JGoLayout™

Automatic Layout Library

for JGo™

User Guide

This guide provides information on using the classes provided in the **JGoLayout™** library that is an add-on to **JGo™**.

January 2005

Northwoods Software Corporation
142 Main St.
Nashua, NH 03060

http://www.nwoods.com/go

mailto:JGo@nwoods.com

CONTENTS

PREFACE

Purpose of this guide:

This guide provides an overview of the classes available in the JGo Layout class library and instructions for incorporating auto-layout functionality into JGo applications.

For more detailed information about the classes and members in the JGo Layout class library, see the JavaDoc-produced API reference.

Who should use this guide:

This guide is intended for application programmers using the JGo Layout library.

Structure of this guide:

This guide is organized as follows:

- Introduction – summarizes the capabilities of the JGo Layout software.
- The Layout Demo Sample Application – introduces the Layout Demo sample application.
- JGo Layout Concepts – describes the overall design of the JGo Layout classes.
- Quickly Adding Layout to Your JGo Application – describes the minimal additions required to add JGo Layout functionality to a JGo application.
- Advanced Options – summarizes some of the most useful options available in the JGo Layout classes.

Assumptions:

This manual assumes you are familiar with Java and JGo programming concepts and terminology. If you are not, please refer to your Java or JGo documentation or online help.

1. INTRODUCTION

The JGo Layout class library is a set of classes built to interface with the JGo class library and provide support for automatically laying out graphs (node & arc diagrams).

Although the classes in the JGo Layout class library are not subclasses of classes in the JGo class library, many aspects of the layout routines take advantage of the fact that JGo objects are targets of the layout.

JGo Layout currently supports two general auto-layout routines: *a force-directed auto-layout* routine and a *layered-digraph auto-layout* routine. The force-directed auto-layout routine is intended for use with all types of graph – undirected graphs as well as directed graphs. The layered-digraph auto-layout routine is intended specifically for use with directed graphs.

Figure 1 and Figure 2 illustrate a sample graph before and after applying force-directed automatic layout.

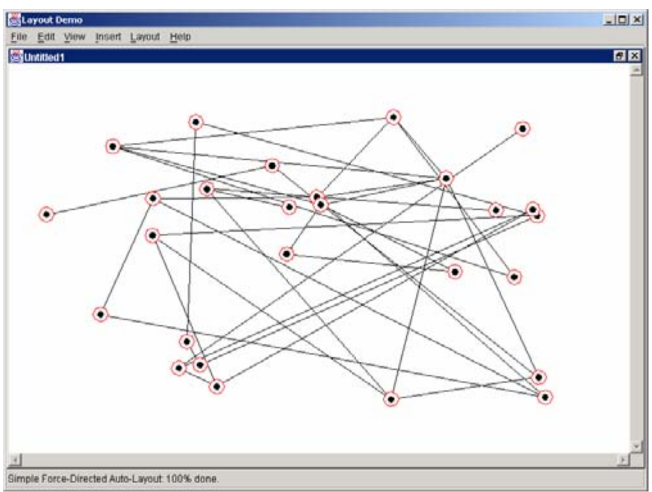


Figure 1. Sample graph before layout

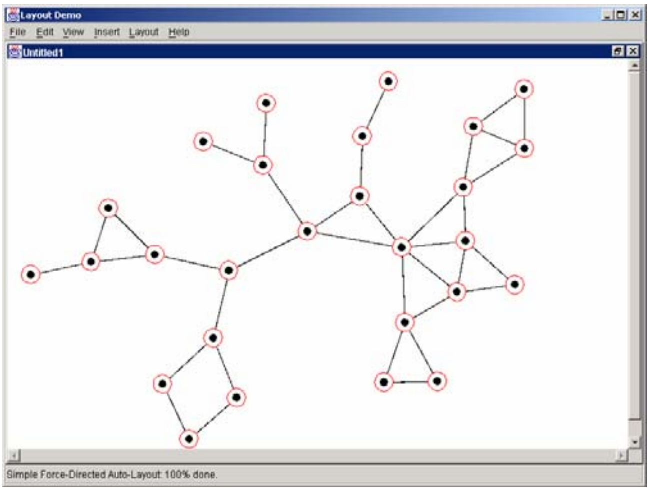


Figure 2. Sample graph after Force-Directed Auto-Layout

Figure 3 and Figure 4 illustrate a sample graph before and after applying layered-digraph automatic layout.

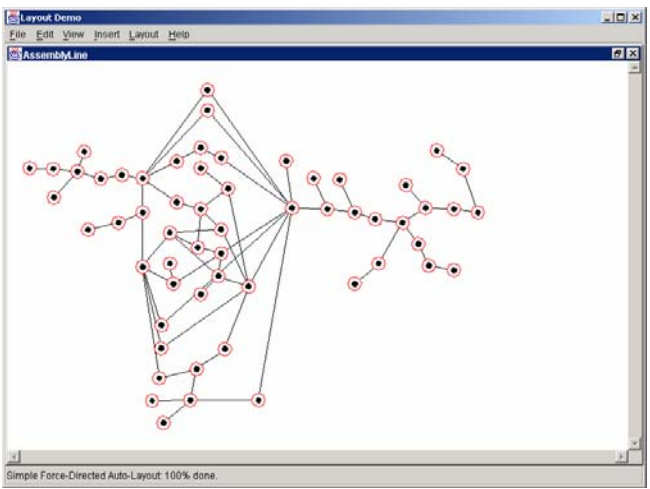


Figure 3. Sample graph before layout

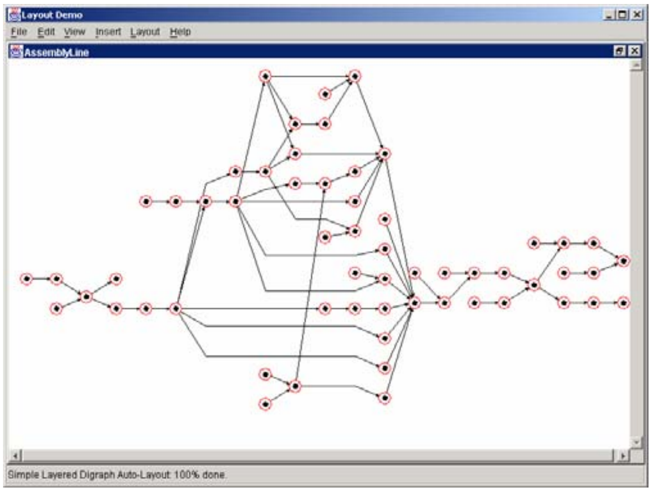


Figure 4. Sample graph after Layered-Digraph Auto-Layout

The JGo Layout class library is designed to be flexible and extensible. All Layout objects are easily subclassed for application-specific specialization. New Layout objects can be easily added to the existing framework.

2. THE LAYOUT DEMO SAMPLE APPLICATION

Introduction to the “Layout Demo” Sample Application

“Layout Demo” is the primary sample application for the JGo Layout library.

The goal of Layout Demo is to demonstrate as many features of the JGo Layout library as possible, but to remains simple enough so that most of what you see in Layout Demo are fundamental capabilities of JGo Layout.

Note: Layout Demo is not suitable as a sample application for learning about JGo. Layout Demo takes advantage of JGo primarily as a framework for drawing graphs.

