IBM ILOG Diagram for .NET V2.0

Using Graph Layout Algorithms

June 2009

Contents

Using Graph Layout Algorithms

Graph layout algorithms optimize the display of *node*s and links with respect to each other in *graph*s such as network topologies for telecommunication networks and systems management applications.

In This Section

Introducing Graph Layout

Introduces the IBM® ILOG® Diagram for .NET graph layout.

Getting Started with Graph Layout Algorithms

Explains how to configure and execute graph layout algorithms on a graph.

Using the Base Class GraphLayout

Describes the base class GraphLayout and its related generic features and parameters.

Graph Layout Algorithms

Introduces the graph layout algorithms of IBM ILOG Diagram for .NET.

Performing Layout on Nested Graphs

Explains how to define and perform graph layout algorithms on *nested graph*s.

Using Advanced Features

Describes the advanced features of graph layout algorithms in
IBM ILOG Diagram for .NET.

Introducing Graph Layout

This section provides some background information on graphs and layouts, illustrates the appropriate methodology for using algorithms in various applications and contains a brief introduction to the ready-to-use *layout algorithm*s provided with IBM® ILOG® Diagram for .NET.

In This Section

What is IBM ILOG Diagram for .NET Graph Layout

Provides a brief introduction to the IBM ILOG Diagram for .NET graph layout.

Graph Layout: A Brief Introduction

Provides some background information about *graph layout* in general, not specifically related to IBM ILOG Diagram for .NET graph layout.

The Graph Layout Algorithms

Illustrates the ready-to-use layout algorithms.

Common Features

Lists the common features shared by the graph layout algorithms.

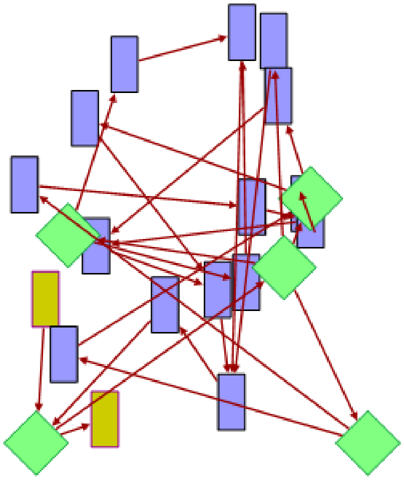
What is IBM ILOG Diagram for .NET Graph Layout

Many types of complex business data can be best visualized as a set of nodes and interconnecting links, more commonly called graph or network. Examples of graphs include business organizational charts, workflow diagrams, telecom network displays, and genealogical *tree*s. 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 IBM® ILOG® Diagram for .NET graph layout algorithms help.

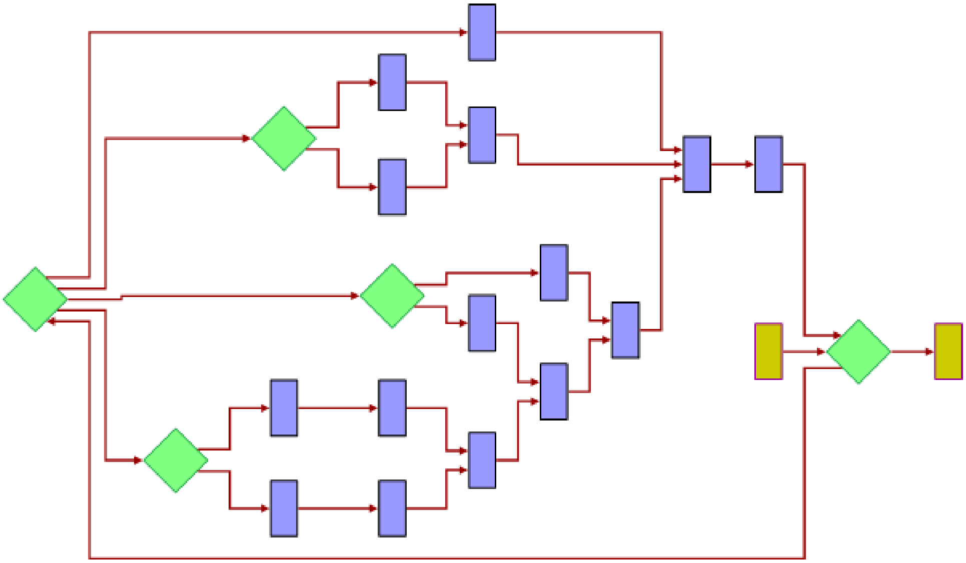
IBM ILOG Diagram for .NET graph layout provides high-level, ready-to-use relationship visualization services. It allows you to take any "messy" graph and apply a sophisticated graph layout algorithm to rearrange the positions of the nodes and links. The result is a more readable and understandable picture.

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

In the following illustration, no formal layout algorithm is used. The nodes were placed randomly when the graph was drawn.



In the following illustration the drawing is obtained by using one of the layout algorithms provided with IBM ILOG Diagram for .NET.



In the second drawing, the layout algorithm has distributed the nodes in levels, avoiding overlapping nodes and allowing to easily follow the flow of information. This drawing presents a much more readable layout than does the first drawing.

Graph Layout: A Brief Introduction

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 tell 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.

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 crossing*s
- Minimizing the total area of the drawing
- Minimizing the number of bends (in *orthogonal drawing*s)
- 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 *cycle*s). 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. In some cases, 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

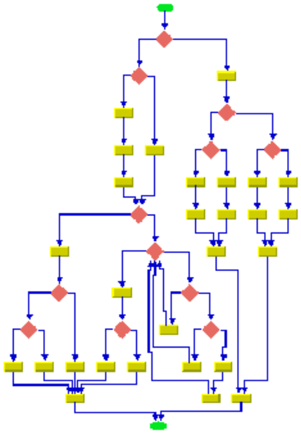
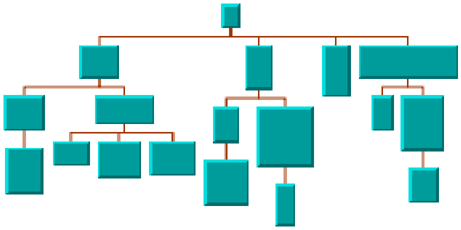
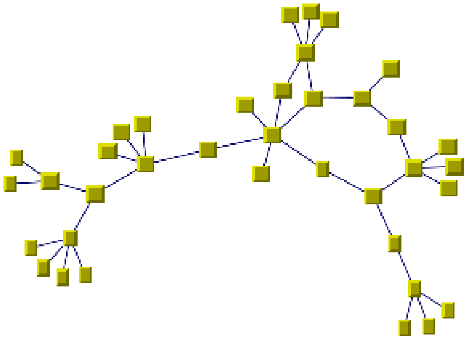
  The layout algorithm is completely redone ("from scratch") each time the graph is changed.

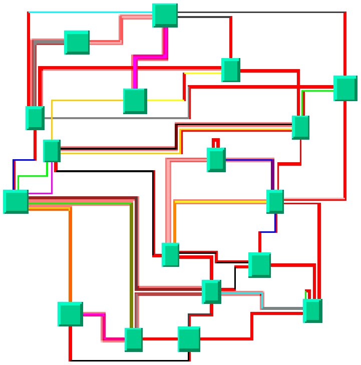
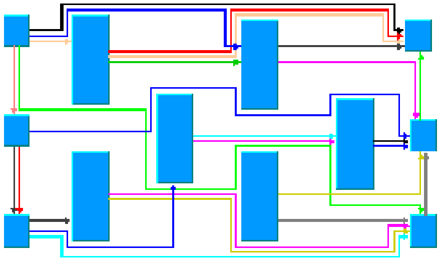
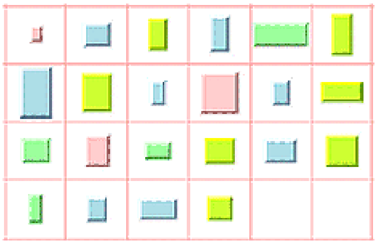
- Incremental layout

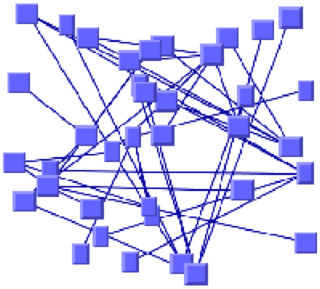
  When the layout algorithm is performed a second time on a modified graph, it tries to preserve the stability of the layout as much as possible. The layout is not performed again from scratch. The layout algorithm also tries to save CPU time by using the previous layout as an initial solution. Some layout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

The Graph Layout Algorithms

The namespace ILOG.Diagrammer.GraphLayout of IBM® ILOG® Diagram for .NET provides numerous ready-to-use layout algorithms. They are shown below with a short description and sample illustrations. In addition, you can develop new layout algorithms using the generic layout framework.

Hierarchical Layout A layout algorithm that arranges nodes in horizontal or vertical levels such that the links flow in a uniform direction.	
Tree Layout 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.	
Force-Directed Layout A layout algorithm that can be used to lay out any type of graph and allows you to specify the length of the links.	

Short Link Layout A layout algorithm for "short" links that reshapes the links of a graph without moving the nodes.	
Long Link Layout A layout algorithm for "long" orthogonal links that reshapes the links of a graph without moving the nodes.	
Grid Layout A layout algorithm that arranges the disconnected nodes of a graph in rows, in columns, or in the cells of a grid.	

