ILOG JViews 5.5

Graph Layout User's Manual

(Excerpt)

December 2002

Table of Contents

About This Manual

The ILOG JViews Component Suite provides special support for applications that need to display graphs, or networks, of nodes and links. Any graphic object can be defined to behave like a node and can be connected to other nodes via links, which themselves can have many different forms. Used in conjunction with layout algorithms, the ILOG JViews grapher is often used to display network topologies for telecommunications networks and systems management applications.

What Is in This Manual

The ILOG JViews Graph Layout module provides high-level, ready-to-use graph drawing services that allow you to obtain readable representations easily.

This manual contains the following chapters:

◆ Chapter 1, *Introducing the Graph Layout Module* describes the Graph Layout module of ILOG JViews and its features.

◆ Chapter 2, *Basic Concepts* provides background information and basic concepts for using Graph Layout.

◆ Chapter 3, *Getting Started with Graph Layout* provides information to get started quickly using Graph Layout.

- Chapter 4, *Layout Algorithms* describes the layout algorithms provided with the Graph Layout module.
- Chapter 5, *Using Advanced Features* provides information on using a layout report, using layout event listeners, using a graph model, laying out a non-JViews grapher, layout out disconnected graphs, saving layout parameters to a file, laying out a portion of a graph, laying out graphs with nonzoomable objects, and defining new types of layouts.
- Chapter 6, *Automatic Label Placement* describes the label layout facilities provided with the Graph Layout module.
- Chapter 7, *Using Graph Layout Beans* shows you how to use the ILOG JViews Beans and the Graph Layout Beans when creating an applet within an Integrated Development Environment (IDE).

At the end of the manual you will find a *Glossary* containing definitions of the basic technical terms used in this manual.

Related Documentation

The following documentation may provide helpful information when using ILOG JViews Graph Layout.

Books

The first book dedicated to graph layout has been published:

Di Battista, Giuseppe, Peter Eades, Roberto Tammassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*, Prentice Hall, 1999 (see `http://www.cs.brown.edu/people/rt/gdbook.html` or `http://www.prenhall.com/books/esm_0133016153.html`).

Graph layout is closely related to graph theory, for which extensive literature exists. See:

Clark, John and Derek Allan Holton. *A First Look at Graph Theory.* World Scientific Publishing Company, 1991.

For a mathematics-oriented introduction to graph theory, see:

Diestel,Reinhard, *Graph Theory*, 2nd ed. Springer-Verlag, 2000.

A more algorithmic approach may be found in:

Gibbons, Alan. *Algorithmic Graph Theory.* Cambridge University Press, 1985.

Gondran, Michel and Michel Minoux. *Graphes et algorithmes*, 3rd ed. Eyrolles, Paris, 1995 (in French).

Bibliographies

A comprehensive bibliographic database of papers in computational geometry (including graph layout) can be found:

The Geometry Literature Database (`http://compgeom.cs.uiuc.edu/~jeffe/compgeom/biblios.html`)

The recommended bibliographic survey paper is the following:

Di Battista, Giuseppe, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. “Algorithms for Drawing Graphs: an Annotated Bibliography.” *Computational Geometry: Theory and Applications* 4 (1994): 235-282 (also available at `http://www.cs.brown.edu/people/rt/gd-biblio.html`).

Journals

The following are electronic journals:

Journal of Graph Algorithms and Applications (`http://www.cs.brown.edu/publications/jgaa`)

Algorithmica (`http://link.springer-ny.com/link/service/journals/00453/`)

Computational Geometry: Theory and Applications (`http://www.elsevier.nl/inca/publications/store/5/0/5/6/2/9/`)

Journal of Visual Languages and Computing (`http://www.academicpress.com/jvlc`)

The following journals occasionally publish papers on graph layout:

Information Processing Letters (`http://www.elsevier.nl/inca/publications/store/5/0/5/6/1/2/`)

Computer-aided Design (`http://www.elsevier.nl/inca/publications/store/3/0/4/0/2/`)

IEEE Transactions on Software Engineering (`http://www.computer.org/tse/`)

Many papers are presented at conferences in Combinatorics and Computer Science.

Conferences

An annual Symposium on Graph Drawing has been held since 1992. The proceedings are published by Springer-Verlag in the *Lecture Notes in Computer Science* series. For the 2001 edition, see `http://link.springer.de/link/service/series/0558/tocs/t2265.htm`. For the 2003 Symposium, to be held in Perugia, Italy, see `http://www.gd2003.org`.

Introducing the Graph Layout Module

This chapter introduces you to the Graph Layout module of ILOG JViews. The following topics are covered:

- *What is the Graph Layout Module of ILOG JViews?*
- *Composition of the ILOG JViews Graph Layout Module*
- *The ILOG JViews Graph Layout Algorithms*
- *Features of the ILOG JViews Graph Layout Module*
- *ILOG JViews Graph Layout Module in User Interface Applications*

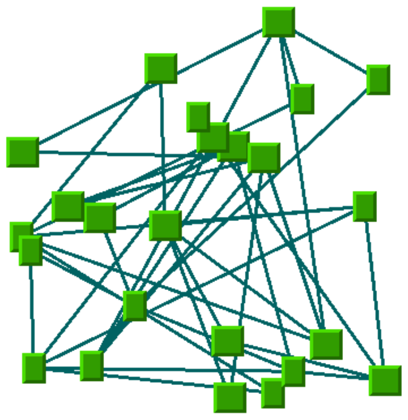
What is the Graph Layout Module of ILOG JViews?

Many types of complex business data can best be visualized as a set of nodes and interconnecting links, more commonly called a graph or a network. Examples of graphs include business organizational charts, workflow diagrams, telecom network displays, and genealogical trees. Whenever these graphs become large or heavily interconnected, it becomes difficult to see the relationships between the various nodes and links (the “edges”). This is where ILOG JViews Graph Layout algorithms help.

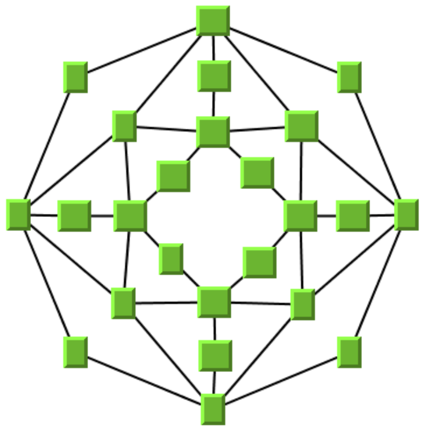
The ILOG JViews Graph Layout module provides high-level, ready-to-use relationship visualization services. It allows you to take any “messy” graph and apply a sophisticated graph layout algorithm to rearrange the positions of the nodes and links. The result is a more readable and understandable picture.

Take a look at two sample drawings of the same graph.