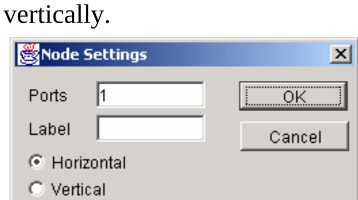
Layout Demo Menus

This section describes the Layout Demo menu commands.

File Commands	Description
New	Opens a new DemoDocument, which is a simple class, derived from JGoDocument.
Open	Opens an existing DemoDocument using a binary file format.
Close	Closes the active DemoDocument.
Save, Save As	Saves the active DemoDocument using a binary format.
Print	Printing support provided by JGo.
Exit	Exits Layout Demo

Edit Commands	Description
Cut	Copies the current selection from the document to the clipboard, while removing the selection from the document.
Copy	Copies the current selection from the document to the clipboard.
Paste	Pastes a previously cut or copied selection from the clipboard.
Delete	Deletes the selected items.
Select All	Selects all items.

View Commands	Description
Zoom Normal	Sets the current scale to 100%.
Zoom In	Adds 10% to the current scale.
Zoom Out	Subtracts 10% from the current scale.
Zoom To Fit	Sets the current scale to the largest scale such that the entire document is visible.
Toggle Grid	Turns the background grid on or off.
Toggle Arrowheads	Turns arrowheads on links on or off.

Insert Commands	Description
Basic Node	Opens a dialog box for creating a new node. The dialog prompts for the number of ports, an optional node label, and whether the ports should be oriented horizontally or vertically. 

Layout Commands	Description
Random Auto-Layout	Performs a randomizing auto-layout on the document.
Force-Directed Auto-Layout	Performs a force-directed auto-layout on the document.
Layered-Digraph Auto-Layout	Performs a layered-digraph auto-layout on the document.

Help Commands	Description
About	Opens a message box with information about the LayoutDemo application.

Layout Demo Quick Start

This section provides a quick introduction to the Layout Demo application and the auto-layout routines.

Force-Directed Auto-Layout

First, we examine the force-directed auto-layout routine. To begin, create a number of one-port nodes using the **Basic Node** menu item or by double clicking on the background. Move them around and link them together into a cycle to create a graph similar to that in Figure 5. Notice that the nodes have an initial color of red. LayoutDemo uses the color of a node to demonstrate some of the customizable aspects of the Layout routines. Double-click anywhere inside of a node to change its color.

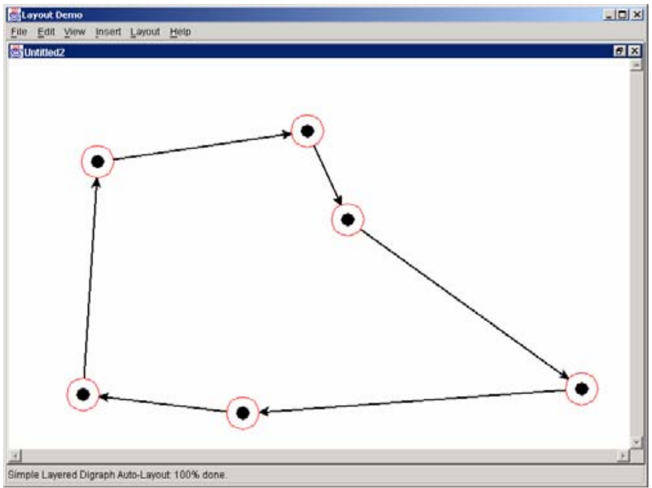


Figure 5. Example 1

Nodes can be linked together by clicking on a port and dragging towards another port. A successfully created link will draw a directed arrow from one node to the other.

After creating a graph, choose the **Force-Directed Auto-Layout** menu item. This will bring up the dialog box illustrated in Figure 6.

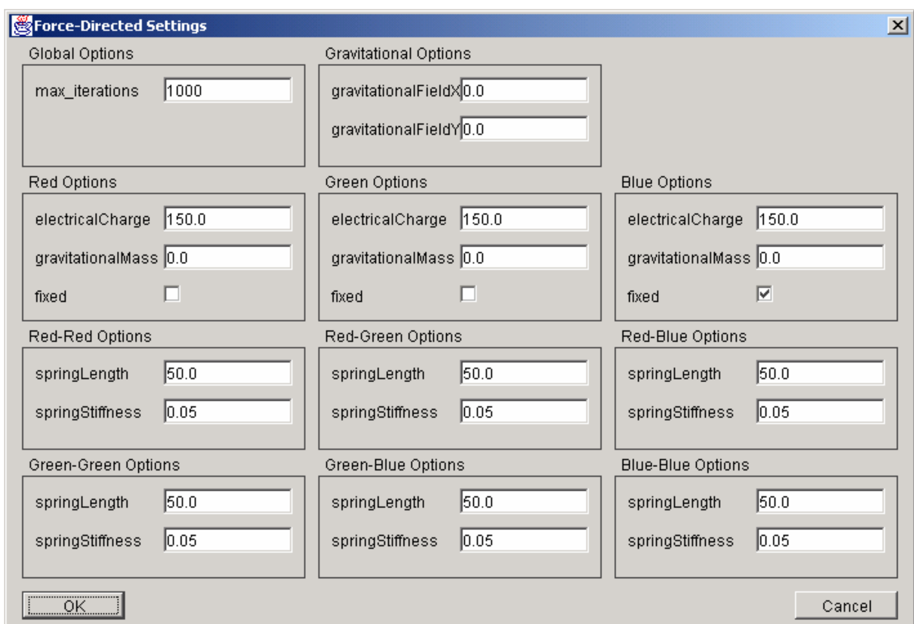


Figure 6. Dialog box for Force-directed Auto-Layout

Examine the different options available for the force-directed auto-layout, but leave the default values and click OK. The graph will animate as it moves towards its final position, similar to that shown in Figure 7.

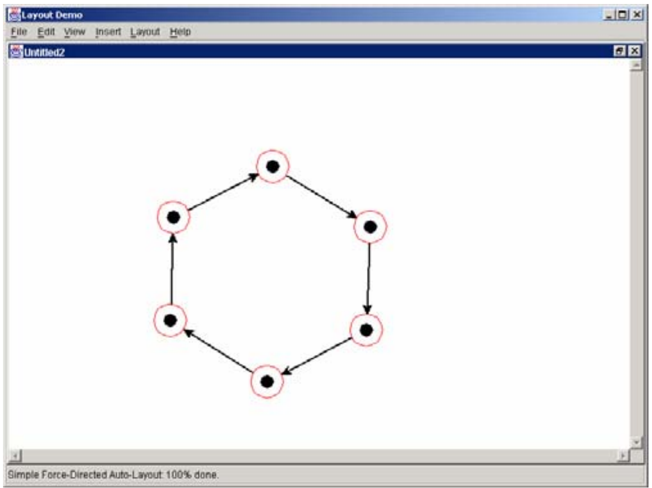


Figure 7. Result of applying Force-Directed Auto-Layout to Example 1

The force-directed auto-layout routine works by viewing a graph as a system of bodies with forces acting between the bodies. The routine tries to move each node into a position such that the sum of the forces acting on the node is zero. In particular, nodes are replaced by electrically charged particles that repel each other and links are replaced by springs that connect the particles.

The different options available for the force-directed auto-layout allow you to adjust the characteristics of the particles and springs that determine the layout of the graph.

See what happens when you change some of the default values. Choose the **Force-Directed Auto-Layout** menu item, but change the value of **electricalCharge** under **Red Options** to 300.0 and click **OK**. Notice that with a higher electrical charge, the nodes repel each other more, and the result is a graph with greater distances between adjacent nodes.