Random Layout A layout algorithm that moves the nodes of the graph at randomly computed positions inside an user-defined region.	

Common Features

The graph layout algorithms share the following features:

- **Customizable**: All graph layout algorithms can be tailored through code and interactively in Visual Studio using the Diagram Designer.
- **Adaptable to any graph data structure**: You can program your own graph data structures and apply an IBM® ILOG® Diagram for .NET graph layout algorithm to them.
- **Extensible**: you can easily use the generic framework to implement your own graph layout algorithm, or to combine smaller layout algorithms into a larger one.
- **Suitable for nested graphs**: The graph layout framework provides capabilities to lay out graphs that contain other graphs as nodes. It can even route *intergraph link*s that run between different *subgraph*s of a nested graph.
- **Economic and automatic**: The graph layout framework has capabilities to perform a layout only when needed, that is, when a parameter or a detail of the graph has changed. Furthermore, the framework has the capability to react automatically to such a change.
- **Selective**: You can apply different layout algorithms to different parts of a graph. For instance, you can apply a layout only to the nodes and links meet user-defined conditions.
- **Time controlled**: All layout algorithms can be set to stop automatically when a time has elapsed. Some layout algorithms can even be interrupted during runtime.

Getting Started with Graph Layout Algorithms

This section explains how to configure and perform a graph layout algorithm on a graph contained in an IBM® ILOG® Diagram for .NET graphic container.

For more information on how to create graphs see *Creating Diagrams with Nodes and Links* and *Introducing Link and Anchor Classes*.

The code extracts contained in this section can be found in the QuickStart/GraphLayout sample.

Creating and Configuring a Graph Layout Algorithm

The following example shows how to create an instance of the TreeLayout class and sets some of its parameters:

```
TreeLayout treeLayout = new TreeLayout();
treeLayout.FlowDirection = TreeLayoutFlowDirection.Bottom;
treeLayout.GlobalLinkStyle = TreeLayoutGlobalLinkStyle.OrthogonalStyle;
Dim treeLayout As TreeLayout = New TreeLayout
treeLayout.FlowDirection = TreeLayoutFlowDirection.Bottom
treeLayout.GlobalLinkStyle = TreeLayoutGlobalLinkStyle.OrthogonalStyle
```

Attaching a Graph Layout Algorithm to a Graphic Container

Now you are going to attach the graph layout object to the graphic container to which it will be applied. You can attach two different graph layout algorithms to a graphic container by using the GraphLayout and LinkLayout properties of the GraphicContainer.

The **GraphLayout** property contains a graph layout algorithm that arranges the nodes of the graph like the HiearchicalLayout, TreeLayout, ForceDirectedLayout or GridLayout classes.

Note: *These algorithms often change the shapes of the links.*

The **LinkLayout** property contains a second graph layout algorithm that only changes the shapes of the links without moving the nodes like the ShortLinkLayout or LongLinkLayout algorithms.

In most cases, these two properties ease the configuration of the graph layout algorithms. For example, to build a business process diagram, a user would need to arrange the nodes of the diagram (using the algorithm stored in the **GraphLayout** property) while editing. He would also need to route the links (using the algorithm stored in the **LinkLayout** property) to reduce link crossings, without moving the nodes again, because he might have moved some nodes manually.

The following example shows how to attach a graph layout algorithm to a graphic container (here referred to as group) using the **GraphLayout** property:

```
Group group;
group.GraphLayout = treeLayout;
Dim group As Group
group.GraphLayout = treeLayout
```

Executing a Graph Layout Algorithm

To execute a graph layout algorithm attached to a graphic container, use the PerformGraphLayout method of the **GraphicContainer** class. This method executes the graph layout algorithms defined by the **GraphLayout** and/or **LinkLayout** properties.

The behavior of the **PerformGraphLayout** method is controlled by the following properties of the **GraphicContainer** class:

- GraphLayoutActive: if this property is **true**, the graph layout algorithm contained in the **GraphLayout** property will be called (if it is not **null**).

- LinkLayoutActive: if this property is **true**, the graph layout algorithm contained in the **LinkLayout** property will be called (if it is not **null**). The Link Layout algorithm will always be called after the graph layout algorithm when both are specified and active.
- LayoutRecursively: if this parameter is **true**, the algorithm is executed recursively on all the subcontainers contained in the graphic container. See *Performing Layout on Nested Graphs* for more details.

By default, all the three properties are **true**, so the **PerformGraphLayout** method performs both the graph layout and Link Layout algorithms (if they are specified) on the graphic container and recursively on its subcontainers.

The following example shows how to execute the graph layout algorithms attached to a **Group**:

```
group.PerformGraphLayout();
group.PerformGraphLayout()
```

Using the Base Class GraphLayout

This section introduces you to the API of the base class GraphLayout and describes the generic features and parameters.

In This Section

Using the API of the Base Class GraphLayout

Explains how to use the base class **GraphLayout**.

Layout Parameters and Features in the GraphLayout Base Class

Explains how to use the parameters and features defined by the base class **GraphLayout**.

Using the API of the Base Class GraphLayout

This section covers the following topics:

- *The Base Class: GraphLayout*
- *Instantiating a Subclass of GraphLayout*
- *Attaching/Detaching a Graph*
- *Performing a Layout*

The Base Class: GraphLayout

The GraphLayout 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: HierarchicalLayout, TreeLayout, ForceDirectedLayout, ShortLinkLayout, LongLinkLayout, GridLayout, and RandomLayout. You can also create your own subclasses to implement other layout algorithms. See *Defining Your Own Type of Layout*.

Despite the fact that only subclasses of **GraphLayout** are directly used to obtain the layouts, it is still useful 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.

> ***Warning:*** *In many cases, the basic operations such as specifying the layout to be applied to a Graphic Container and performing it can be done using the simple API directly on the class Graphic Container, as explained in Getting Started with Graph Layout Algorithms.*

Instantiating a Subclass of GraphLayout

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

```
TreeLayout layout = new TreeLayout();
Dim layout As TreeLayout = New TreeLayout
```

Attaching/Detaching a Graph

> ***Warning:*** *In many cases, attaching the layout can be done using the simple API directly on the class Graphic Container, as explained in Getting Started with Graph Layout Algorithms.*

You must attach the graph before performing the layout. The method Attach, defined on the class **GraphLayout**, allows you to specify the graph you want to lay out. For example:

```
...
GraphicContainer graph = new Group();
/* Add nodes and links to the container here */
layout.Attach(container);
...
Dim graph As GraphicContainer = New Group
' Add nodes and links to the container here
layout.Attach(graph)
```

The **Attach** method does nothing if the specified graph is already attached. If a different graph is attached, this method first detaches this old graph, then attaches the new one. You can obtain the attached graph using the method GetGraphicContainer. If the graph is attached in this way, a default graph model is created internally. For details on the graph model, see *Using the Graph Model*. The attached graph model can be obtained by calling the method GetGraphModel.

> ***Warning:*** *You are not allowed to attach a default model created internally to any other layout instance, nor to use it in any way once it has been detached from the layout instance. For details, see Using the Class GraphicContainerAdapter.*

After layout, when you no longer need the layout instance, you should call the method Detach.

If the **Detach** method is not called, some objects may not be garbage-collected. This method also performs clean-up operations on the graph, such as removing properties that may have been added to the graph objects by the layout algorithm. It also removes layout parameters of nodes and links.

> ***Note:*** *A layout instance should stay attached as long as its layout parameters are relevant for the graph. Only when the layout parameters, and therefore the entire layout instance, become irrelevant for this graph should it be detached.*

Performing a Layout

> ***Warning:*** *In many cases, performing the layout can be done using the simple API directly on the class Graphic Container, as explained in Getting Started with Graph Layout Algorithms.*

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

```
GraphLayoutReport PerformLayout()
GraphLayoutReport PerformLayout(bool force, bool traverse)
```

The first version of the method is equivalent to the second called with a false value for both arguments. 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 graph 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 have been added or removed.
- Nodes or links have been moved or reshaped.
- The value of a layout parameter has been modified.
- The transformer of a view (class DiagramView) of the graph 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 traverse determines whether the layout is performed recursively in a nested graph. For details, see *Performing Layout on Nested Graphs*.

The protected abstract method Layout 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 GraphLayoutReport (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 Graph Layout Report.*)

Note that the layout report that is returned can be an instance of a subclass of **GraphLayoutReport** depending on the particular subclass of **GraphLayout** you are using. For example, it will be an instance of LongLinkLayoutReport if you are using the class LongLinkLayout. Subclasses of **GraphLayoutReport** are used to store layout algorithm-dependent information.

Further Information

You can find more information about the class **GraphLayout** in the following sections:

- *Layout Parameters and Features in the GraphLayout Base Class* contains the methods that are related to the customization of the layout algorithms.
- *Using Event Handlers* tells you about the layout event handler mechanism.
- *Defining Your Own Type of Layout* tells you how to implement new subclasses.

For details on **GraphLayout** and other graph layout classes, see the API Reference Documentation.

Layout Parameters and Features in the GraphLayout Base Class

The GraphLayout base class defines a number of generic features and parameters. These features and parameters can be used to customize the layout algorithms.

Although the **GraphLayout** class defines the generic parameters, it does not control how they are used by its subclasses. Each layout algorithm (that is, each subclass of **GraphLayout**) supports a subset of the generic features and determines the way in which it uses the generic parameters. When you create your own layout algorithm by subclassing **GraphLayout**, you decide whether you want to use the features and the way in which you are going to use them.