Here no format layout algorithm was used. The nodes were placed randomly when the graph was drawn.



Using one of the layout algorithms provided in ILOG JViews, the following drawing was obtained:



In the second drawing, the layout algorithm has distributed the nodes quite uniformly, avoiding overlapping nodes and showing the symmetries of the graph. This drawing presents a much more readable layout than does the first drawing.

Composition of the ILOG JViews Graph Layout Module

The ILOG JViews Graph Layout module is composed of the following packages:

- `ilog.views.graphlayout`: A high-level, generic framework for the graph layout services provided by ILOG JViews.

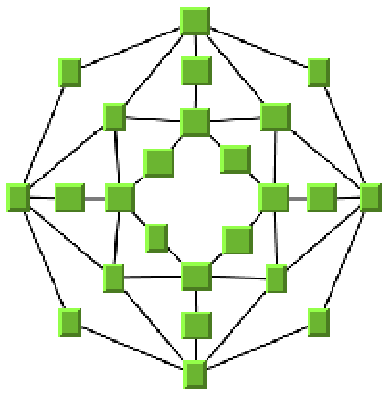
- Layout algorithm packages:
  - `ilog.views.graphlayout.bus`: A layout algorithm designed to display bus network topologies (that is, a set of nodes connected to a bus node).
  - `ilog.views.graphlayout.circular`: A layout algorithm that displays graphs representing interconnected ring and/or star network topologies.
  - `ilog.views.graphlayout.grid`: A layout algorithm that arranges the disconnected nodes of a graph in rows, in columns, or in the cells of a grid.
  - `ilog.views.graphlayout.hierarchical`: A layout algorithm that arranges nodes in horizontal or vertical levels such that the links flow in a uniform direction.
  - `ilog.views.graphlayout.link`: A layout algorithm that reshapes the links of a graph without moving the nodes.

    `ilog.views.graphlayout.link.longlink`: For long orthogonal links.
    `ilog.views.graphlayout.link.shortlink`: For short links.
  - `ilog.views.graphlayout.multiple`: A facility that combines multiple layout algorithms into one graphic object.
  - `ilog.views.graphlayout.random`: A layout algorithm that moves the nodes of the graph at randomly computed positions inside an user-defined region.
  - `ilog.views.graphlayout.recursive`: A layout algorithm that can be used to control the layout of nested graphs (containing subgraphs and intergraph links).
  - `ilog.views.graphlayout.springembedder`: A layout algorithm that can be used to lay out any type of graph.
  - `ilog.views.graphlayout.topologicalmesh`: A layout algorithm that can be used to lay out cyclic graphs.
  - `ilog.views.graphlayout.tree`: A layout algorithm that arranges the nodes of a tree horizontally or vertically, starting from the root of the tree. A radial layout mode allows you to arrange the nodes of a tree on concentric circles around the root of the tree.
  - `ilog.views.graphlayout.uniformlengthedges`: A layout algorithm that can be used to lay out any type of graph and allows you to specify the length of the links.
- `ilog.views.graphlayout.labellayout`: A layout algorithm for automatic placement of labels.
  - `ilog.views.graphlayout.labellayout.annealing`: For close label positioning.
  - `ilog.views.graphlayout.labellayout.random`: For random placement.
- `ilog.views.graphlayout.swing`: Swing components useful for creating applications mixing ILOG JViews Graph Layout and Swing.

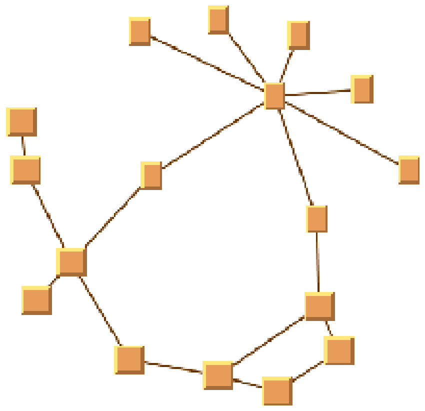
The ILOG JViews Graph Layout Algorithms

The Graph Layout module of ILOG JViews provides numerous ready-to-use layout algorithms. They are shown below with sample illustrations. In addition, new layout algorithms can be developed using the generic layout framework of the ILOG JViews Graph Layout module.

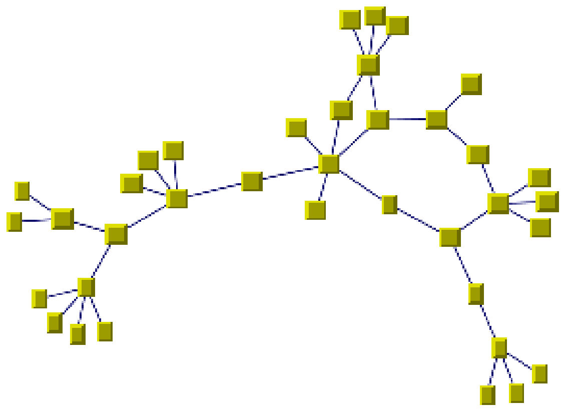
Topological Mesh Layout (TML)



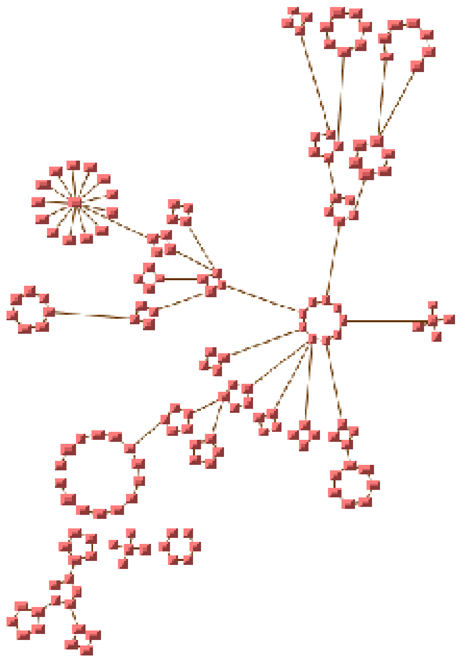
Spring Embedder Layout



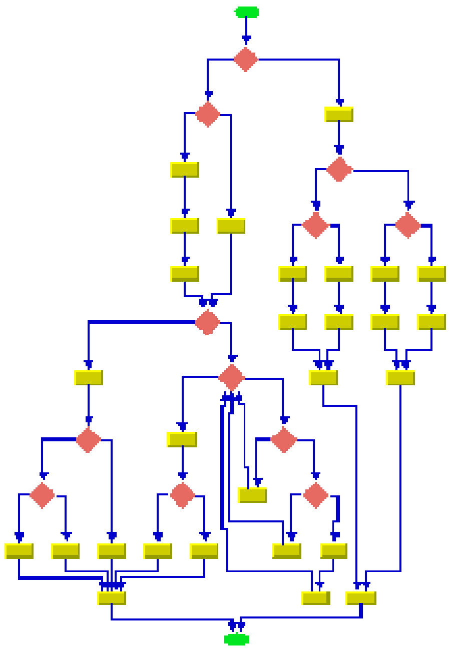
Uniform Length Edges Layout



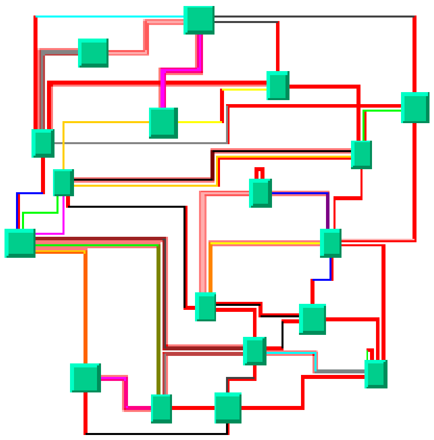
Circular Layout (Ring/Star)