On the other hand, if you change the value of **springStiffness** under **Red-Red Options** to 0.2 and click **OK**, then the stronger springs will result in a graph with smaller distances between adjacent node.

As a final example, move one node some distance away from the rest of the nodes. Double-click on the node to change its color to green. Choose the **Force-Directed Auto-Layout** menu item, select **fixed** under **the Green Options**, and change the value of **springLength** under **Red-Green Options** to 100.0 and click OK. Now, the green node will remain fixed and the other nodes move towards it. Further, the longer **springLength** between the red and green nodes will result in a greater distance between the red and green nodes than between the red nodes, as illustrated in Figure 8.

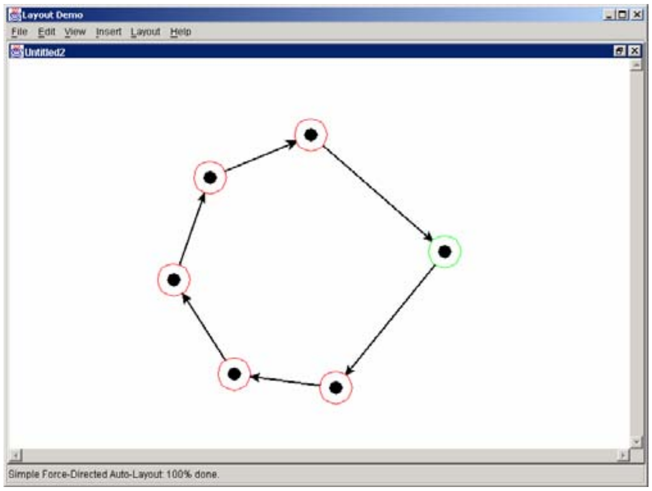


Figure 8. Result of changing parameters

Try adjusting the values of the other parameters to see their effect on the layout.

Setting a **gravitationalFieldX** and **gravitationalFieldY** induces a field over the entire document. The gravitational field only affects nodes with a **gravitationalMass**. Try values of 1.0 for **gravitationalFieldX** and 1.0 for **gravitationalMass**.

Layered-Digraph Auto-Layout

Next, we examine the layered-digraph auto-layout routine. Create a new document, create a number of one-port nodes using the **Basic Node** menu item or by double clicking on the background, move them around, and link them together into a tree similar to that shown in Figure 9.

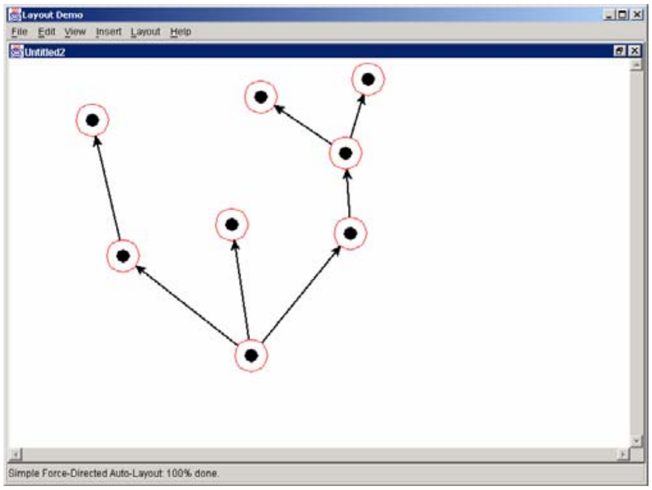


Figure 9. Sample Directed Graph

Now, choose the **Layered-Digraph Auto-Layout** menu item. This will bring up a large dialog box, similar to that shown in Figure 10.

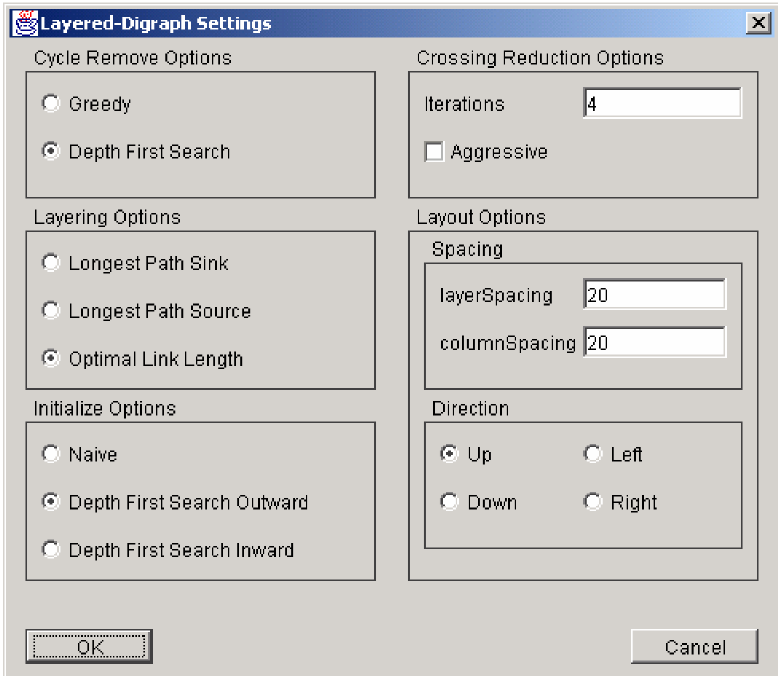


Figure 10. Layered Digraph Auto-Layout Options dialog box

Examine the different options available for the layered-digraph auto-layout, but leave the default values and click **OK**. The graph will be redrawn in its final position as shown in Figure 11.

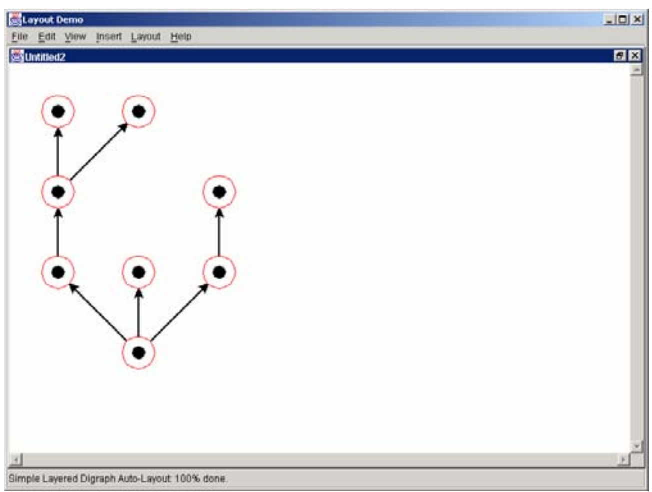


Figure 11. Resulting layout after Layered-Digraph Auto-Layout

The layered-digraph auto-layout routines works as follows: the nodes in the graph are placed into layers such that all of a node's predecessors are in a higher layer and all of a node's successors are in a lower layer; the routine then heuristically permutes the orders of each node within a layer such that the total number of link-crossings is reduced.

Finally, the routine adjusts the positions of each node within a layer to reduce the number of bends required by the links. In order to layout arbitrary directed graphs, the layered-digraph routine removes cycles from graphs by temporarily reversing some links.