The **GraphLayout** base class defines the following generic features:

- *Allowed Time*
- *Automatic Layout*
- *Coordinates Mode*
- *Layout of Connected Components*
- *Layout Region*
- *Link Clipping*
- *Link Connection Box*
- *Percentage of Completion Calculation*
- *Preserve Fixed Links*
- *Preserve Fixed Nodes*
- *Random Generator Seed Value*
- *Stop Immediately*

If you are using one of the subclasses provided with IBM® ILOG® Diagram for .NET, check the documentation for that subclass to know whether it supports a given parameter and whether it interprets the parameter in a particular way.

Allowed Time

Several layout algorithms can be designed to stop computation when a user-defined time specification is exceeded. This may be done for different reasons: as a security measure to avoid a long computation time on very large graphs or as an upper limit for algorithms that iteratively improve a current solution and have no other criteria to stop the computation.

To specify that the allowed duration, use the property AllowedTime. The time is in milliseconds. The default value is 32000 (32 seconds).

If you write your own subclass of **GraphLayout**, use the method IsLayoutTimeElapsed to know whether the specified time was exceeded.

To indicate whether a subclass of **GraphLayout** supports this mechanism, use the method SupportsAllowedTime. The default implementation returns **false**. A subclass can override this method to return **true** to indicate that this mechanism is supported.

Automatic Layout

For some layout algorithms, it may be suitable to have the layout automatically performed again after each change of the graph, that is, when a node or link moves, is added, or is removed. *Automatic layout* is most useful for link layouts, in a situation where the shape of the links must remain optimal after each editing action of the end-user. It also works well with other layout algorithms, such as ForceDirectedLayout, that offer an incremental behavior, that is, for which a small change of the graph usually produces only a small change of the layout. Automatic layout is generally not suitable for non-incremental layout algorithms.

To enable automatic layout, use the property AutoLayout.

The following hints are important:

- Automatic layout works well if the GraphicContainer instance is not attached to other layouts. If multiple layouts are used for the same **GraphicContainer** instance, they may mutually affect each other. In this case, it is recommended to have at most one of the multiple layouts in automatic mode.
- The following example shows how to perform multiple changes all at the same time in the **GraphicContainer** instance when automatic layout is switched on. Automatic layout is performed only once at the end of all the changes:

```
layout.Attach(graph);
layout.AutoLayout = true;
   ...
// switch the notification of changes off
graph.setContentsAdjusting(true);
try {
    // ... perform multiple changes without any automatic layout
    ...
} finally {
    // now the graph notifies layout about the changes:
    // therefore, only one automatic layout is performed
    graph.setContentsAdjusting(false);
}
```

For more information about automatic layout, see the method PerformAutoLayout in the API Reference Documentation.

Coordinates Mode

The geometry, that is, the position and size, of the graphic objects that are used to represent nodes and links in a **GraphicContainer** instance is subject to a transformer (see the class Transform). By default, the layout algorithms consider the geometry of the nodes and links of a **GraphicContainer** in a coordinate space that is appropriate for most cases. In some situations, it can be useful to specify a different coordinate space. For details, see *Choosing the Layout Coordinate Space*.

To specify, for instance, the view coordinate space, use the property CoordinatesMode.

The values are:

◆ **CoordinatesMode.GraphicContainerCoordinates**

The geometry of the graph is computed using the coordinate space of the **GraphicContainer** attached to the layout instance, without applying any transformation.

Use this mode:

- if you visualize the graph at zoom level 1, or
- if you do not visualize it at all (no DiagramView), or
- if the graph contains only objects which drawing grows and shrinks proportionally with the zoom level.

In all these cases, there is no need to take the transformer zoom level into account during the layout.

Note that in this mode the dimensional parameters of the layout algorithms are considered specified in the coordinates of the Graphic Container.

◆ **CoordinatesMode.ViewCoordinates**

The geometry of the graph is computed in the coordinate space of the **DiagramView**. More exactly, all the coordinates are transformed using the current reference transformer.

This mode should be used if you want the dimensional parameters of the layout algorithms to be considered as being specified in diagram view coordinates.

◆ **CoordinatesMode.InverseViewCoordinates**

The geometry of the graph is computed using the coordinate space of the diagram view and then applying the inverse transformation. This mode is equivalent to the "graphic container coordinates" mode if the geometry of the graphic objects strictly obeys the transformer, that is, the drawing of all objects grows and shrinks proportionally with the zoom level. (A small difference may exist because of the limited precision of the computations.)

On the contrary, if the drawing of some graphic objects does not grow and shrink proportionally with the zoom level (for example, links with a maximum width), this mode gives different results from the graphic container coordinates mode. These results are optimal if the graph is visualized using the same transformer as the one taken into account during the layout.

Note that in this mode the dimensional parameters of the layout algorithms are considered specified in graphic container coordinates.

The default mode is **CoordinatesMode.InverseViewCoordinates**.

See Also *Specifying the Coordinates Mode*

Layout of Connected Components

The base class **GraphLayout** provides generic support for the layout of a *disconnected graph* (composed of *connected components*).

To enable the placement of disconnected graphs, use the property LayoutOfConnectedComponentsEnabled.

> ***Note:** One of the layout classes, HierarchicalLayout, has a built-in algorithm for placing connected components. This algorithm is enabled by default and fits the most common situations. For this layout class, the generic mechanism provided by the base class **GraphLayout** is disabled by default.*

When enabled, a default instance of the class GridLayout is used internally to place the disconnected graphs. If necessary, you can customize this layout:

```
GridLayout gridLayout = new GridLayout();
gridLayout.LayoutMode = GridLayoutMode.TileToRows;
gridLayout.TopMargin = 20f;

layout.LayoutOfConnectedComponents = gridLayout;
Dim gridLayout As GridLayout = New GridLayout
gridLayout.LayoutMode = GridLayoutMode.TileToRows
gridLayout.TopMargin = 20F
layout.LayoutOfConnectedComponents = gridLayout
```

For Experts

The various capabilities of the class **GridLayout** cover most of the likely needs for the placement of disconnected graphs. However, if necessary, you can write your own subclass of **GraphLayout** to place disconnected graphs and specify it instead of **GridLayout**:

```
MyGridLayout myGridLayout = new MyGridLayout();
// Settings for myGridLayout, if necessary
layout.LayoutOfConnectedComponents = gridLayout;
Dim gridLayout As MyGridLayout = New MyGridLayout
' Settings for myGridLayout, if necessary
layout.LayoutOfConnectedComponents = gridLayout
```

To indicate whether a subclass of **GraphLayout** supports this mechanism, use the method SupportsLayoutOfConnectedComponents.

The default implementation returns **false**. You can write a subclass to override this behavior.

Layout Region

Some layout algorithms can control the size of the graph drawing and can take into account a user-defined *layout region*.

The layout region the layout region is usually the rectangle that the graph must fit (exactly or approximately) after the layout is performed, or the rectangle which influences the position and/or size of the resulting layout.

To specify the layout region, use the property LayoutRegion. The layout region is interpreted according to the LayoutRegionMode property:

- **GraphLayoutRegionMode.RectangleInGraphicContainerCoordinates**

The value of the property **LayoutRegion** is interpreted as a rectangle in the coordinate system of the GraphicContainer to which this layout is attached.

- **GraphLayoutRegionMode.RectangleInViewCoordinates**

The value of the property **LayoutRegion** is interpreted as a rectangle in the coordinate system of the reference view. The reference view is the first view (an object of type IDiagramView) that displays the **GraphicContainer** to which this layout is attached.

- **GraphLayoutRegionMode.ViewBounds**

The value of the property **LayoutRegion** is ignored, and the layout region is the bounds of the reference view.

To access the layout region, use the method GetSpecLayoutRegion. This method returns the rectangle that defines the specified layout region. The dimensions of the rectangle are in the coordinates of the attached graphic container.

The layout algorithms call a different method: GetCalcLayoutRegion. This method first tries to use the layout region specification by calling the method **GetSpecLayoutRegion**. If this method does not return an invalid rectangle (**Rectangle2D.Invalid**), this rectangle is returned. Otherwise, the method tries to estimate an appropriate layout region according to the number and size of the nodes in the attached graph. If no graph is attached, or the attached graph is empty, it returns a default rectangle (0, 0, 1000, 1000).

To indicate whether a subclass of **GraphLayout** supports the layout region mechanism, use the method SupportsLayoutRegion.

The default implementation returns **false**. A subclass can override this method in order to return **true** to indicate that this mechanism is supported.

> ***Note:** The implementation of the protected abstract method Layout is solely responsible for whether the layout region is taken into account when calculating the layout, and in which manner.*

Link Clipping

If the nodes of a graph have a nonrectangular shape such as a triangle, rhombus, or circle, the layout algorithms can place the connection points of the links exactly on the border of the shape, instead of placing them at the border of the bounding box of the nodes.

The clipping of the connection points can be enabled or disabled using the property LinkClipping. The default value is **true**.

Link Connection Box

If a layout algorithm calculates specific connection points, it places the connection points of links by default at the border of the bounding box of the nodes, symmetrically with respect to the middle of each side. Sometimes it may be necessary to place the connection points on a rectangle smaller or larger than the bounding box, eventually in a nonsymmetric way. For instance, this can happen when labels are displayed below or above nodes. This can be achieved by specifying a link connection box provider. This is an interface which allows you to specify, for each node, a node box different from the bounding box that is used to connect the links to the node.