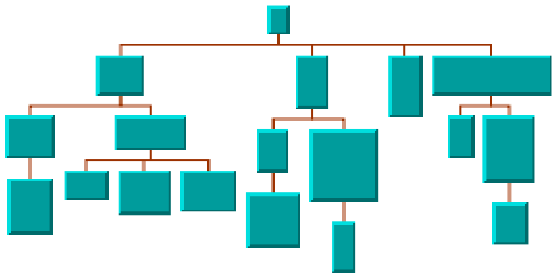
Hierarchical Layout



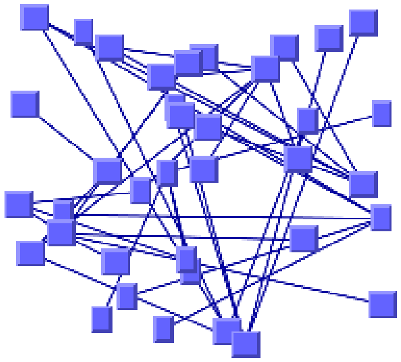
Link Layout



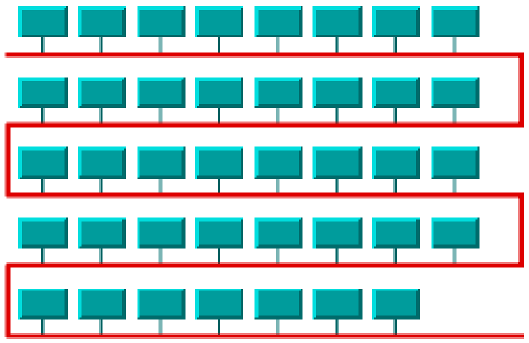
Tree Layout



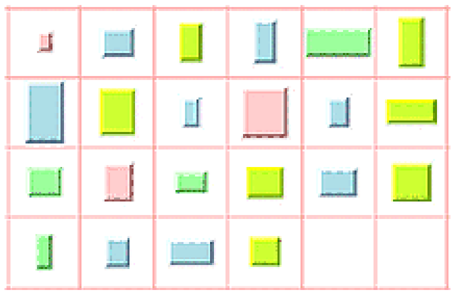
Random Layout



Bus Layout



Grid Layout



Features of the ILOG JViews Graph Layout Module

The Graph Layout module provides the following features for using the layout algorithms. (Note that some of these features are not supported by all the algorithms provided with ILOG JViews.)

- ◆ Capability to fit the layout into a manager view or a user-defined rectangle
- ◆ Capability to take into account the size of the nodes when performing the layout to avoid overlapping nodes
- ◆ Capability to perform the layout using only the nodes and links that are on user-defined layers of the graph
- ◆ Capability to perform the layout only on those parts of the graph that meet user-defined conditions
- ◆ Capability to use non-JViews graphers
- ◆ Layout reports providing information concerning the behavior of the layout algorithm
- ◆ Layout-event listeners that can receive and report information during the graph layout
- ◆ A generic framework for customizing the layout algorithms. The following generic features and parameters are defined. (Note that not all the layout algorithms provided with ILOG JViews support all these parameters. Whether a generic parameter is supported depends on the particular layout algorithm.)
  - Allowed Time

    This parameter allows the layout algorithm to stop computation when a user-defined time specification is exceeded.
  - Animation

    This parameter allows the layout algorithm to redraw the graph after each iteration or step.

- Fixed Nodes

  This parameter allows the layout algorithm to preserve the location of the specified nodes. Certain nodes can be specified as fixed and will not be moved when the layout is performed. The layout algorithm can "pin" specified nodes in place.

- Fixed Links

  This parameter allows the layout algorithm to preserve the shape of the specified links. Certain links can be specified as fixed and will not be reshaped when the layout is performed. The layout algorithm can "pin" specified links in place.

- Filtering

  The layout algorithms are able to perform the layout using only the nodes and links that are on user-defined layers of the grapher, or to exclude nodes and links on an individual basis.

- Layout of Connected Components

  This parameter allows you to automatically lay out the connected components of a disconnected graph.

- Layout Region

  This parameter allows the layout algorithm to control the size of the graph drawing.

- Percentage Completion Calculation

  This parameter allows the layout algorithm to provide an estimation of how much of the layout has been completed.

- Random Generator Seed Value

  This parameter allows the layout algorithm to use randomly generated numbers that can be initialized with a user-defined seed value. These seed values are then used during layout computations to produce different layouts of the graph.

- Save Parameters to Named Properties

  This parameter provides support for saving the layout parameters as named properties in `.ivl` files.

- Stop Immediately

  This parameter allows the layout algorithm to stop layout computation immediately when an outside event occurs.

- Use Default Parameters

  This parameter allows the layout algorithm to return to using default parameter settings after the default settings have been modified.

ILOG JViews Graph Layout Module in User Interface Applications

Many fields use graph drawings and graph layouts in user interface applications. Therefore, the ILOG JViews Graph Layout module is particularly well-suited for these kinds of applications. The following is a list of some of the fields where the graph layout capabilities of the ILOG JViews Graph Layout module can be used:

- **Telecom and Networking**
  - LAN Diagrams (*Bus Layout, Circular Layout*)
  - WAN Diagrams (*Spring Embedder, Uniform Edge Length*)
- **Electric Engineering**
  - Logic Diagrams (*Hierarchical Layout*)
  - Circuit Block Diagrams (*Hierarchical Layout*, *Link Layout*, *Bus Layout*)
- **Industrial Engineering**
  - Industrial Process Charts (*Hierarchical Layout*)
  - Schematic Design Diagrams (*Link Layout*, *Hierarchical Layout*)
  - Equipment/Resource Control Charts (*Bus Layout*, *Link Layout*)
- **Business Processing**
  - Workflow Diagrams (*Hierarchical Layout*)
  - Process Flow Diagrams (*Hierarchical Layout*)
  - Organization Charts (*Tree Layout*, *Circular Layout*)
  - Entity Relation Diagrams (*Link Layout*)
  - PERT Charts (*Hierarchical Layout*)
- **Software Management/Software (Re-)Engineering**
  - UML Diagrams (*Hierarchical Layout*, *Tree Layout*)
  - Flow Charts (*Hierarchical Layout*)
  - Data Inspector Diagrams (*Link Layout*, *Hierarchical Layout*)
  - Call Graphs (*Spring Embedder, Uniform Edge Length Layout*, *Hierarchical Layout, Tree Layout*)
- **CASE Tools**
  - Design Diagrams (*Link Layout, Hierarchical Layout*)
  - Dependency Diagrams (*Spring Embedder, Uniform Edge Length Layout*)

- **Data Base and Knowledge Engineering**
  - Semantic Networks (*Uniform Edge Length Layout, Spring Embedder, Topological Mesh Layout*)
  - Decision Trees (*Tree Layout*)
  - Database Query Graphs (*Spring Embedder, Uniform Edge Length Layout, Hierarchical Layout*)
  - Qualitative Reasoning and other Artificial Intelligence Diagrams (*Topological Mesh Layout, Spring/Uniform Edge Length Layout*)
- **The World Wide Web**
  - Web Site Maps (*Tree Layout*)
  - Web Hyperlink Neighborhood (*Spring Embedder, Uniform Edge Length Layout, Circular Layout*)

C H A P T E R

2

Basic Concepts

In this chapter, you will learn about some basic concepts and background information that will help you when using the ILOG JViews Graph Layout module. The following topics are covered:

- *Graph Layout: A Brief Introduction*
- *Graph Layout in ILOG JViews*
- *The Base Class: IlvGraphLayout*
- *Basic Operations with IlvGraphLayout*
- *Layout Parameters and Features in IlvGraphLayout*

Graph Layout: A Brief Introduction

This section provides some background information about graph layout in general, not specifically related to the ILOG JViews Graph Layout module.

Simply speaking, a graph is a data structure which represents a set of entities, called nodes, connected by a set of links. (A node can also be referred to as a vertex. A link can also be referred to as an edge or a connection.) In practical applications, graphs are frequently used to model a very wide range of things: computer networks, software program structures, project management diagrams, and so on. Graphs are powerful models because they permit applications to benefit from the results of graph theory research. For instance, efficient methods are available for finding the shortest path between two nodes, the minimum cost path, and so on.

Graph layout is used in graphical user interfaces of applications that need to display graph models. To lay out a graph means to draw the graph so that an appropriate, readable representation is produced. Essentially, this involves determining the location of the nodes and the shape of the links. For some applications, the location of the nodes may be already known (based on the geographical positions of the nodes, for example). However, for other applications, the location is not known (a pure “logical” graph) or the known location, if used, would produce an unreadable drawing of the graph. In these cases, the location of the nodes must be computed.

But what is meant by an “appropriate” drawing of a graph? In practical applications, it is often necessary for the graph drawing to respect certain quality criteria. These criteria may vary depending on the application field or on a given standard of representation. It is often difficult to speak about what a good layout consists of. Each end user may have different, subjective criteria for qualifying a layout as “good”. However, one common goal exists behind all the criteria and standards: the drawing must be easy to understand and provide easy navigation through the complex structure of the graph.

These topics are pursued in the following sections:

◆ *What is a Good Layout?*

◆ *Methods for Using Layout Algorithms*

What is a Good Layout?

To deal with the various needs of different applications, many classes of graph layout algorithms have been developed. A layout algorithm addresses one or more quality criteria, depending on the type of graph and the features of the algorithm, when laying out a graph. The most common criteria are:

◆ Minimizing the number of link crossings

◆ Minimizing the total **area** of the drawing

- Minimizing the number of **bends** (in orthogonal drawings)
- Maximizing the smallest **angle** formed by consecutive incident links
- Maximizing the display of **symmetries**

How can a layout algorithm meet each of these quality criteria and standards of representation? If you look at each individual criteria, some can be met quite easily, at least for some classes of graphs. For other classes, it may be quite difficult to produce a drawing that meets the criteria. For example, minimizing the number of link crossings is relatively simple for trees (that is, graphs without cycles). However, for general graphs, minimizing the number of link crossings is a mathematical NP-complete problem (that is, with all known algorithms, the time required to perform the layout grows very fast with the size of the graph.)

Moreover, if you want to meet several criteria at the same time, an optimal solution simply may not exist with respect to each individual criteria because many of the criteria are mutually contradictory. Time-consuming trade-offs may be necessary. In addition, it is not a trivial task to assign weights to each criteria. Multicriteria optimization is, in most cases, too complex to implement and much too time-consuming. For these reasons, layout algorithms are often based on heuristics and may provide less than optimal solutions with respect to one or more of the criteria. Fortunately, in practical terms, the layout algorithms will still often provide reasonably readable drawings.

Methods for Using Layout Algorithms

Layout algorithms can be employed in a variety of ways in the various applications in which they are used. The most common ways of using an algorithm are the following:

- **Automatic layout**

  The layout algorithm does everything without any user intervention, except perhaps the choice of the layout algorithm to be used. Sometimes a set or rules can be coded to choose automatically (and dynamically) the most appropriate layout algorithm for the particular type of graph being laid out.

- **Semiautomatic layout**

  The end user is free to improve the result of the automatic layout procedure by hand. At times the end user can move and “pin” nodes at desired locations and perform the layout again. In other cases, a part of the graph is automatically set as “read-only” and the end user can modify the rest of the layout.

- **Static layout**

  The layout algorithm is completely redone (“from scratch”) each time the graph is changed.

- **Incremental layout**

  When the layout algorithm is performed a second time on a modified graph, it tries to preserve the stability of the layout as much as possible. The layout is not performed again from scratch. The layout algorithm also tries to economize CPU time by using the previous layout as an initial solution. Some layout algorithms and layout styles are incremental by nature. For others, incremental layout may be impossible.

Graph Layout in ILOG JViews

In ILOG JViews, graphs are instances of the class `IlvGrapher`. We call these instances *graphers*. The nodes, which are instances of `IlvGraphic`, and the links, which are instances of `IlvLinkImage`, "know" how to draw themselves. The nodes can have arbitrary coordinates or they can be "placed" interactively or by code. All that needs to be done to lay out the grapher in order to obtain a readable drawing is to compute and to assign appropriate coordinates for the nodes. In some cases, the shape of the links may also need to be modified. The main task of the Graph Layout module is to provide support for the operation of laying out a grapher—that is, drawing the graph.