In addition, the nodes can be assigned to layers using one of three layering techniques. The iterations value under **Crossing Reduction Options** determines how long the routine looks for ways to reduce the link crossings; however, values higher than 8 rarely have a profound affect on the final drawing. The aggressive option under **Crossing Reduction Option** chooses whether or not to augment the standard crossing reduction step with additional aggressive, but time consuming, passes. Finally, the **layerSpacing** and **columnSpacing** values determine how much space is reserved between adjacent layers and columns. The direction option determines the orientation of the directed links.

Figure 12 illustrates a more complicated graph which has been drawn using the layered-digraph auto-layout routine.

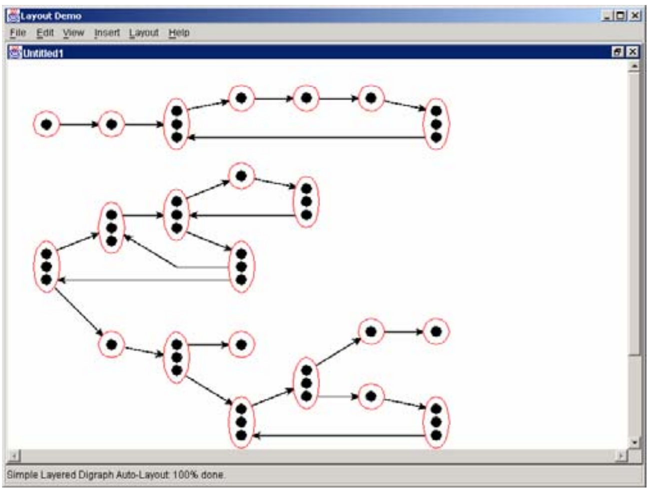


Figure 12. Result of applying Layered-Digraph Auto-Layout to more complex graph

This graph shows the consideration that the layered-digraph auto-layout routines give to nodes with multiple ports. The relative positions of ports within a node are used both in reducing the number of link crossing and in straightening the links.

3. JGo Layout Concepts

Design Philosophy

JGo Layout has been designed to be easy to use, general enough to meet the requirements of a large array of JGo applications, and extensible enough to allow application-specific requirements to be incorporated with minimal effort.

This design philosophy has led to a set of auto-layout classes that export a simple, public interface, but make use of a number of protected functions to provide hooks for specialization.

The default implementations of these functions should be adequate for most applications, but subclassing the JGo Layout classes will often lead to better layouts.

JGoNetwork, JGoNetworkNode, and JGoNetworkLink

The JGoNetwork class provides an abstract view of a JGoDocument as a network (graph) of nodes and directed links. These nodes and links generally correspond to top-level JGoObjects in a JGoDocument.

The JGoNetwork class provides a framework for manipulating the state of nodes and links without affecting the JGoDocument objects. A JGoNetwork is composed of JGoNetworkNodes and JGoNetworkLinks.

By default, a JGoNetwork is constructed from a JGoDocument by adding all top-level JGoObjects that are not ports or links as nodes to the network. Alternatively, a JGoSelection object can be used instead such that only those JGoObjects that are selected are added as nodes and links to the network.

All top-level JGoLinks as are added, by default, as links to the network. If a JGoSelection is provided, then only those JGoLinks that are selected are added as links to the network. Note that links which are selected, but whose corresponding to- and from-nodes are not selected, will not be added to the network.

The majority of applications will simply let the auto-layout class construct the network from a document. They need to construct an auto-layout class from the current JGoDocument or JGoSelection. However, more sophisticated results can be achieved by combining modifications to the JGoNetwork with auto-layout subclasses written to recognize the modifications. In particular, nodes and links, which have no relationship to any JGo object on the screen, can be introduced into the JGoNetwork to influence the final layout. The JGoNetworkNode and JGoNetworkLink classes provide get and set methods for Objects used to hold user information, nodeUserData and linkUserData respectively, which can be used to mark or otherwise distinguish particular nodes and links in the network.

Those interested in writing subclasses of the auto-layout classes should familiarize themselves with the JGoNetworkNode and JGoNetworkLink classes, particularly the getJGoObject() method. This method returns the top-level JGoObject (in the JGoDocument) which is represented by the JGoNetworkNode or JGoNetworkLink. You can construct a JGoNetwork manually, or modify an existing JGoNetwork, using the addNode, deleteNode, linkNodes, addLink, and deleteLink methods. The "add" and "delete" methods are overloaded to either work with JGoNetworkNodes and JGoNetworkLinks directly, or more conveniently when modifying a network, to work with JGoObjects and JGoLinks.

The majority of functions in the auto-layout classes that can be overridden to provide specialized layout routines take JGoNetworkNode or JGoNetworkLink parameters.

The getJGoObject() method will be useful for tailoring the function result to application specific details. However, be aware that some auto-layout classes introduce "artificial" nodes or links, which do not correspond to any top-level JGoObject. For these nodes and links, getJGoObject() returns null.

JGoAutoLayout

All of the auto-layout routines are contained in subclasses of the JGoAutoLayout class. Although the JGoAutoLayout class performs no layout, it defines the public interface inherited by all auto-layout classes. In particular, all auto-layout classes will inherit the following methods:

```
public abstract void performLayout();
public void progressUpdate(double progress) {}
```

The performLayout method is called to perform the actual layout. Since performLayout() is an abstract method in the JGoAutoLayout class, the JGoAutoLayout class is an abstract class; hence, no JGoAutoLayout object can be created.

The progressUpdate method is called by subclasses of JGoAutoLayout at various times with a parameter between 0.0 and 1.0, to indicate the progress through the layout routine. By default, progressUpdate does nothing, but subclasses could override it to provide feedback about the progress of the layout.

In addition, JGoAutoLayout defines a set of constructors that can be used to create an Layout. Any subclass of JGoAutoLayout should call one of these constructors in its own constructor. The default constructor creates an AutoLayout with a null network and a null document. Until the network is set to a non-null value using setNetwork(JgoNetwork n), performLayout() will return without doing anything. The one-argument constructors take in a JGoDocument or a JGoSelection and create a JGoNetwork from the document or selection. The two-argument constructor takes in a JGoDocument and a JGoNetwork. All of these constructors create AutoLayouts that require no other setup before they can perform layouts.

JGoForceDirectedAutoLayout

The JGoForceDirectedAutoLayout class provides an auto-layout algorithm for graphs, which utilizes a force-directed method. The graph is viewed as a system of bodies with forces acting between the bodies. The algorithm seeks a configuration of the bodies with locally minimal energy, i.e., a position such that the sum of the forces on each body is zero.

The JGoForceDirectedAutoLayout class currently makes use of three sets of forces: electrical forces, gravitational forces, and spring forces. Obviously no physical forces are actually used in the layout routine, and the physical model is not 100% accurate. For example, forces always act along lines connecting the centers of nodes, but the distances between nodes are calculated with the size of the node taken into consideration. Hence, there may be some curious results when using the routine on networks with oddly shaped nodes. However, the physical analogy makes the layout routine easier to understand.

Each node in the input network is assigned an electrical charge. Each node repels each other node with a force proportional to the product of their electric charges and inversely proportional to the square of their distance. In addition, each point in the document can be assigned a “horizontal electrical field” and a “vertical electrical field.” A node is acted upon by a force that is proportional to the product of the node’s charge and the field at the node’s location.

Each node in the input network is also assigned a gravitational mass. Although gravitational forces are not exerted between node, each point in the document can be assigned a “horizontal gravitational field” and a “vertical gravitational field.” A node is acted upon by a force that is proportional to the product of the node’s mass and the field at the node’s location.