To set a link connection box provider, use the property LinkConnectionBoxProvider.

You implement the link connection box provider by defining a class that implements the interface ILinkConnectionBoxProvider. This interface defines the following method:

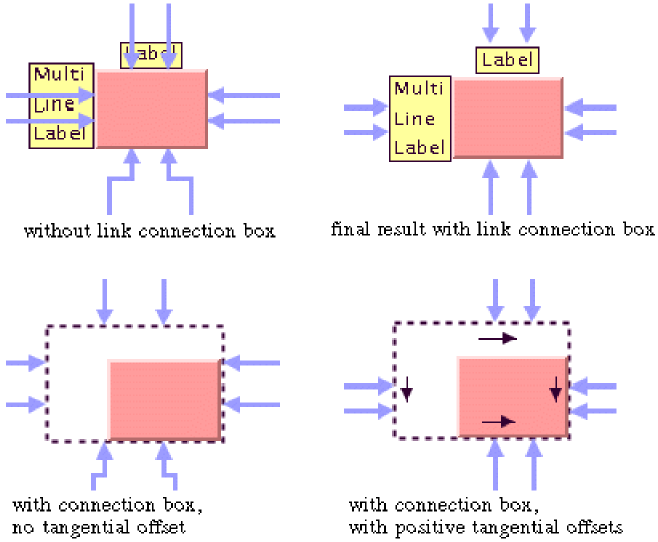
```
Rectangle2D GetBox (
  IGraphModel graphModel,
  Object node
)
Function GetBox ( _
  graphModel As IGraphModel, _
  node As Object _
) As Rectangle2D
```

This method allows you to return the effective rectangle on which the connection points of the links are placed.

A second method defined on the interface allows the connection points to be "shifted" tangentially, in a different way for each side of each node:

```
float GetTangentialOffset (
  IGraphModel graphModel,
  Object node,
  Direction nodeSide
)
Function GetTangentialOffset ( _
  graphModel As IGraphModel, _
  node As Object, _
  nodeSide As Direction _
) As Single
```

The following illustration shows an effect of link connection box provider:



without link connection box

final result with link connection box

with connection box,
no tangential offset

with connection box,
with positive tangential offsets

For instance, to define a link connection box provider that returns a link connection rectangle that is smaller than the bounding box for all nodes that match a given criteria, and shifts up the connection points on the left and right side of all the nodes, implement the interface as follows:

```
public class MyProvider : ILinkConnectionBoxProvider
{
  public Rectangle2D GetBox(IGraphModel graphModel, Object node)
  {
    Rectangle2D rect = graphModel.BoundingBox(node);
    if (node.Name == "Obj1") {
      // need a rect that is 4 units smaller
      rr.Inflate(-4, -4);
    }
    return rect;
  }

  public float GetTangentialOffset(IGraphModel graphModel,
                                   Object node, NodeSide side)
  {
    switch (nodeSide) {
      case NodeSide.Left:
      case NodeSide.Right:
        return -10; // shift up with 10 for both left and right side
      case NodeSide.Top:
      case NodeSide.Bottom:
      default:
        return 0; // no shift for top and bottom side
    }
  }
}
Public Class MyFilter
Implements INodeSideFilter

  Public Function GetBox(ByVal graphModel As IGraphModel,  _
                         ByVal node As Object) _
                                      As Rectangle2D
   Dim rect As Rectangle2D = graphModel.BoundingBox(node)
   ' For some nodes, we need a rect that is 4 units smaller
   If node.Name = "Obj1" Then
     rr.Inflate(-4, -4)
   End If
   Return rect
  End Function

  Public Function GetTangentialOffset(ByVal graphModel As IGraphModel, _
                              ByVal node As Object, _
                              ByVal side As NodeSide) As Single
   Dim rect As Rectangle2D = graphModel.BoundingBox(node)
  Select side
   Case NodeSide.Left, NodeSide.Right
     Return -10
   Case NodeSide.Top, NodeSide.Bottom, Else
     Return 0
   End Select
End Function
End Class
```

Some layout algorithms allow you to use the link connection box interface and the link clip interface in a combined way. It is specific to each layout algorithm how the interfaces will be used and which connection points are the final result.

To indicate whether a subclass of **GraphLayout** supports the link connection box provider, use the method SupportsLinkConnectionBox.

The default implementation returns **false**. You can write a subclass to override this method in order to return **true** to indicate that this mechanism is supported.

Percentage of Completion Calculation

Some layout algorithms can provide an estimation of how much of the layout has been completed. This estimation is made available as a percentage value that is stored in the graph layout report. When the algorithm starts, the percentage value is set to 0. The layout algorithm calls from time to time the property IncreasePercentageComplete to increase the percentage value by steps until it reaches 100.

To see an example of how to read the percentage value during the running of a layout, see *Graph Layout Event Handlers*.

To indicate whether a subclass of **GraphLayout** supports this mechanism, use the method SupportsPercentageComplete.

The default implementation returns **false**. A subclass can override this method to return **true** to indicate that this mechanism is supported.

Preserve Fixed Links

At times, you may want some links of the graph to be "pinned" (that is, to stay in their current shape when the layout is performed). You need a way to indicate the links that the layout algorithm cannot reshape. This makes sense especially when using a semi-automatic layout (the method where the end user fine-tunes the layout by hand after the layout is completed) or when using an incremental layout (the method where the graph and/or the shape of the links is modified after the layout has been performed, and then the layout is performed again).

To specify that a link is fixed:

```
public virtual void SetFixed (
  Object nodeOrLink,
  bool value
)
Public Overridable Sub SetFixed ( _
  nodeOrLink As Object, _
  value As Boolean _
)
```

If the value parameter is set to **true**, it means that the link is fixed. To obtain the current setting for a link:

```
public virtual bool GetFixed (
  Object nodeOrLink
)
Public Overridable Function GetFixed ( _
  nodeOrLink As Object _
) As Boolean
```

The default value is **false**. To remove the fixed attribute from all links in the graph, use the method UnfixAllLinks.

The fixed attributes on links will be taken into consideration only if you also set the property PreserveFixedLinks to **true**.

To indicate whether a subclass of **GraphLayout** supports this mechanism, use the method SupportsPreserveFixedLinks.

The default implementation returns **false**. A subclass can override this method in order to return **true** to indicate that this mechanism is supported.

Preserve Fixed Nodes

At times, you may want some nodes of the graph to be "pinned" (that is, to stay in their current position when the layout is performed). You need a way to indicate the nodes that the layout algorithm cannot move. This makes sense especially when using a semi-automatic layout (the method where the end user fine-tunes the layout by hand after the layout is completed) or when using an incremental layout (the method where the graph and/or the position of the nodes is modified after the layout has been performed, and then the layout is performed again).

To specify that a node is fixed:

```
public virtual void SetFixed (
  Object nodeOrLink,
  bool value
)
Public Overridable Sub SetFixed ( _
  nodeOrLink As Object, _
  value As Boolean _
)
```

If the value parameter is set to **true**, it means that the node is fixed. To obtain the current setting for a node:

```
public virtual bool GetFixed (
  Object nodeOrLink
)
Public Overridable Function GetFixed ( _
  nodeOrLink As Object _
) As Boolean
```

The default value is **false**. To remove the fixed attribute from all nodes in the graph, use the method UnfixAllNodes.

The fixed attributes on nodes will be taken into consideration only if you also set the property PreserveFixedNodes to **true**.

To indicate whether a subclass of **GraphLayout** supports this mechanism, use the method SupportsPreserveFixedNodes.

The default implementation returns **false**. A subclass can override this method in order to return **true** to indicate that this mechanism is supported.

Random Generator Seed Value

Some layout algorithms use random numbers (or randomly chosen parameters) for which they accept a user-defined *seed value*. For example, the Random Layout uses the random generator to compute the coordinates of the nodes.

Subclasses of **GraphLayout** that are designed to support this mechanism allow the user to choose one of three ways of initializing the random generator:

- With a default value that is always the same.
- With a user-defined seed value that can be changed when re-performing the layout.
- With an arbitrary seed value, which is different each time. In this case, the random generator is initialized based on the system time.

The user chooses the initialization option depending on what happens when the layout algorithm is performed again on the same graph. If the same seed value is used, the same layout is produced, which may be the desired result. In other situations, the user may want to produce different layouts in order to select the best one. This can be achieved by performing the layout several times using different seed values.

To specify the seed value, use the property SeedValueForRandomGenerator. The default seed value is 0.

The user-defined seed value is used only if the property UseSeedValueForRandomGenerator is set to **true**.

To indicate whether a subclass of **GraphLayout** supports this parameter, use the method SupportsRandomGenerator.

The default implementation returns **false**. A subclass can override this method in order to return **true** to indicate that this parameter is supported.

Stop Immediately

Several layout algorithms can stop computation when an external event occurs, for instance when the user hits a "Stop" button. To stop the layout, you can call the method StopImmediately.

This method is typically called in a multithreaded application from a separate thread that is not the layout thread. The method returns **true** if the stop was initiated and false if the algorithm cannot stop. The method returns immediately, but the layout thread usually needs some additional time after initiating the stop to clean up data structures.

The consequences of stopping a layout process depend on the specific layout algorithm. Some layout algorithms have an iterative nature. Stopping the iteration process results in a slight loss of quality in the drawing, but the layout can still be considered valid. Other layout algorithms have a sequential nature. Interrupting the sequence of the layout steps may not result in a valid layout. Usually, these algorithms return to the situation before the start of the layout process.