The Graph Layout module of ILOG JViews benefits from its integration with the graph visualization and graph manipulation features of the ILOG JViews core library. This core library provides a wide range of very useful features to build powerful graphic interfaces easily:

- Predefined, extensible types of graphic objects for nodes and links
- A customizable mechanism to choose the contact points between links and nodes
- Grapher interactor classes
- Multiple views of the same grapher
- Management of multiple layers
- Support for nested graphs
- Selections management
- Events management
- Listeners on the contents of the grapher and/or on the views
- Printing facilities
- User-defined properties on nodes and links
- Input/output operations

For details on these features, see the *ILOG JViews Graphics Framework User's Manual*.

> ***Note:*** *The Graph Layout module allows you to add layout capabilities to applications that do not use the ILOG JViews grapher. For details, see Laying Out a Non-JViews Grapher on page 303.*

The Base Class: IlvGraphLayout

The `IlvGraphLayout` class is the base class for all layout algorithms. This class is an abstract class and cannot be used directly. You must use one of its subclasses (`IlvTopologicalMeshLayout`, `IlvSpringEmbedderLayout`, `IlvUniformLengthEdgesLayout`, `IlvTreeLayout`, `IlvHierarchicalLayout`, `IlvLinkLayout`, `IlvRandomLayout`, `IlvBusLayout`, `IlvCircularLayout`, `IlvGridLayout`). You can also create your own subclasses to implement other layout algorithms.

Despite the fact that only subclasses of `IlvGraphLayout` are directly used to obtain the layouts, it is still necessary to learn about this class because it contains methods that are inherited (or overridden) by the subclasses. And, of course, you will need to understand it if you subclass it yourself.

You can find more information about the class `IlvGraphLayout` in the following sections:

- *Basic Operations with IlvGraphLayout* on page 17 tells you about the basic methods you need using the subclasses of `IlvGraphLayout`.
- *Layout Parameters and Features in IlvGraphLayout* on page 20 contains the methods that are related to the customization of the layout algorithms.
- *Using Event Listeners* on page 294 tells you about the layout event listener mechanism.
- *Defining a New Type of Layout* on page 328 tells you how to implement new subclasses.

For details on `IlvGraphLayout` and other graph layout classes, see the *ILOG JViews Graph Layout Reference Manual*.

Basic Operations with IlvGraphLayout

When subclassing `IlvGraphLayout` or when using one of its subclasses, you will normally use the basic methods described in the following sections:

- *Instantiating a Subclass of IlvGraphLayout*
- *Attaching a Grapher*

- *Performing a Layout*
- *Detaching a Grapher*

Instantiating a Subclass of IlvGraphLayout

The class `IlvGraphLayout` is an abstract class. It has no constructors. You will instantiate a subclass as shown in the following example:

```
IlvLinkLayout layout = new IlvLinkLayout();
```

If you want to use the layout report that is returned by the layout algorithm, you need to declare a handle for the appropriate layout report class, as shown in the following example:

```
IlvGraphLayoutReport layoutReport;
```

For more information on the layout report, see *Using a Layout Report* on page 292.

Attaching a Grapher

You must attach the grapher before performing the layout. The following method, defined on the class `IlvGraphLayout`, allows you to specify the grapher you want to lay out:

```
   void attach(IlvGrapher grapher)
```

For example:

```
...
IlvGrapher grapher = new IlvGrapher();
/* Add nodes and links to the grapher here */
layout.attach(grapher);
```

The `attach` method does nothing if the specified grapher is already attached. Otherwise, it first detaches the grapher that is already attached, if any. You can obtain the attached grapher using the method `getGrapher()`. If the grapher is attached in this way, a default graph model is created internally. For details on the graph model, see *Using the Graph Model* on page 297. The attached graph model can be obtained by:

```
IlvGraphModel graphModel = layout.getGraphModel();
```

***Warning:** Notice that such an internally created model is not allowed to be attached to any other layout instance, nor to be used in any way once it has been detached from the layout instance. For details, see Using the IlvGrapherAdapter on page 302.*

After layout, when you no longer need the layout instance, you should call the `detach` method. If the `detach` method is not called, some objects may not be garbage-collected. This method also performs cleaning operations on the grapher (properties that may have been added by the layout algorithm on the grapher's objects are removed).

```
void detach()
```

Performing a Layout

The `performLayout` method starts the layout algorithm using the currently attached grapher and the current settings for the layout parameters. The method returns a report object that contains information about the behavior of the layout algorithm.

```
IlvGraphLayoutReport performLayout()
IlvGraphLayoutReport performLayout(boolean force, boolean redraw)
```

The first version of the method simply calls the second one with a `false` value for the first argument and a `true` value for the second argument. If the argument `force` is `false`, the layout algorithm first verifies whether it is necessary to perform the layout. It checks internal flags to see whether the grapher or any of the parameters have been changed since the last time the layout was successfully performed. A “change” can be any of the following:

- Nodes or links were added or removed.
- Nodes or links were moved or reshaped.
- The value of a layout parameter was modified.
- The transformer of a manager view (`IlvManagerView`) of the grapher has changed.

Users often do not want the layout to be computed again if no changes occurred. If there were no changes, the method `performLayout` returns without performing the layout. Note that if the argument `force` is passed as `true`, the verification is no longer performed.

The argument `redraw` determines whether a redraw of the graph is requested. For details, see *Redrawing the Grapher after Layout* on page 296.

The protected abstract method `layout(boolean redraw)` is then called. This means that the control is passed to the subclasses that are implementing this method. The implementation computes the layout and moves the nodes to new positions and/or reshapes the links.

The `performLayout` method returns an instance of `IlvGraphLayoutReport` (or of a subclass) that contains information about the behavior of the layout algorithm. It tells you whether the algorithm performed normally, or whether a particular, predefined case occurred. (For a more detailed description of the layout report, see *Using a Layout Report* on page 292.)

Note that the layout report that is returned can be an instance of a subclass of `IlvGraphLayoutReport` depending on the particular subclass of `IlvGraphLayout` you are using. For example, it will be an instance of `IlvTopologicalMeshLayoutReport` if you are using the class `IlvTopologicalMeshLayout`. Subclasses of `IlvGraphLayoutReport` are used to store layout algorithm-dependent information.

You must call the method `performLayout` inside a `try` block because it can throw an exception. The exception can be of the type `IlvGraphLayoutException` or one of its subclasses, `IlvInappropriateGraphException` and `IlvInappropriateLinkException`. The first indicates internal problems in the layout algorithm or an unexpected situation. The second exception indicates that a particular grapher cannot be laid out with the layout algorithm. For example, the Topological Mesh Layout cannot be used on an acyclic graph (that is, a pure tree). The third exception indicates that a particular type of link (or link connector) cannot be used for this layout. The recommended type of link is `IlvPolylineLinkImage` or `IlvSplineLinkImage` (or subclasses). For layouts that do not reshape the links by adding intermediate points, the class `IlvLinkImage` can also be used.