Finally, each link in the input network is assigned a spring length and spring stiffness. Each link between a pair of nodes exerts a force on the nodes proportional to the product of the spring stiffness and the difference between the spring length and the distance between the nodes.

Additionally, a node can be “fixed,” which means that the node will not be moved by the layout routine, but it will exert forces on other nodes in the network.

The force-directed layout is an iterative process. At each iteration, the placement of the nodes in the document results in forces acting upon each node. Each node is moved a distance proportional to the magnitude of the forces acting upon it. This process is repeated until the forces on each node are reduced to zero, in which case a local equilibrium has been found, or until a maximum number of iterations have been reached.

JGoLayeredDigraphAutoLayout

The JGoLayeredDigraphAutoLayout class provides an auto-layout algorithm for directed graphs. The method uses a hierarchical approach for creating drawings of directed graphs with vertices arranged in layers. The layout algorithm consists of four-major steps: Cycle Removal, Layer Assignment, Crossing Reduction, and Straightening and Packing.

In the Cycle Removal step, all directed cycles are removed from the input network by temporarily reversing some number of links. Two cycle removal routines are provided: Greedy Cycle Removal and Depth First Search Cycle Removal. With Greedy Cycle

Removal, the idea is to induce an order on all nodes in the network (U_1, U_2, U_3, ..., U_k) such that for the majority of links L = (U_i, U_j) it is true that $i < j$. All links L = (U_i, U_j) such that $i > j$ are reversed. With Depth First Search Cycle Removal, a depth first search is performed on the input network. A link L = (U, V) not in the depth first forest is reversed if U was discovered and finished by the depth first search after V was discovered but before it was finished. The Greedy Cycle Removal routine tends to reverse a smaller number of links, but the Depth First Search Cycle Removal tends to preserve a "natural" order to the nodes in the network.

In the Layering step, all nodes in the input network are assigned to layers. If there is a link L = (U, V), then layer(U) ≥ layer(V). Three layering routines are provided: Longest Path Sink Layering, Longest Path Source Layering, and Optimal Link Length Layering. Figure 13 and Figure 14 illustrate the results of each of these.

With Longest Path Sink Layering, every sink node (a node with no links leaving the node) appears in layer 0 and every node is placed as close as possible to a sink.

With Longest Path Source Layering, every source node (a node with no links entering the node) appears in the maximum layer and every node is placed as close as possible to a source.

With Optimal Link Length Layering, nodes are placed in layers to minimize the total weighted link length, where the length of a link L = (U, V) is given by layer(U) – layer(V). For more information about Optimal Link Length Layering, please refer to the Advanced Options section of this guide.

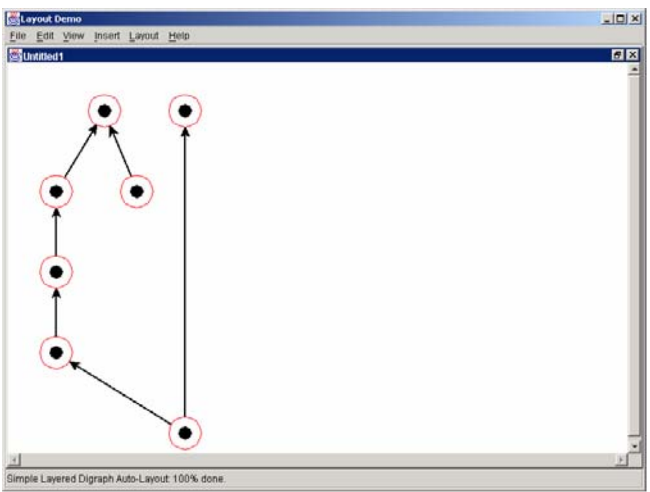


Figure 13. Longest Path Sink Layering

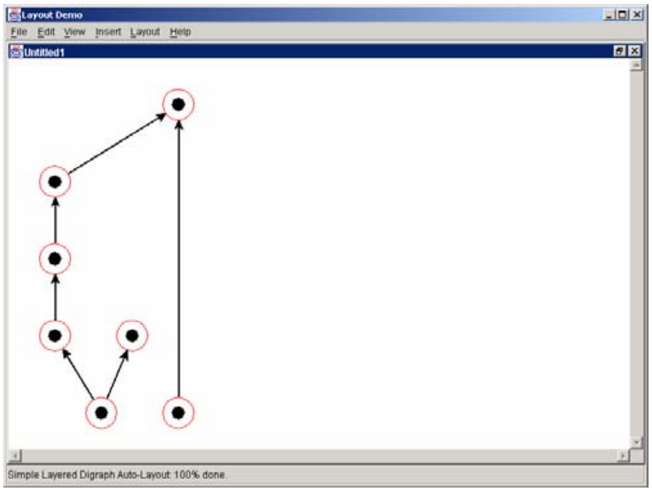


Figure 14. Longest Path Source Layering

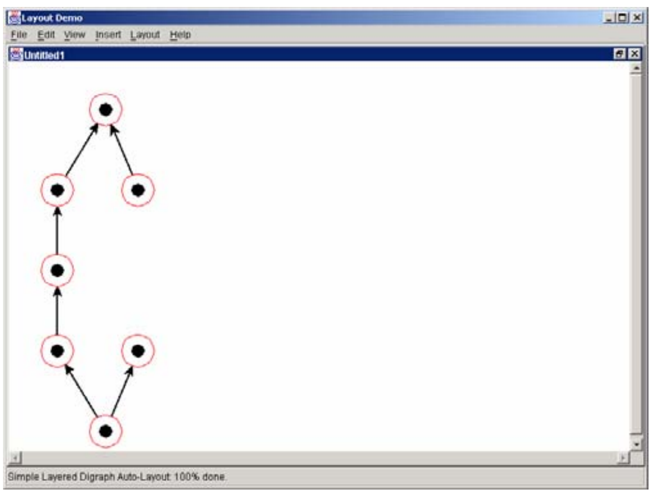


Figure 15. Optimal Link Length Layering

Following the Layering step, there are two minor steps that prepare the network for later steps. The Make Proper step converts the input network into a proper digraph; i.e., artificial nodes and links are introduced into the network such that every link is between nodes in adjacent layers. This has the effect of breaking up long links into a sequence of artificial nodes.

The Initialize Indices step assigns every node (both real and artificial) in the input network an index number, such that nodes in the same layer will be labeled with consecutive indices in left to right order. Three initialization routines are provided: Naïve Initialization, Depth First Out Initialization, and Depth First In Initialization. With Naïve Initialization, nodes are assigned indices as they are encountered in a sweep of the network. Because of the way networks are stored, this has the effect of initially placing all "artificial" nodes to the right of all "real" nodes. With Depth First Out and Depth First Search In, nodes are assigned indices as they are encountered in a depth first search of the network, either from sources outward or from sinks inward.

The Crossing Reduction step reorders nodes within layers to reduce the total number of link crossings in the network. The basic technique is to sweep back and forth over the layers, using heuristics to reduce the number of link crossings between adjacent layers. The first heuristic sorts the nodes in layer by their median and barycenter values, which are calculated by the nodes' neighbors in the adjacent layers. The second heuristic uses a bubble-sort technique on a layer to exchange adjacent nodes whenever doing so reduces the number of link crossings between the layer and its adjacent layers. In addition to the basic sweeping technique, there is an optional aggressive crossing reduction step.