To indicate whether a subclass of **GraphLayout** supports this mechanism, use the method SupportsStopImmediately.

The default implementation returns **false**. You can write a subclass to override this method in order to return **true** to indicate that this mechanism is supported.

Graph Layout Algorithms

This section introduces the graph layout algorithms of IBM® ILOG® Diagram for .NET.

In This Section

Link Layout

Describes the Link Layout algorithm.

Grid Layout

Describes the Grid Layout algorithm.

Random Layout

Describes the Random Layout algorithm.

Determining the Appropriate Layout Algorithm

When using the graph layout package, 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 graph*s. Therefore, it is important to lay out a graph using the appropriate layout algorithm.

Table 1, *Layout Algorithms and Common Types of Graphs* 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 Force-Directed Layout, Tree Layout, and Hierarchical Layout could all be used. Use the illustrations in the table cells to help you further narrow your choice.

You can use the Recursive Layout to control the layout of nested graphs (containing *subgraph*s and intergraph links). This is in particular useful if different layout styles should be applied to different subgraphs. The other layout algorithms such as Force-Directed Layout, Tree Layout, and Hierarchical Layout treat only *flat graph*s (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.

You can use the Multiple Layout 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.

The following table shows Layout Algorithms and Common Types of Graphs.

Table 1 *Layout Algorithms and Common Types of Graphs*

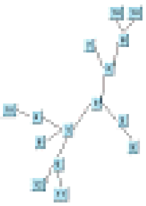
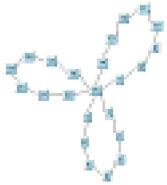
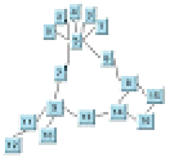
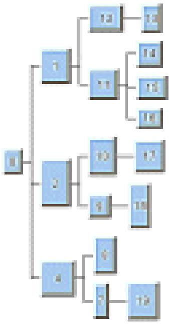
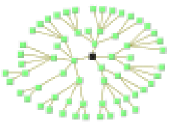
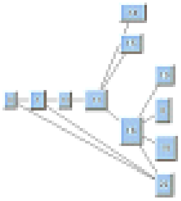
Layout	Domain-Independent Graphs			
	Trees	Cyclic Graphs	Combination of Cycles and Trees	Any Graph
Force-Directed Layout		 Preferable to avoid heavily interconnected graphs (large number of cycles)		N/A
Tree Layout	 	N/A		N/A

Table 1 Layout Algorithms and Common Types of Graphs

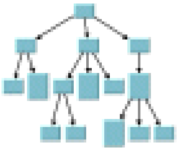
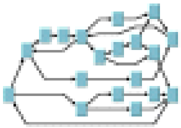
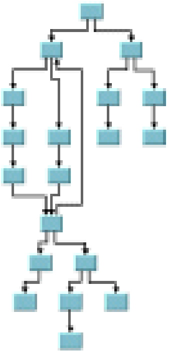
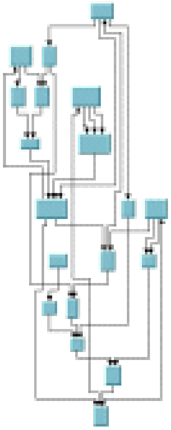
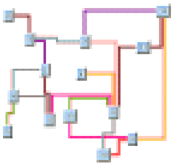
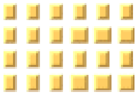
Layout	Domain-Independent Graphs			
	Trees	**Cyclic Graphs**	**Combination of Cycles and Trees**	**Any Graph**
Hierarchical Layout				
Link Layout	N/A	N/A	N/A	
Grid Layout	N/A	N/A	N/A	 Note that the algorithm does not take into account the links between the nodes.

Table 1 Layout Algorithms and Common Types of Graphs

Layout	Domain-Independent Graphs			
	Trees	Cyclic Graphs	Combination of Cycles and Trees	Any Graph
Recursive Layout	N/A	N/A	N/A	Nested graphs.
Multiple Layout	N/A	N/A	N/A	Combination of multiple different layout algorithms on the same graph (in particular for nested graphs).

Typical Ways for Choosing the Layout

This section provides a few methodological hints for designing applications using graph layout. In simple cases, an application will use one fixed graph layout algorithm for each view - if your application falls in this case, you probably don't need to read this section. In more complex applications, the user should be able to choose among different layouts, or the application itself should determine the layout. This section gives some hints on how this can be done and suggests the corresponding procedures in the following subsections.

The choice of the appropriate algorithm for a graph can be done either by the end user at run time or by the programmer when he develops the application. This process can be semiautomatic, when the user is involved, or automatic when the application does everything with no user intervention. This section explains the difference between these two modes and gives you the corresponding procedures in the following subsections:

- *Choosing a Layout Algorithm*
- *Choosing the Layout Algorithm Dynamically*
- *Hard-coding a Layout at Programming Time*
- *Hard-coding a Layout at Run Time*

Choosing a Layout Algorithm

As a programmer of applications, you can choose *Semiautomatic Layout* to involve the end user in the choice of the layout, or *Automatic Layout*, in which case the application does everything with no end user action.

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 to find the best layout. The application may possibly 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 GraphLayoutUtil class provides the method IsTree(GraphLayout layout, Object startNode).

See also *Attaching/Detaching a Graph*.

Automatic Layout

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

- chosen dynamically at run time by means of 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.

Choosing the Layout Algorithm Dynamically

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 Force-Directed 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*.)
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.

Hard-coding a Layout 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 the Diagram 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 graphs in `.IVN` 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.)

Hard-coding a Layout 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 1, *Layout Algorithms and Common Types of Graphs*.
4. Try out the algorithm on one or more samples.

Generic Features and Parameters Support

The graph layout generic features and parameters described in *Layout Parameters and Features in the GraphLayout Base Class* 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.

The following table 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, GraphLayout

Layout Algorithm	**Allowed Time**	**Fixed Links**	**Fixed Nodes**	**Layout of Connected Components**	**Layout Region**	**Link Clipping**	**Link Connection Box**	**Percentage Complete**	**Random Generator Seed Value**	**Stop Immediately**
Force-Directed Layout	Yes	-	Yes	Yes	Yes	Yes	Yes	-	-	Yes
Link Layout	Yes	Yes	Yes	Yes	-	Yes	Yes	Yes	-	Yes
Hierarchical Layout	Yes	Yes	Yes	Yes	-	Yes	Yes	Yes	-	Yes
Link Layout	Yes	Yes	-	-	-	-	Yes	-	-	Yes
Random Layout	Yes	-	Yes	-	Yes	-	-	Yes	Yes	Yes
Grid Layout	Yes	-	Yes	-	Yes	-	-	-	-	Yes
Recursive Layout	Yes	-	-	-	-	-	-	Yes	-	Yes
Multiple Layout	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. The following table provides additional information about the behavior of the layout algorithms.

Layout Algorithm	Do the initial positions of the nodes affect the layout?	How do I get a different layout of the same graph when I perform the layout a second time?
Force-Directed 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. See *Preferred Length*.
Link 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. See *Level Index Parameter*. You can use specified node position indices to change the node order within the levels. See *Position Index Parameter*. You can change the layout by changing the link priorities. See *Link Priority Parameter*. To change only the dimensions of the graph, use the various offset parameters.
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).

Layout Algorithm	Do the initial positions of the nodes affect the layout?	How do I get a different layout of the same graph when I perform the layout a second time?
Grid Layout	Yes (if incremental mode is switched on)	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.

Hierarchical Layout

This section describes the Hierarchical Layout algorithm (class HierarchicalLayout).

In This Section

Samples

Provides some sample drawings produced by the algorithm.

What Types of Graphs?

Explains the type of graph on which you can use the algorithm.

Application Domains

Explains the application domains for the algorithm.

Features

Lists the features of the algorithm.

Limitations

Explains the limitations of the algorithm.

Brief Description of the Algorithm

Provides a short description of the Hierarchical Layout algorithm.

Code Sample

Provides a sample of code that shows how to use the algorithm.

Generic Features and Parameters

Explains what generic features of the graph layout library are supported and how.

Specific Parameters

Presents the parameters of the Hierarchical Layout algorithm.

Incremental Mode

Explains the incremental mode of the Hierarchical Layout algorithm.

Layout Constraints

Explains the constraints system in the Hierarchical Layout algorithm.

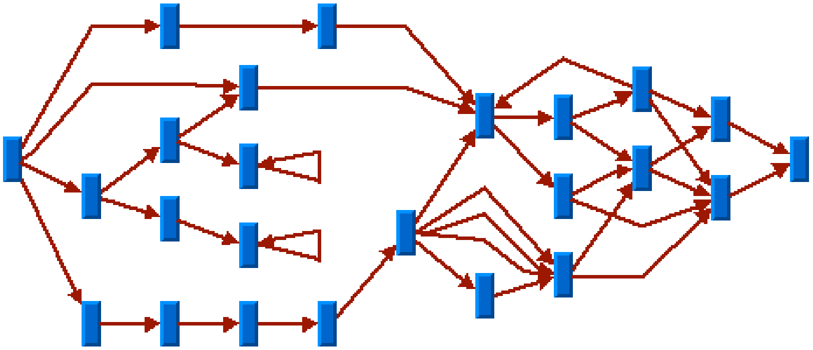
For Experts: More Indices

Presents additional expert information.

