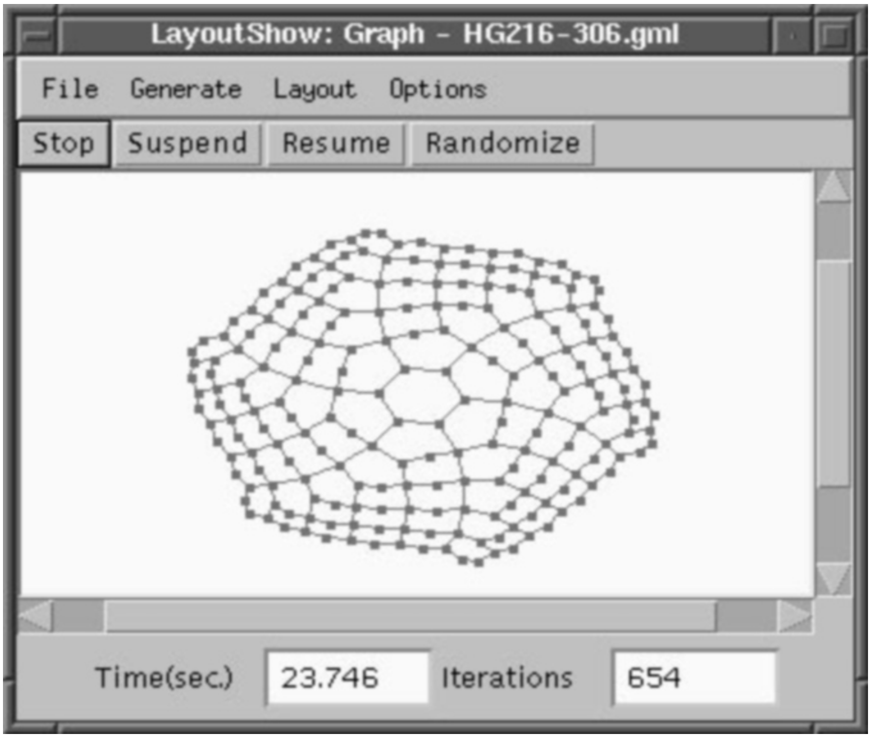


Fig. 2. Snapshot of LayoutShow's main window.

overlap [15]. This middle panel has two overlapping components: the default is a `Canvas` (see Figure 2), and the other one is a `Panel` with 10 `Canvas` objects (laid out in 5 columns and 2 rows) to support the simultaneous execution of Spring algorithms on different initial random layouts of a graph in CostChosen initial layout. The user can choose the number of initial layouts that are used where the maximum number of initial random layouts is 10. Once the algorithms for all initial layouts terminate, then the graph layout quality value of the resulting layouts will appear on top of each canvas with the lowest one flashing in red (see [1] for this quality value's formula). Then after a delay, the resulting layout with the lowest quality value will appear on the single canvas of main window.

4 Bugs and Limitations

The known bugs and limitations of the current implementation of LayoutShow are as follows

- LayoutShow currently does not provide graph editing facilities. This is a feature that will be added to the system.