The basic sweeping technique sweeps across all layers of the network, potentially discarding some improvement between one pair of layers because of crossings introduced elsewhere in the graph. Better results can sometimes be obtained by the aggressive technique, which spends more time examining subsets of the layers for local improvements, independent of the rest of the graph. Nodes with multiple ports are recognized by the crossing reduction heuristics and crossings between links that connect to the same node are correctly calculated.

The Straightening and Packing step positions the nodes within each layer to reduce the total number of link bends in the network and to reduce the total width of the network. The basic technique is to sweep back and forth over the layers, using heuristics to reduce the number of link bends between adjacent layers. The heuristics are designed to give higher priority to straightening links that have multiple bend points. In addition, the locations of ports within a node are used to better align links with their connecting points. Between sweeps, the network is "packed" to reduce the total width.

The final step is to Layout Nodes and Links. This step simply translates the position of a node in a layer into a screen position. It also inserts bend points into links that extend across multiple layers. The node and layer spacing parameters and the direction parameter determine the exact layout.

4. QUICKLY ADDING LAYOUT TO YOUR JGO APPLICATION

Integrating JGo AutoLayout into an existing JGo application is very easy. This section will take you through the steps of adding JGo Layout to a generic JGo application.

In the discussion that follows, we will assume that SimpleJGoApp is an existing JGo application In particular, we will assume the existence of the following class: SimpleJGoAppView.

Add an import statement near the beginning of the file:

```
import com.nwoods.jgo.layout.*;
```

In this example, we will invoke the auto-layout routines from simple functions. We will not pay attention to how these methods are called. They may be automatically called when a document is opened, or when the document changed. Or, as is the case in LayoutDemo, they may be run at the user's command.

To add a function to perform Layered-Digraph-Auto-Layouts, create a function with the following code:

```
void layerAction()

{
  JGoLayeredDigraphAutoLayout layout = new
             JGoLayeredDigraphAutoLayout(getDocument());
  layout.performLayout();
}
```

To add a function to perform Force-Directed-Auto-Layouts, create a function with the following code:

```
void forceAction()
{
  JGoForceDirectedAutoLayout layout = new
             JGoForceDirectedAutoLayout(getDocument());
  layout.performLayout();
}
```

That's it!

The constructors for JGoLayeredDigraphAutoLayout and JGoForceDirectedAutoLayout used above initialize the auto-layout options to default values. Clearly, these values will not be suitable for all applications. See the Advanced Options section of this guide and the JGo Layout Reference for details regarding customizing the auto-layout routines. Further customization is available by subclassing the JGo AutoLayout classes.

5. Advanced Options

This section provides details regarding customizing the JGo AutoLayout routines. Referring to the JGo AutoLayout API Reference documentation will be helpful when reading this section.

JGoForceDirectedAutoLayout

Most of the customization available in the JGoForceDirectedAutoLayout class is accessed through overriding methods. However, one critical option can be accessed through the class constructors:

```
JGoForceDirectedAutoLayout(JGoDocument doc, JGoNetwork network,
    int Nmax_iterations)
```

The `Nmax_iterations` parameter sets the maximum number of iterations that the routine should use in looking for a local equilibrium. Be aware that networks with large numbers of nodes and links require more processing during each iteration, so raising the maximum number of iterations is not recommended.

If a JGoForceDirectedAutoLayout is constructed without specifying a maximum number of iterations, it uses a default of 1000. To change the default, use the static method:

```
public static void setDefaultMaxIterations(int x)
```

The following methods are available to customize the "forces" used by the JGoForceDirectedAutoLayout class:

```
protected double getSpringStiffness(JGoNetworkLink pLink)
protected double getSpringLength(JGoNetworkLink pLink)
protected double getElectricalCharge(JGoNetworkNode pNode)
protected double getElectricalFieldX(Point xy)
protected double getElectricalFieldY(Point xy)
protected double getGravitationalMass(JGoNetworkNode pNode)
protected double getGravitationalFieldX(Point xy)
protected double getGravitationalFieldY(Point xy)
protected boolean isFixed(JGoNetworkNode pGoNode)
```

Keeping in mind the description of the force-directed auto-layout routine given in the JGo AutoLayout Concepts section of this guide, the nature of each of these methods should be clear. By default, links have a stiffness of 0.05 and a length of 50, nodes have an electrical charge of 150, a gravitational mass of 0, and are not fixed, and every point in the document has both an electrical field and a gravitational field of 0.0 in both directions.

These methods can be used in a variety of ways to influence the final layout of the nodes in the document. For example, the LayoutDemo sample application overrides the getElectricalFieldX and getElectricalFieldY methods as follows:

```
public double getElectricalFieldX(Point xy)
{
  double border = 200.0;
  double min = 0;

  double force = 300.0;
  if (xy.x <= min)
    return force;
  else if (xy.x <= min + border)
    return (force / ((min - xy.x) * (min - xy.x)));

  return 0.0;
}

public double getElectricalFieldY(Point xy)
{
  double border = 200.0;
  double min = 0;

  double force = 300.0;
  if (xy.y <= min)
    return force;
  else if (xy.y <= min + border)
    return (force / ((min - xy.y) * (min - xy.y)));

  return 0.0;
}
```

This effectively places an “electrical” border along the axes of the document, which prevents nodes from being forced off of the document. The Layout sample application also overrides the other methods in order to use custom values for different colored nodes and links.

If you want to contrain the nodes to be within a particular rectangle, you can modify these overrides as follows:

```
public double getElectricalFieldX(Point xy)
{
  double border = 200.0;
  Point doctopleft = getDocument().getDocumentTopLeft();
  double min = 0; // doctopleft.x;
  double max = doctopleft.x + getDocument().getDocumentSize().width;

  double force = 300.0;
```

```
  if (xy.x <= min)
    return force;
  else if (xy.x <= min + border)
    return (force / ((min - xy.x) * (min - xy.x)));

  if (xy.x >= max)
    return -force;
  else if (xy.x >= max - border)
    return (-force / ((xy.x - max) * (xy.x - max)));

  return 0.0;
}

public double getElectricalFieldY(Point xy)
{
  double border = 200.0;
  Point doctopleft = getDocument().getDocumentTopLeft();
  double min = 0; // doctopleft.y;
  double max = doctopleft.y + getDocument().getDocumentSize().height;

  double force = 300.0;
  if (xy.y <= min)
    return force;
  else if (xy.y <= min + border)
    return (force / ((min - xy.y) * (min - xy.y)));

  if (xy.y >= max)
    return -force;
  else if (xy.y >= max - border)
    return (-force / ((xy.y - max) * (xy.y - max)));

  return 0.0;
}
```

However, if there are too many nodes for the size of the box (here determined by getDocumentSize()), there may be too much “pressure” and some nodes may “explode” out of the box.

By adjusting the values of the springLength and springStiffness, one can achieve a number of sophisticated results. For example, by increasing the springLength between red and green nodes, it is possible to group the nodes by color as illustrated in Figure 16. Keep in mind that the colors of nodes are part of the LayoutDemo application, and not a part of the JGo Layout code itself.