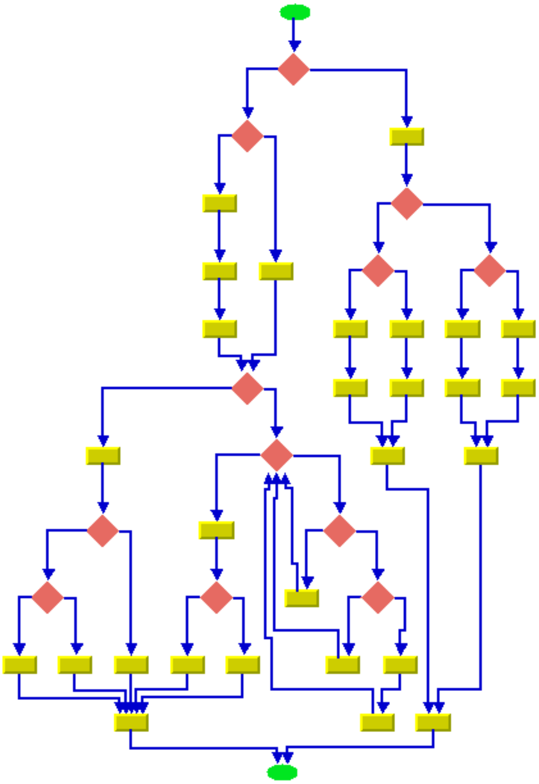
Samples

Here are some sample drawings produced with the Hierarchical Layout.

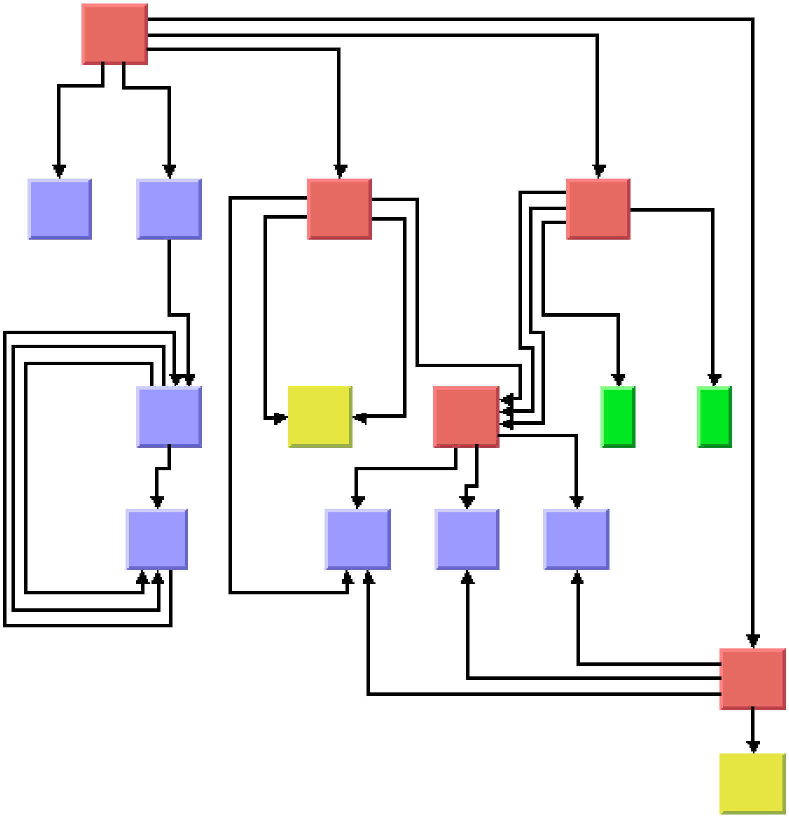
The following illustration shows a sample layout with self-loops, *multiple links*, and *cycle*s.



The following illustration shows a flowchart with orthogonal link style.



The following illustration shows a sample layout with ports and orthogonal link style.



What Types of Graphs?

Any type of graph:

- Preferably graphs with directed links (The algorithm takes the link direction into account.)
- *Connected graphs* and *disconnected graphs*
- *Planar graphs* and *nonplanar graphs*

Application Domains

Application domains of the Hierarchical Layout include:

- Electrical engineering (logic diagrams, circuit block diagrams)
- Industrial engineering (industrial process diagrams, schematic design diagrams)

- Business processing (workflow diagrams, process flow diagrams, PERT charts)
- Software management/software (re-)engineering (UML diagrams, flowcharts, data inspector diagrams, call graphs)
- Database and knowledge engineering (database query graphs)
- CASE tools (designs diagrams)

Features

- Organizes nodes without overlaps in horizontal or vertical levels.
- Arranges the graph such that the majority of links are short and flow uniformly in the same direction (from left to right, from top to bottom, and so on).
- Reduces the number of link crossings. Most of the time, produces drawings with no crossings or only a small number of crossings.
- Often produces balanced drawings that emphasize the symmetries in the graph.
- Supports self-links (that is, links with the same origin and destination node), multiple links between the same pair of nodes, and cycles.
- Efficient, scalable algorithm. Produces a nice layout for most sparse and medium-dense graphs relatively quickly, even if the number of nodes is very large.
- Provides several alignment and offset options.
- Supports port specifications where links attach the nodes. Allows you to specify which side of a node (top, bottom, left, right) a link can be connected to or to specify which relative port position should be used for the connection.
- Supports layout constraints. Allows you to specify relative positional constraints, for instance, that a node is above another node or left of another node.
- Incremental and nonincremental mode. In incremental mode, the previous position of nodes are taken into account. Positions the nodes without changing the relative order of the nodes so that the layout is stable on incremental changes of the graph.
- The computation time depends on the number of nodes, the number of levels, and the number of links that cross several levels. Most of the time, the links are placed between adjacent levels, which keeps the computation time small.

Limitations

- The algorithm tries to minimize the number of link crossings (which is generally an NP-complete problem). It is mathematically impossible to solve this problem quickly for any graph size. Therefore, the algorithm uses a very fast heuristic that obtains a good layout, but not always with the theoretical minimum number of link crossings.

- The algorithm tries to place the nodes such that all links point uniformly in the same direction. It is impossible to place cycles of links in this way. For this reason, it sometimes produces a graph where a small number of links are reversed to point into the opposite direction. The algorithm tries to minimize the number of reversed links (which, again, is an NP-complete problem). Therefore, the algorithm uses a very fast heuristic resulting in a good layout, but not always with the theoretical minimum number of reversed links.

- The computation time required to obtain an appropriate drawing depends most significantly on the number of bends in the links. Since the algorithm places one bend whenever a link crosses a level, the number of bends can grow relatively quickly if the layout requires many long links that span several levels. Therefore, the layout process may become very time-consuming for dense graphs (the number of links is relatively high compared to the number of nodes) or for graphs that require a large number of node levels.

Brief Description of the Algorithm

This algorithm works in four steps.

Step 1: Leveling

The nodes are partitioned into groups. Each group of nodes forms a level. The objective is to group the nodes in such a way that the links always point from a level with smaller index to a level with larger index.

Step 2: Crossing Reduction

The nodes are sorted within each level. The algorithm tries to keep the number of link crossings small when, for each level, the nodes are placed in this order on a line. This ordering results in the relative position index of each node within its level.

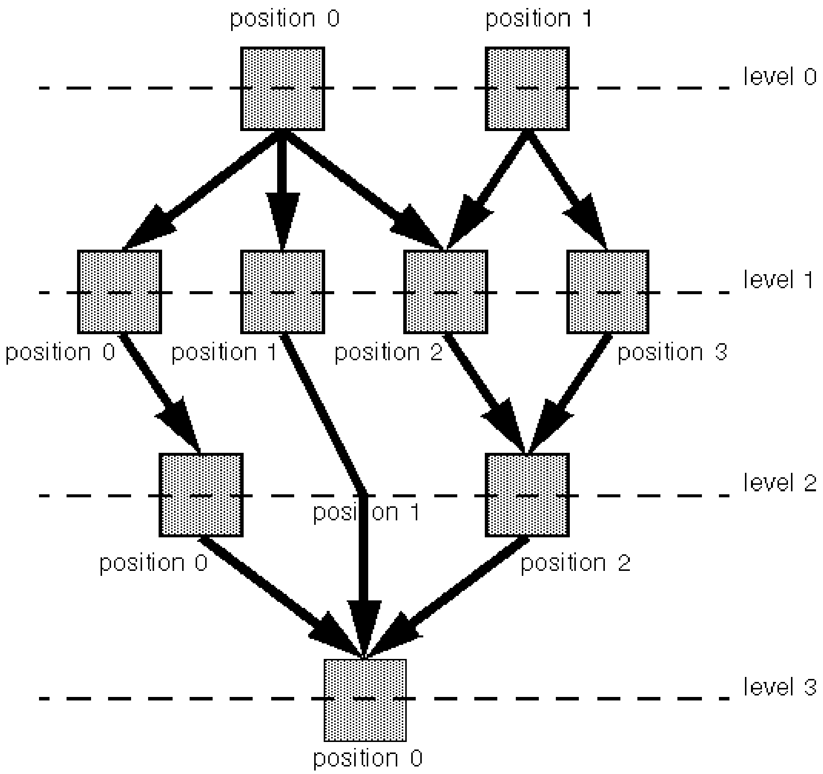
Step 3: Node Positioning

From the level indices and position indices, balanced coordinates for the nodes are calculated. For instance, for a layout where the link flow is from top to bottom, the nodes are placed along horizontal lines such that all nodes belonging to the same level have (approximately) the same y-coordinate. The nodes of a level with a smaller index have a smaller y-coordinate than the nodes of a level with a higher index. Within a level, the nodes with a smaller position index have a smaller x-coordinate than the nodes with a higher position index.