Detaching a Grapher

You call the `detach` method when you no longer need the layout instance of an attached grapher. If the `detach` method is not called, some objects may not be garbage-collected. This method also performs cleaning operations on the grapher (properties that may have been added by the layout algorithm on the grapher's objects are removed).

```
void detach()
```

C H A P T E R

3

Getting Started with Graph Layout

This chapter provides information to get started using the Graph Layout module of ILOG JViews. The following topics are covered:

- *Basic Steps for Using Layout Algorithms: A Summary*
- *Sample Java Application*

Basic Steps for Using Layout Algorithms: A Summary

To use the layout algorithms provided by the Graph Layout module of ILOG JViews, you will usually perform the following steps:

1. Create a grapher object (`IlvGrapher`) and fill it with nodes and links.
2. Create an instance of the layout algorithm (any subclass of `ilog.views.graphlayout.IlvGraphLayout`).
3. Declare a handle for the corresponding layout report. The layout report is an object in which the layout algorithm stores information about its behavior. For details see *Using a Layout Report* on page 292.
4. Attach the grapher to the layout instance.
5. Modify the default settings for the layout parameters, if needed.
6. Call the performLayout method inside a `try` block.
7. Read and display information from the layout report.
8. Catch the exceptions.
9. When the layout instance is no longer needed, detach the grapher from the layout instance.

An application that illustrates these steps is provided in *Sample Java Application* on page 38.

Sample Java Application

You can use this application as an example to get started with the layout algorithms of the Graph Layout module. The example uses the Orthogonal Link Layout, but the principles are mostly the same for any of the other layouts.

The source code of the application is named `LayoutSample1.java` and can be found at the location:

`<installdir>/doc/usermansrc/graphlayout/LayoutSample1.java`

To compile and run the example, do the following:

1. Go to the `graphlayout` directory at the above path. (On Windows 98-2000-ME-NT-XP, you must open a DOS Console).

2. Set the `CLASSPATH` variable to the ILOG JViews library and the current directory.

On Windows 98-2000-NT-XP:	`.;<installdir>\classes\jviewsall.jar`
On Unix:	`.:<installdir>/classes/jviewsall.jar`

3. Compile the application:

```
javac LayoutSample1.java
```

4. Run the application:

```
java LayoutSample1
```

The `LayoutSample1.java` contains the following code:

```
// the ILOG JViews Graphic Framework
import ilog.views.*;
// the ILOG JViews Graph Layout Framework
import ilog.views.graphlayout.*;
// the ILOG JViews Link Layout
import ilog.views.graphlayout.link.*;
// the Java AWT package
import java.awt.*;

public class LayoutSample1
{
  public static final void main(String[] arg)
  {
    // Create the layout instance
    IlvLinkLayout layout = new IlvLinkLayout();
    // Create the grapher instance
    IlvGrapher grapher = new IlvGrapher();
    // Create the manager view instance
    IlvManagerView view = new IlvManagerView(grapher);
     // An AWT Frame to display
    Frame frame = new Frame("Layout Sample");
    // The name of the IVL file containing the graph data
    final String fileName = "SampleGraph1.ivl";

    // Put the manager view inside an AWT Frame and show it
    frame.add(view);
    frame.setSize(600, 600);
    frame.setVisible(true);

    // Fill the grapher with nodes and links from a JViews IVL file.
    // Alternatively, the nodes and links could be created by code.
    try {
      grapher.read(fileName);
    } catch (Exception e) {
      System.out.println("could not read " + fileName);
      return;
    }
    // Attach the grapher to the layout instance
    layout.attach(grapher);
    try {
      // Perform the layout and get the layout report
      IlvGraphLayoutReport layoutReport = layout.performLayout();

      int code = layoutReport.getCode();

      // Print information from the layout report (optional)
      System.out.println("layout done in " +
                         layoutReport.getLayoutTime() +
                         " millisec., code = " +
                         code + " (" +
                         layoutReport.codeToString(code) + ")");
    }

    // Catch the exceptions
    catch (IlvGraphLayoutException e) {
      System.out.println(e.getMessage());
    }

    // Detach the grapher from the layout instance
    layout.detach();
  }
  // ... further methods ...
}
```

The sample Java application produces the graph shown in Figure 3.1:

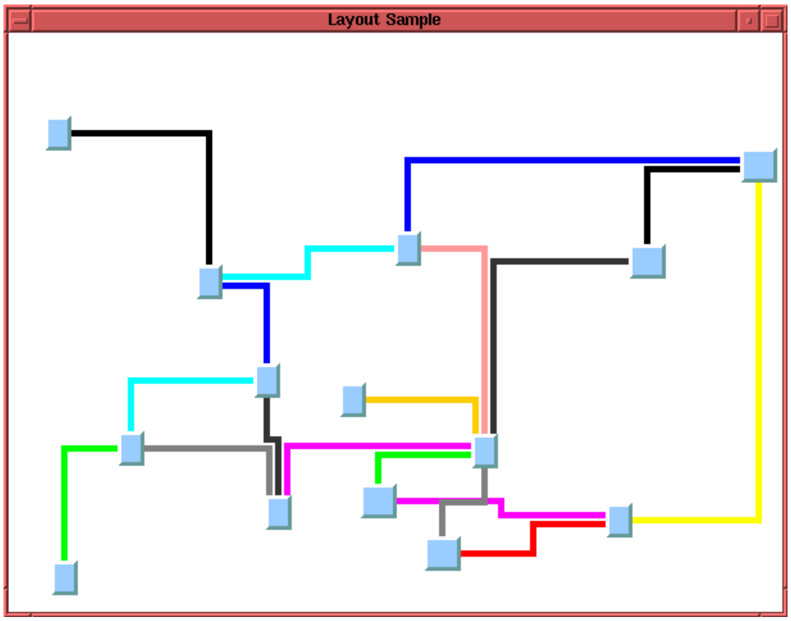


Figure 3.1 *Output from Sample Java Application*

C H A P T E R

4

Layout Algorithms

This chapter describes the layout algorithms of the ILOG JViews Graph Layout module. The following topics are covered:

Determining the Appropriate Layout Algorithm

When using the Graph Layout module, you need to determine which of the ready-to-use layout algorithms is appropriate for your particular needs. Some layout algorithms can handle a wide range of graphs. Others are designed for particular classes of graphs and will give poor results or will reject graphs that do not belong to these classes. For example, a Tree Layout algorithm is designed for tree graphs, but not cyclic graphs. Therefore, it is important to lay out a graph using the appropriate layout algorithm.

Table 4.1 can help you determine which of the layout algorithms is best suited for a particular type of graph.

- Across the top of the table are various classifications of different types of graphs.
- The layout algorithms appear on the left side of the table.
- Table cells containing illustrations indicate when a layout algorithm is applicable for a particular type of graph.