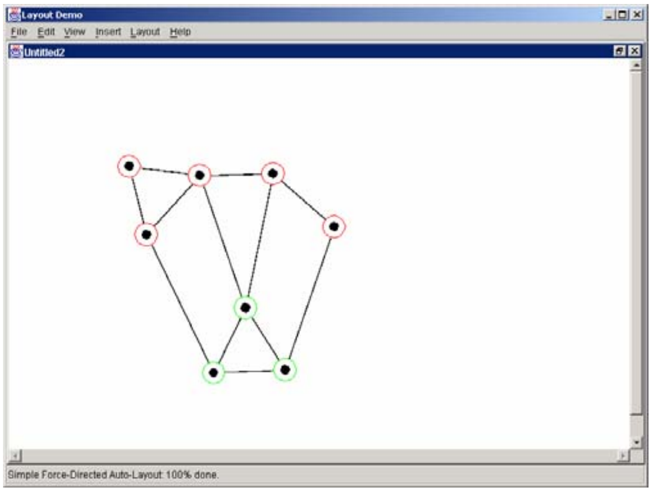


Figure 16. Sample graph after adjusting spring length and thickness

You can use the gravitational field values to influence the layout of tree-like networks. For example, consider the following two networks:

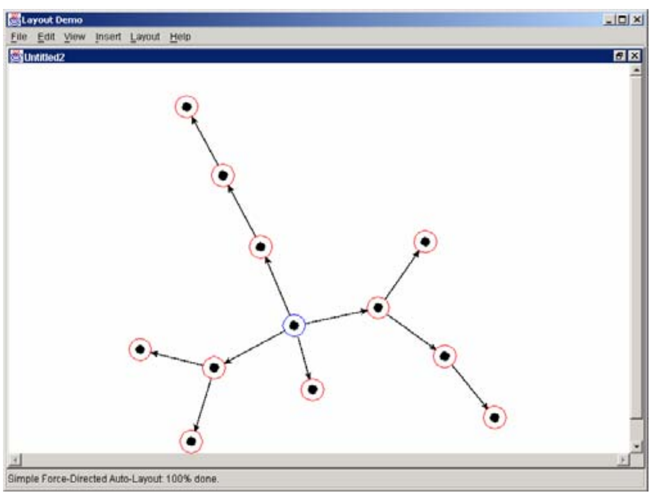


Figure 17. Sample graph before applying gravity field

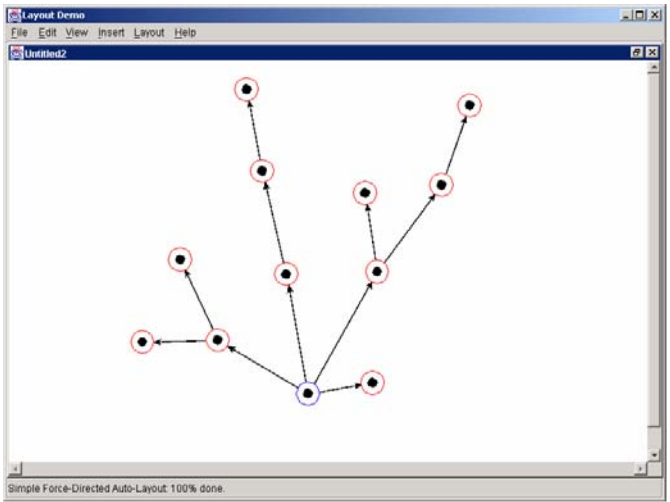


Figure 18. Sample graph after applying gravity field

In both networks, the blue root node is fixed. In the network of Figure 17, no gravitational field has been set. In the network of Figure 18, a slight gravitational field pointing upward has been added, which results in a more natural layout for a tree.

JGoForceDirectedAutoLayout has two other methods that can be overridden:

```
protected boolean updatePositions()
protected void layoutNodesAndLinks(boolean final)
```

The updatePositions method is used each iteration to calculate the forces on each node and to move the node to its new position; it returns true if additional iterations are needed to find a local equilibrium. Overriding the updatePositions method is not recommended, but could be used to add new forces to the layout.

The layoutNodesAndLinks method is used to update the physical locations of the "real" nodes on the screen to reflect the layout. By default, the layoutNodesAndLinks method redraws the screen every 10 iterations. One reason to override this method would be to decrease the frequency of screen redraws, which would decrease the time used to find a local equilibrium.

JGoLayeredDigraphAutoLayout

Most of the customization available in the JGoLayeredDigraphAutoLayout class is accessed through the class constructors:

```
JGoLayeredDigraphAutoLayout(JGoDocument doc,
int NlayerSpacing, int NcolumnSpacing,
int NdirectionOption, int NcycleremoveOption,
int NlayeringOption, int NinitializeOption,
int Niterations, int NaggressiveOption)

JGoLayeredDigraphAutoLayout(JGoDocument doc, JGoNetwork network,
int NlayerSpacing, int NcolumnSpacing,
int NdirectionOption, int NcycleremoveOption,
int NlayeringOption, int NinitializeOption,
int Niterations, int NaggressiveOption)
```

See the JGo AutoLayout Class Reference Guide for a detailed description of these parameters. The NlayerSpacing and NcolumnSpacing parameters determine the minimum space (in logical units) between nodes in adjacent layers and columns. Generally, since nodes have width and height, additional space is reserved around nodes. However, NcolumnSpacing will determine the minimum space between long links that are drawn parallel and adjacent to one another, as illustrated in Figure 19.

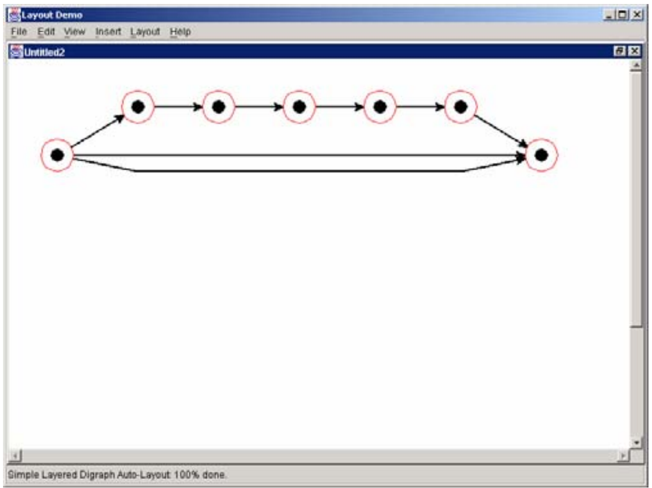


Figure 19. A graph showing the use of NcolumnSpacing

The Niterations option determines the number of sweeps used during the Crossing Reduction step. Experience has shown that values above 8 almost never affect the final drawing of the network.

The JGoLayeredDigraphAutoLayout also has a number of methods that can be overridden. These can generally be divided into three categories. The first category of methods override principle steps of the layered-digraph routine:

```
protected void removeCycles()
protected void assignLayers()
protected void makeProper()
protected void initializeIndices()
protected void initializeColumns()
protected void reduceCrossings()
protected void straightenAndPack()
protected void layoutNodesAndLinks()
```

These methods can be overridden to customize the layout algorithm, but care should be taken to ensure proper initialization and termination of each method. There is little reason to override most of these methods, since particular cycle removal, layering, and initialization routines can be specified through the constructor. However, one may wish to override the layoutNodesAndLinks method in order to take advantage of the added functionality of sub-classes of JGoLink; for example, a sub-class that tracked bend points and allowed them to be repositioned by the application.