Step 4: Link Routing

The shapes of the links are calculated such that the links bypass the nodes at the level lines. In many cases, this requires that a bend point be created whenever a link needs to cross a level line. In a top-to-bottom layout, these bend points have the same y-coordinate as the level line they cross. (Note that these bend points also obtain a position index).

The following illustration shows how the Hierarchical Layout algorithm uses the level and positions indices to draw the graph.



The steps of the layout algorithm can be affected in several ways. For instance, you can specify the level index that the algorithm should choose for a node in Step 1 or the relative node position within the level in Step 2. You can also specify the justification of the nodes within a level and the style of the links shapes.

Code Sample

The following example shows how to perform a Hierarchical Layout on a graph by using the HierarchicalLayout class:

```
using System;
using ILOG.Diagrammer;
using ILOG.Diagrammer.Graphic;
using ILOG.Diagrammer.GraphLayout;


…
Group group = new Group();
```

```
// Fill the group with nodes here

HierarchicalLayout layout = new HierarchicalLayout ();

group.GraphLayout = layout;
group.PerformGraphLayout();
Imports System
Imports ILOG.Diagrammer
Imports ILOG.Diagrammer.Graphic
Imports ILOG.Diagrammer.GraphLayout

Dim group As Group = New Group

' Fill the group with nodes here

Dim layout As HierarchicalLayout = New HierarchicalLayout

group.GraphLayout = layout

group.PerformGraphLayout()
```

Generic Features and Parameters

The HierarchicalLayout class supports the following generic features defined in the GraphLayout class (see *Layout Parameters and Features in the GraphLayout Base Class*):

- *Allowed Time*
- *Layout of Connected Components*
- *Link Clipping*
- *Link Connection Box*
- *Percentage of Completion Calculation*
- *Preserve Fixed Links*
- *Preserve Fixed Nodes*
- *Stop Immediately*

The following sections describe the particular way in which these parameters are used by this subclass.

Allowed Time

The layout algorithm stops if the allowed time setting has elapsed. For a description of this layout parameter in the GraphLayout class, see *Allowed Time*. If the layout stops early because the allowed time has elapsed, the nodes and links are not moved from their positions before the layout call and the result code in the layout report is **GraphLayoutReportCode.StoppedAndInvalid**.

Layout of Connected Components

The layout algorithm can use the generic mechanism to layout *connected component*s. For more information about this mechanism, see *Layout of Connected Components*. When using this mechanism, each component is laid out in its own individual level structure. Nodes of the first level of one component may be placed at a different position than nodes of the first level of another component.

The generic mechanism to layout connected components is, however, switched off by default. In this case, the layout algorithm can still handle disconnected graphs. It merges all components into a global level structure.

Link Clipping

The layout algorithm can use a link clip provider to clip the connection points of a link. (See *Link Clipping*)

This is useful if the nodes have a nonrectangular shape such as a triangle, rhombus, or circle. If no link clip provider is used, the links are normally connected to the bounding boxes of the nodes, not to the border of the node shapes. See *Using the Link Clipping* for details of the link clipping mechanism.

Link Connection Box

The layout algorithm can use a link connection box provider (see *Link Connection Box*) in combination with the link clip provider. If no link clip provider is used, the link connection box provider has no effect. For details see *Using a Link Connection Box Provider*.

Percentage of Completion Calculation

The layout algorithm calculates the estimated percentage of completion. This value can be obtained from the layout report during the run of the layout. (For a detailed description of this features, see *Percentage of Completion Calculation* and *Graph Layout Event Handlers*.)

Preserve Fixed Links

The layout algorithm does not reshape the links that are specified as fixed. In fact, *fixed links* are completely ignored. For more information on link parameters in the GraphLayout class, see *Preserve Fixed Links* and *Link Style*.

Preserve Fixed Nodes

The layout algorithm does not move the nodes that are specified as fixed. (For more information on node parameters in the GraphLayout class, see *Preserve Fixed Nodes*. Moreover, the layout algorithm ignores *fixed node*s completely and also does not route the links that are incident to the fixed nodes. This can result in unwanted overlapping nodes and link crossings. However, this feature is useful for individual, disconnected components that can be laid out independently.

Stop Immediately

The layout algorithm stops after cleanup if the method StopImmediately is called. If the layout stops early because the allowed time has elapsed, the nodes and links are not moved from their positions before the layout call and the result code in the layout report is **GraphLayoutReportCode.StoppedAndInvalid**.

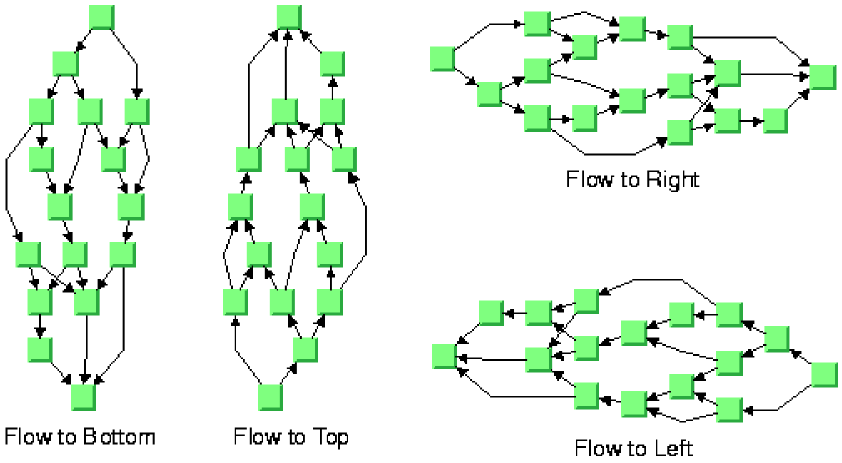
Specific Parameters

The following parameters are specific to the HierarchicalLayout class.

Flow Direction

The flow direction parameter specifies the direction in which the majority of the links should point. If the flow direction is to the top or to the bottom, the node levels are oriented horizontally and the links mostly vertically. If the flow direction is to the left or to the right, the node levels are oriented vertically and the links mostly horizontally.

If the flow direction is to the bottom, the nodes of the level with index 0 are placed at the top border of the drawing. The nodes with level index 0 are usually the root nodes of the drawing (that is, the nodes without incoming links). If the flow direction is to the top, the nodes with level index 0 are placed at the bottom border of the drawing. If the flow direction is to the right, the nodes are placed at the left border of the drawing.



Flow to Bottom

Flow to Top

Flow to Right

Flow to Left

To specify the flow direction towards the bottom use the FlowDirection property.

The valid values for the flow direction are defined by the LayoutFlowDirection enumeration:

- **LayoutFlowDirection.Right** (the default)

- **LayoutFlowDirection.Left**
- **LayoutFlowDirection.Bottom**
- **LayoutFlowDirection.Top**

Level Justification

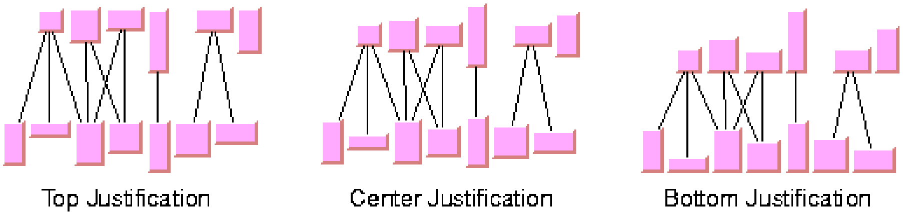
If the layout uses horizontal levels, the nodes of the same level are placed approximately at the same y-coordinate. The nodes can be justified, depending on whether the top border, or the bottom border, or the center of all nodes of the same level should have the same y-coordinate.

If the layout uses vertical levels, the nodes of the same level are placed approximately at the same x-coordinate. In this case, the nodes can be justified to be aligned at the left border, at the right border, or at the center of the nodes that belong to the same level.

To specify the level justification towards the top use the LevelJustification property.