By identifying the general characteristics of the graph you want to lay out, you can see from the table whether a layout algorithm is suited for that particular type of graph.

For example, if you know that the structure of the graph is a tree, you can look at the Domain-Independent Graphs/Trees column to see which layout algorithms are appropriate. The Spring Embedder Layout, Uniform Length Edges Layout, Tree Layout, and Hierarchical Layout could all be used. Use the illustrations in the table cells to help you further narrow your choice.

The Recursive Layout can be used to control the layout of nested graphs (containing subgraphs and intergraph links). This is in particular useful if different layout styles should be applied to different subgraphs. The other layout algorithms such as Spring Embedder Layout, Tree Layout, and Hierarchical Layout treat only flat graphs (unless otherwise noted), that is, a specific layout instance is only able to lay out the nodes and links of the attached graph, but not the nodes and links of its subgraphs. The Recursive Layout allows you to specify which flat layout is used for which subgraph, and it traverses the entire nested graph recursively when applying the layout. As result, the entire nested graph is laid out.

The Multiple Layout can be used to combine several different layouts into one instance. In this case, they become *sublayouts* of the Multiple Layout instance. This is useful in particular for nested graphs when used in combination with the Recursive Layout. The Multiple Layout ensures that the normal layout, the routing of the intergraph links, and the layout of labels are applied in the correct order to a nested graph.

Click on an image to see a larger graph.

Table 4.1 Layout Algorithms and Common Types of Graphs

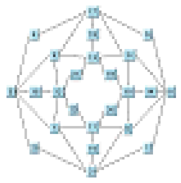
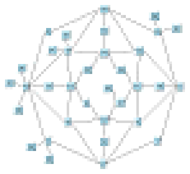
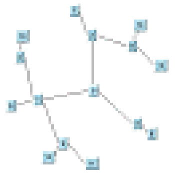
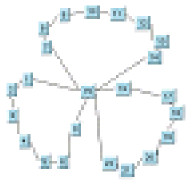
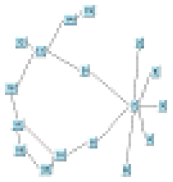
Layout	Domain-Independent Graphs				Telecom-Oriented Representations
	Trees	Cyclic Graphs	Combination of Cycles and Trees	Any Graph	
Topological Mesh Layout			 Requires (semi)manual refinements		
Spring Embedder Layout		 Preferable to avoid heavily interconnected graphs (large number of cycles)			

Table 4.1 Layout Algorithms and Common Types of Graphs (Continued)

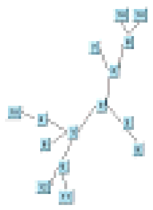
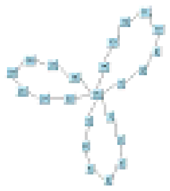
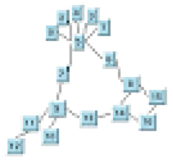
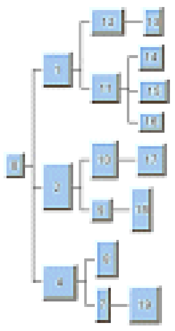
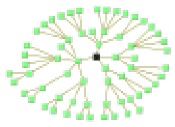
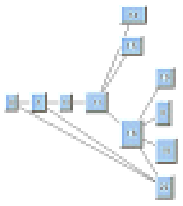
Layout	Domain-Independent Graphs				Telecom-Oriented Representations
	Trees	Cyclic Graphs	Combination of Cycles and Trees	Any Graph	
Uniform Length Edges Layout		 Preferable to avoid heavily interconnected graphs (large number of cycles)			
Tree Layout	 				

Table 4.1 Layout Algorithms and Common Types of Graphs (Continued)

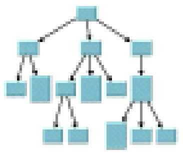
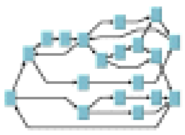
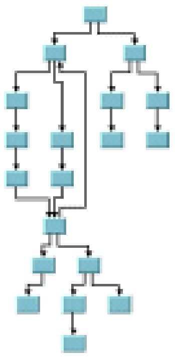
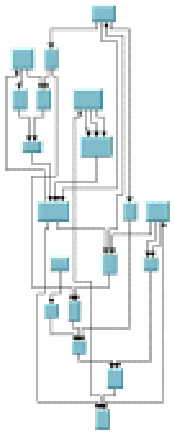
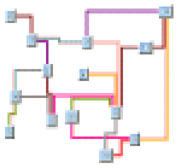
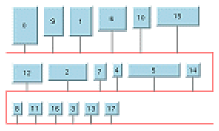
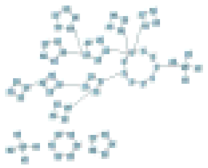
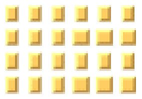
Layout	Domain-Independent Graphs				Telecom-Oriented Representations
	Trees	Cyclic Graphs	Combination of Cycles and Trees	Any Graph	
Hierarchical Layout					
Link Layout					
Bus Layout					 For bus topologies

Table 4.1 Layout Algorithms and Common Types of Graphs (Continued)

Layout	Domain-Independent Graphs				Telecom-Oriented Representations
	Trees	Cyclic Graphs	Combination of Cycles and Trees	Any Graph	
Circular Layout					 For interconnected ring/star topologies
Grid Layout				 Note that the algorithm does not take into account the links between the nodes.	
Recursive Layout				Nested graphs.	
Multiple Layout				Combination of multiple different layout algorithms on the same graph (in particular for nested graphs).	

Typical Ways for Choosing the Layout

Determining an appropriate algorithm for the graph can be done either by the end user at run time or by the programmer of the application. This process can be semiautomatic, where user intervention is involved, or automatic, where the application does everything with no user intervention.

◆ Semiautomatic layout

For applications using a semiautomatic layout, the choice of the layout algorithm is done by the end user. The application can provide a menu or some other way to select the layout algorithm.

In some cases, this may be an iterative process. The user may try different layout algorithms with different values for the parameters and/or may apply manual refinements in order to find the best layout. Eventually, the application can provide some help using textual explanations or by automatically checking the graph to find out to which class it belongs. For example, to detect whether the graph that has been attached to a layout instance is a tree, the `IlvGraphLayoutUtil` class provides the following method:

```
static boolean IsTree(IlvGraphLayout layout, Object startNode)
```

For details on this method, see the corresponding section of the reference manual.

◆ Automatic layout

If an automatic layout is needed, the choice of the layout algorithm can be:

- Dynamically chosen at run time using heuristics or rules to determine the appropriate layout algorithm depending on the structure and/or size of the graph.
- Hard-coded if the developer knows what types of graphs will be used and can determine the appropriate layout algorithm.