The second category of methods override spacing methods:

```
protected int getNodeMinLayerSpace(JGoNetworkNode pNode)
protected int getNodeMinColumnSpace(JGoNetworkNode pNode)
```

These methods determine the minimum number of layers and columns to be reserved around the center point of a node. This allows a node to be positioned by its layer and column, but ensures that two nodes do not overlap in the final drawing. The default implementations of these functions return 0 for nodes that do not correspond to top-level JGo objects. For nodes that do correspond to top-level JGo objects, the width and height of the object determine the space. One may wish to override these methods if there are nodes in the network whose spacing needs cannot be accurately determined from the width and height of the JGo object; for example, nodes which will later have significant text fields associated with them.

The final category of methods override layering methods:

```
protected int getLinkMinLength(JGoNetworkLink pLink)
protected double getLinkLengthWeight(JGoNetworkLink pLink)
```

The getLinkMinLength method indicates the minimum length of the link. For example, if link L = (U,V), then layer(U) – layer(V) ≥ getLinkMinLength(L). The default implementation gives multi-links (multiple links between the same pairs of nodes) a minimum length of 2, and all other links a minimum length of 1. This ensures that multi-links are drawn distinctly, illustrated in Figure 20.

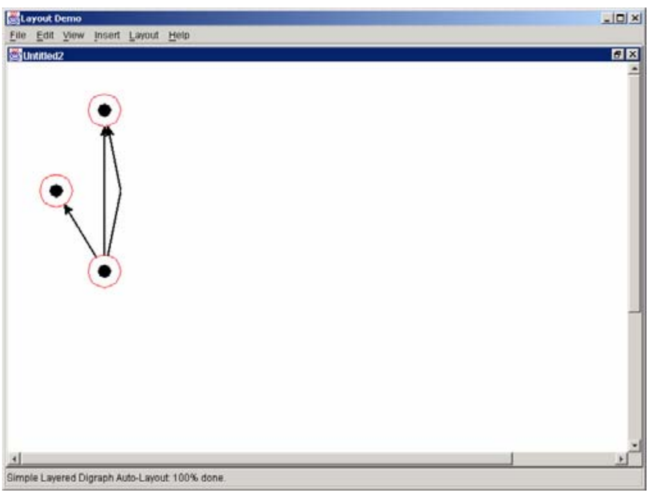


Figure 20. Example use of the linkMinLength method

The Layout Demo sample application overrides the getLinkMinLength method as follows:

```
int getLinkMinLength(JGoNetworkLink pLink)
{
  JGoNetworkNode pFromNode = pLink.getFromNode();
  JGoNetworkNode pToNode = pLink.getToNode();

  if ((pFromNode.getGoObject() != null) &&
       (pToNode.getGoObject() != null)) {
      Color fromColor =
        ((BasicNode)(pFromNode.getGoObject())).getColor();
      Color toColor =
        ((BasicNode)(pToNode.getGoObject())).getColor();

      if (fromColor == toColor) {
          return 1 * super.getLinkMinLength(pLink);
      } else {
          return 2 * super.getLinkMinLength(pLink);
      }
  }

  return super.getLinkMinLength(pLink);
}
```

This automatically doubles the length of the links between nodes of different colors:

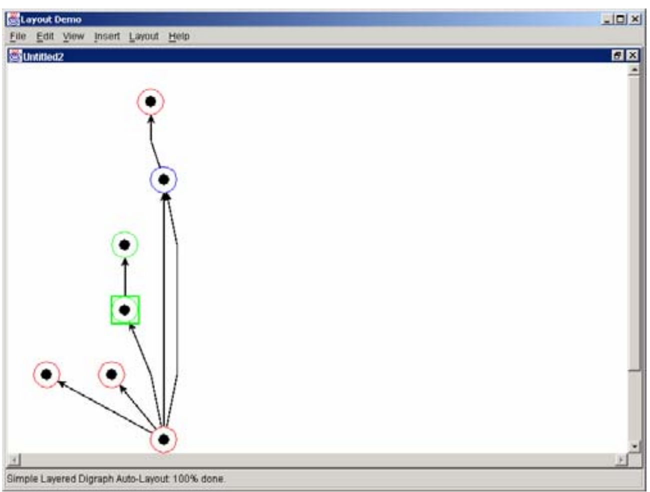


Figure 21. Another example use of the getLinkMinLength method

The getLinkLengthWeight method indicates the weight of the link. The Optimal Link Length Layering routine assigns nodes to layers such that the sum (layer(U) – layer(V)) * getLinkLengthWeight(L) over all L = (U,V) is minimized. By default, all links have a linkLengthWeight of 1.0. The getLinkLengthWeight method can be overridden to increase the "importance" of a link, which means the link will be kept shorter. For example, compare the networks of Figure 22 and Figure 23.

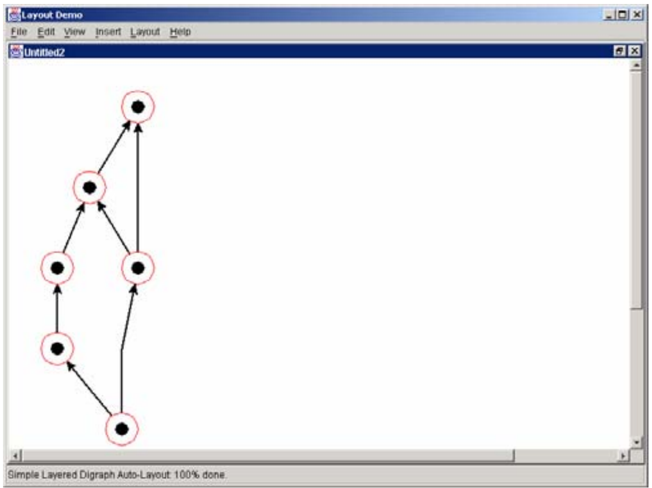


Figure 22. Graph before using getLinkLengthWeight

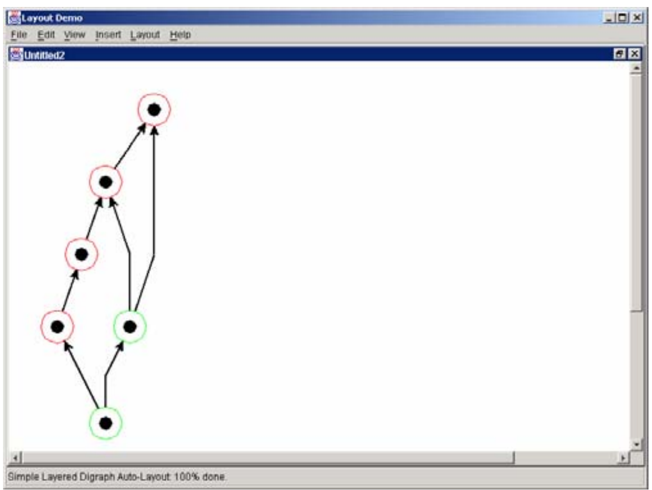


Figure 23. Graph after using getLinkLengthWeight

In both networks, the linkLengthWeight of a link between nodes of the same color is five times the linkLengthWeight of a link between nodes of different colors. Note that in the network on the right, the higher weight of the link between the two green nodes resulted in a shorter link, at the expense of lengthening two links of lesser weight.

AutoLayout and SubGraphs

The autolayout algorithms do not modify the contents of any JGoArea. This is almost always what you would want, except when the area is a JGoSubGraph. If you want to do an autolayout of the children of a JGoSubGraph, you will need to do this recursively. If you program it in a depth-first fashion, you will layout the subgraphs first, thereby changing their sizes. Then the layout can proceed on the graph that includes the JGoSubGraph nodes.

Demo1 includes an example of constructing a JGoNetwork for laying out a JGoSubGraph.