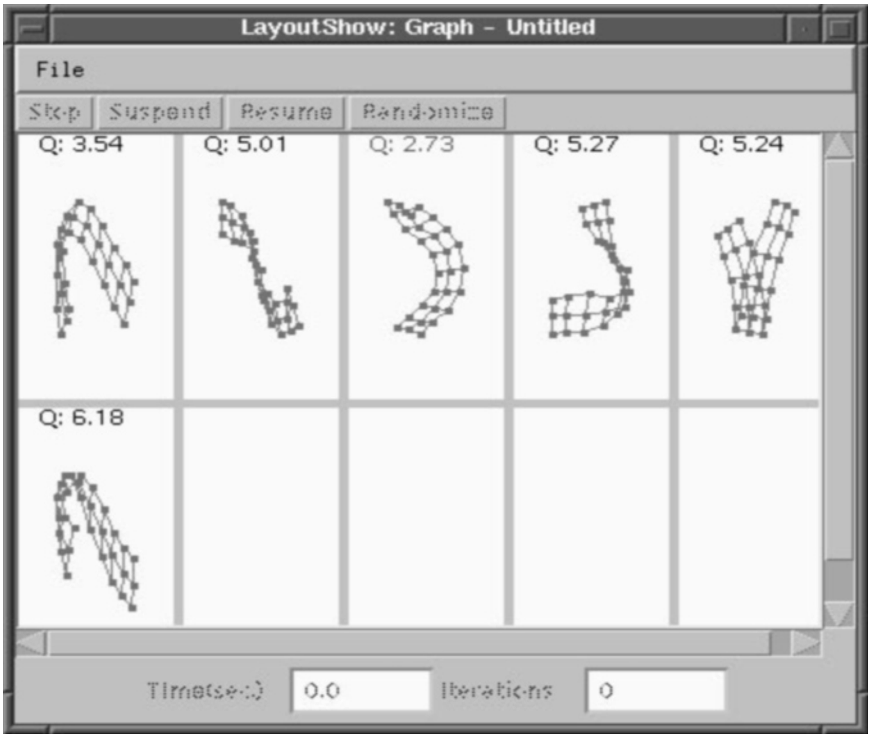


Fig. 3. Snapshot of LayoutShow's main window with a multiple-canvas panel.

- A very simple approach to labeling has been used which results in overlapping labels in some cases.
- Even though, our graph structure supports directed graphs, but the drawing of directed edges have not yet been implemented in LayoutShow.
- Currently, Netscape and Internet Explorer, the two commonly used Internet browsers, can only support Java signed applets [3] if they have a Java Plug-in [14] installed. Our LayoutShow applet is signed by JDK 1.1.6 [13], and tested using Java Plug-in 1.1 [12]. Although, Sun has promised that the final version of Java Plug-in 1.2 would also support the applets signed by JDK 1.1.x, but we have not tested our signed applet using Java Plug-in 1.2.
- The animation in a LayoutShow applet that is running under the default Java virtual machine of a Netscape browser sometimes hangs. We recommend using a Java Plug-in [12].

References

1. L. Behzadi A Cost-oriented Approach to Spring-based Graph Embedding in Layou tShow: a Java Environment for Graph Drawing. Master Thesis, York University, July 1999. Currently available at: `http://www.cs.yorku.cs/~lila/work.html`.
2. S. Bridgeman, A. Garg, and R. Tamassia. A graph drawing and translation service on the WWW. In *Proceedings of Graph Drawing '96*, pages 45–52. Springer-Verlag, 1997.
3. M. Campione, K. Walrath, and A. Huml. *The Java Tutorial Continued : The Rest of the Jdk.* Addison-Wesley, Massachusetts, December 1998.
4. A. Frick, A. Ludwig, and H. Mehldau. A fast adaptive layout algorithm for undirected graphs. In *Proceedings of Graph Drawing '94*, pages 388–403. Springer-Verlag, 1995.
5. T. Fruchterman and E. Reingold. Graph drawing by force-directed placement. *Software-Practice and Experience*, 21:1129–1164, 1991.
6. M. Himsolt. GML: A portable graph file format. Technical report, University of Passau, 94030 Passau, Germany, 1997. Currently available at: `http://www.fmi.uni-passau.de/Graphlet/GML/gml-tr.html`.
7. M. Himsolt. The Graphlet system. In *Proceedings of Graph Drawing '96*, pages 233–240. Springer-Verlag, 1997.
8. T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31:7–15, 1989.
9. D. Lea. *Concurrent Programming in Java.* Addison-Wesley, Massachusetts, 1997.
10. C. McCreary and L. Barowski. VGJ: Visualizing graphs through java. In *Proceedings of Graph Drawing '98*, pages 454–455. Springer-Verlag, 1999.
11. T. Pisanski and J. Shawe-Taylor. Characterizing graph drawing with eigenvectors. Technical report, Royal Holloway, University of London, 1998. Currently available at: `http://www.ijp.si/tomo/papers/papers.htm`.
12. Sun Microsystems. *Java Plug-in Documentation*, 1998. Currently available at: `http://java.sun.com/products/plugin/1.1.2/docs/index.html`.
13. Sun Microsystems. *JDK 1.1 Documentation*, 1998. Currently available at: `http://java.sun.com/docs/index.html`.
14. Sun Microsystems. *Java Plug-in 1.2 Documentation*, 1999. Currently available at: `http://java.sun.com/products/plugin/1.2/docs/index.docs.html`.
15. J. Zukowski. *Java AWT Reference.* O'Reilly and Associates, California, 1997.