Procedure for Dynamically Choosing the Layout Algorithm

If nothing is known about the graphs that the application will need to lay out, the developer can write a routine that automatically chooses the layout algorithm at run-time. The following simple rules could be applied:

1. If the nodes of the graph cannot be moved (they are geo-positioned), use the Link Layout.
2. If the graph is a tree, use the Tree Layout.
3. Otherwise, use one of the layout algorithms that are the less restricted to a given graph category, especially the Uniform Length Edges Layout. (The preferred length of the links could also be computed with respect to the size of the nodes.)

4. If the graph is too large, apply a “divide-and-conquer” strategy. Cut the graph into several subgraphs and apply the layout separately to each subgraph. If the graph is disconnected, you can use the built-in support provided by the layout library to perform this task automatically. (See *Layout of Connected Components* on page 25.)

5. If the graph is nested, use the Recursive Layout algorithm that controls which subgraph is laid out by which (flat) sublayout. Use step 1-4) above to determine the sublayouts for the subgraphs.

Procedure for Hard-coding when Layout is Applied at Programming Time

A special case occurs when the application will deal with only a small set of graphs that are known at the time the application is built. In this case, the layout can be performed at programming time. A possible step-by-step procedure may be the following:

1. Create each graph manually with a graph editor or by code.
2. Try different layout algorithms and choose the best for each graph.
3. Apply manual refinements to the layout if needed.
4. Store the result of the layout by saving the graphers in `.ivl` files.
5. Provide these files with the application.
6. When the application is used, these files will simply be loaded. (There will be no need to perform the layout again since it is already done.)

Procedure for Hard-coding when Layout is Applied at Run Time

If the choice of the layout algorithm is hard-coded, but the layout must be performed at run time because the graphs are not known at programming time, one possible step-by-step procedure for the choice of the appropriate layout algorithm may be the following:

1. Look at sample graphs for your domain.
2. Try to determine some generalities about the properties of the structure and the size of the graph (Is the graph cyclic? Is the graph a tree? Is the graph a combination of the two? What is the number of nodes and links in the graph?)
3. Pick one of the corresponding layout algorithms from Table 4.1 on page 45.
4. Try out the algorithm on one or more samples.

Generic Features and Parameters Support

The generic features and parameters of the Graph Layout module allow you to customize the behavior of the layout algorithms to meet specific needs and to perform useful operations such as saving the layout parameters in a file. Table 4.2 indicates the generic features and parameters that are supported by each layout algorithm. These parameters are defined in the base class for all layout algorithms, `IlvGraphLayout`.

Table 4.2 *Generic Parameters Supported by Layout Algorithms*

Layout Algorithm	Allowed Time	Animation	Fixed Links	Fixed Nodes	Layout of Connected Components	Layout Region	Link Clipping	Link Connection Box	Memory Savings	Percentage Complete	Random Generator Seed Value	Save Parameters to File	Stop Immediately
Topological Mesh Layout (TML)	Yes	Yes		Yes	Yes	Yes	Yes	Yes	Yes			Yes	Yes
Spring Embedder Layout	Yes	Yes		Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Uniform Length Edges Layout	Yes	Yes		Yes	Yes	Yes	Yes	Yes				Yes	Yes
Tree Layout	Yes		Yes	Yes	Yes		Yes	Yes		Yes		Yes	Yes
Hierarchical Layout	Yes		Yes	Yes	Yes		Yes	Yes		Yes		Yes	Yes
Link Layout	Yes	Yes	Yes					Yes				Yes	Yes
Random Layout	Yes			Yes		Yes				Yes	Yes	Yes	Yes
Bus Layout	Yes			Yes	Yes	Yes	Yes	Yes				Yes	Yes
Circular Layout				Yes	Yes	Yes	Yes	Yes				Yes	
Grid Layout	Yes			Yes		Yes						Yes	Yes
Recursive Layout	Yes									Yes		Yes	Yes
Multiple Layout	Yes				Yes					Yes		Yes	Yes

Layout Characteristics

It is often useful to know how certain settings will affect the resulting layout of the graph after the layout algorithm has been applied. Table 4.3 provides additional information about the behavior of the layout algorithms.

Table 4.3 *Layout Characteristics of Layout Algorithms*

Layout Algorithm	**Do the initial positions of the nodes affect the layout?**[1]	**How do I get a different layout of the same graph when I perform the layout a second time?**
Topological Mesh Layout (TML)	No	You can completely change the layout by using the starting node, outer cycle, and fixed nodes parameters. To change only the dimensions of the graph, use the layout region parameter.
Spring Embedder Layout	No	This is the default behavior when using the default parameter settings (the random generator is initialized differently each time). To change only the dimensions of the graph, use the layout region and spring constant parameters.
Uniform Length Edges Layout	Yes	You can completely change the layout by changing the initial positions of the nodes. To change only the dimensions of the graph, use the preferred length of the links or size of the layout region.
Tree Layout	Yes (if incremental mode is switched on)	In incremental mode, you can change the layout by changing the initial positions of the nodes. Furthermore, you can change the layout by selecting a different root node. To change only the dimensions of the graph, use the various offset parameters.
Hierarchical Layout	Yes (if incremental mode is switched on)	In incremental mode, you can change the layout by changing the initial positions of the nodes. Furthermore, you can use specified node level indices to change the level structure. You can use specified node position indices to change the node order within the levels. You can change the layout by changing the link priorities. To change only the dimensions of the graph, use the various offset parameters.

Table 4.3 Layout Characteristics of Layout Algorithms (Continued)

Layout Algorithm	Do the initial positions of the nodes affect the layout?[1]	How do I get a different layout of the same graph when I perform the layout a second time?
Link Layout	Yes	Link Layout routes the links depending on the node positions. It does not move the nodes. You can change the link style option and the dimensional parameters, such as the link offset and final segment length. You can also specify the rules for computing the connection points of the links.
Random Layout	No	This is the default behavior when using the default parameter settings (the random generator is initialized differently each time).
Bus Layout	No, except in incremental mode	You change the dimensions of the graph by using the various dimensional parameters.
Circular Layout	No	You can completely change the layout by using clustering settings and the root clusters parameter. You can change the dimensions of the graph by using the dimensional parameters.
Grid Layout	No, except in incremental mode	You can change various dimensional parameters, layout mode, and so on.
Recursive Layout	Depends on the behavior of the sublayouts applied to the subgraphs.	Depends on the behavior of the sublayouts applied to the subgraphs. You can change the parameters of the sublayouts individually.
Multiple Layout	Depends on the behavior of the sublayout that is applied first.	Depends on the behavior of the sublayouts of the Multiple Layout instance. You can change the parameters of the sublayouts individually.

[1] All of the layout classes provided in ILOG JViews (except the Link Layout) support the fixed nodes mechanism. This means that you can specify nodes that cannot be moved during the layout.