If the flow direction is to the top or to the bottom, the valid values for the level justification are defined by the HierarchicalLayoutLevelJustification enumeration:

- **HierarchicalLayoutLevelJustification.Top**
- **HierarchicalLayoutLevelJustification.Bottom**
- **HierarchicalLayoutLevelJustification.Center** (the default)



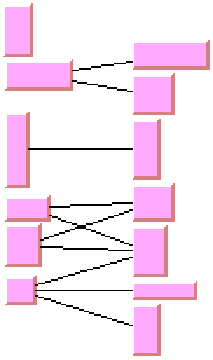
Top Justification

Center Justification

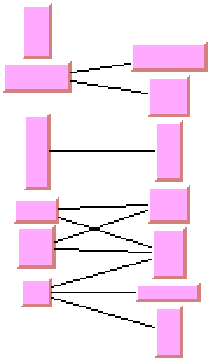
Bottom Justification

If the flow direction is to the left or to the right, the valid values for the level justification are:

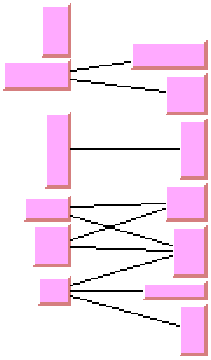
- **HierarchicalLayoutLevelJustification.Left**
- **HierarchicalLayoutLevelJustification.Right**
- **HierarchicalLayoutLevelJustification.Center** (the default)



Left Justification



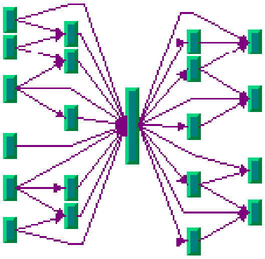
Center Justification



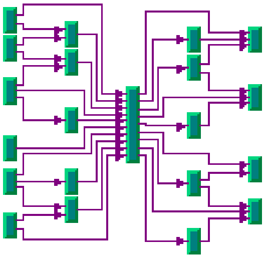
Right Justification

Link Style

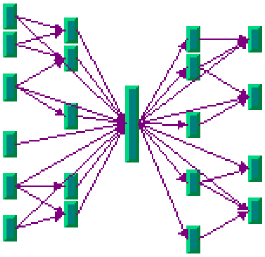
The layout algorithm positions the nodes and routes the links. To avoid overlapping nodes and links, it creates bend points for the shapes of links. The link style parameter controls the position and number of bend points. The link style can be set globally, in which case all links have the same kind of shape, or locally on each link such that different link shapes occur in the same drawing.



Polyline Links



Orthogonal Links



Straight-line Links

Global Link Style

To set the link style use the LinkStyle property.

The valid values for the link style are defined by the HierarchicalLayoutLinkStyle enumeration:

- **HierarchicalLayoutLinkStyle.Polyline**

  All links get a polyline shape. A polyline shape consists of a sequence of line segments that are connected at bend points. The line segments can be turned into any direction. This is the default value.

- **HierarchicalLayoutLinkStyle.Orthogonal**

All links get an orthogonal shape. An orthogonal shape consists of orthogonal line segments that are connected at bend points. An orthogonal shape is a polyline shape where the segments can be turned only in directions of 0, 90, 180 or 270 degrees.

- **HierarchicalLayoutLinkStyle.StraightLine**

  All links get a *straight-line* shape. All intermediate bend points (if any) are removed. This often causes overlapping nodes and links.

- **HierarchicalLayoutLinkStyle.NoReshape**

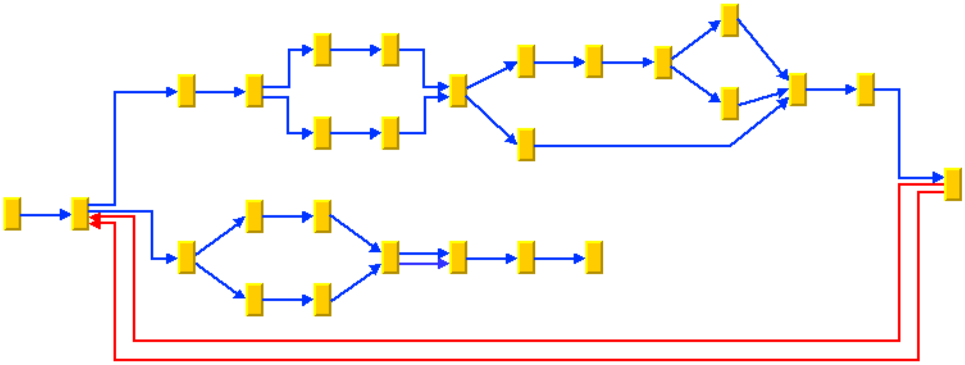
  None of the links is reshaped in any manner. Note, however, that unlike fixed links, the links are not ignored completely. They are still used to calculate the leveling.

- **HierarchicalLayoutLinkStyle.Mixed**

  Each link can have a different link style. The style of each individual link can be set such that different link shapes can occur in the same graph.

Individual Link Style

All links have the same style of shape unless the link style is **HierarchicalLayoutLinkStyle.Mixed**. Only when the global link style is **HierarchicalLayoutLinkStyle.Mixed** can each link have an individual link style.



To specify the style of an individual link, use the methods SetLinkStyle (object link, HierarchicalLayoutLinkStyle style) and GetLinkStyle (object link).

In this case, the link argument must be a graphic link (subclass of Link).

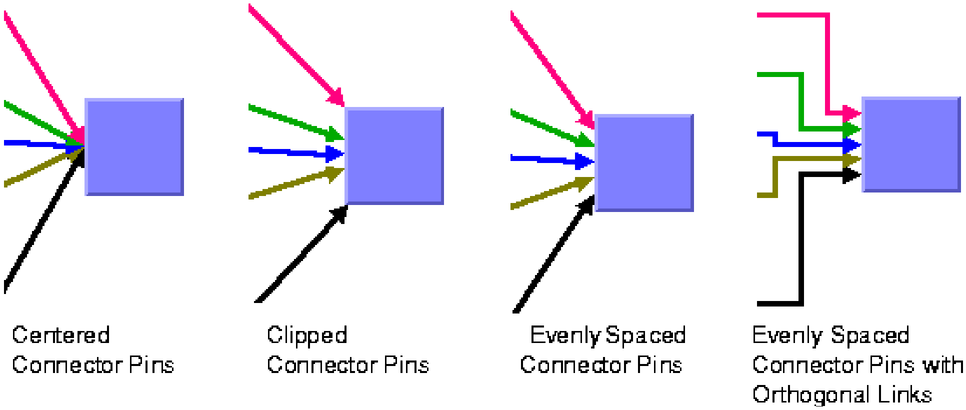
The valid values for the link style of local links are the same as for the global link style:

- **HierarchicalLayoutLinkStyle.Polyline**
- **HierarchicalLayoutLinkStyle.Orthogonal**
- **HierarchicalLayoutLinkStyle.StraightLine**

- **HierarchicalLayoutLinkStyle.NoReshape**

Connector Style

The layout algorithm positions the connection points of links (the anchors) at the nodes automatically. The connector style parameter specifies how these connection points are calculated.



To specify the connector style, use the ConnectorStyle property.

The valid values for style are defined by the HierarchicalLayoutConnectorStyle enumeration:

- **HierarchicalLayoutConnectorStyle.Centered**

  The connection points of the links are placed in the center of the border where the links are attached. This option is well-suited for polyline links and *straight-line* links. It is less well-suited for orthogonal links, because orthogonal links can look ambiguous in this style.

- **HierarchicalLayoutConnectorStyle.Clipped**

  Each link pointing to the center of the node is clipped at the node border. The connector pins are placed at the points on the border where the links are clipped. This option is particularly well-suited for polyline links without port specifications. It should not be used if a port side for any link is specified.

- **HierarchicalLayoutConnectorStyle.EvenlySpaced**

  The connector pins are evenly distributed along the node border. This style guarantees that the connection points of the links do not overlap. This is the best style for orthogonal links and works well for other link styles.

- **HierarchicalLayoutConnectorStyle.Automatic**

The connector style is selected automatically depending on the link style. If any of the links has an orthogonal style or if any of the links has a port side specification, the algorithm chooses evenly spaced connectors. If all the links are straight, it chooses centered connectors. Otherwise, it chooses clipped connectors.

Connection Point Mode

Normally, the layout algorithm is free to choose the termination points of each link. However, the user can specify that the current termination point of a link should be used.

The layout algorithm provides two connection point modes. You can set the connection point mode globally, in which case all connection points have the same mode, or locally on each link, in which case different connection point modes occur in the same drawing.

Global Connection Point Mode

To set the connection point mode for all links, use the OriginPointMode and DestinationPointMode properties.

The valid values for mode are defined by the ConnectionPointMode enumeration:

- **ConnectionPointMode.Free** (the default)

  The layout is free to choose the appropriate position of the connection point on the origin/destination node.

- **ConnectionPointMode.Fixed**

  The layout must keep the current position of the connection point on the origin/ destination node.

- **ConnectionPointMode.Mixed**

  Each link can have a different connection point mode.

Individual Connection Point Mode

All links have the same connection point mode unless the global connection point mode is **ConnectionPointMode.Mixed**. Only when the global connection point mode is set to **ConnectionPointMode.Mixed** can each link have an individual connection point mode.

To set the connection point mode of an individual link, use the methods SetOriginPointMode, GetOriginPointMode, SetDestinationPointMode and GetDestinationPointMode.

The valid values for mode are:

- **ConnectionPointMode.Free** (the default)
- **ConnectionPointMode.Fixed**

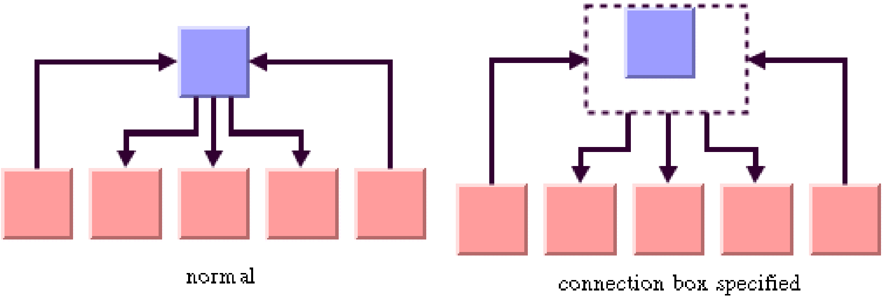
Using a Link Connection Box Provider

By default, the connector style determines how the connection points of the links are distributed on the border of the bounding box of the nodes, symmetrically with respect to the middle of each side. Sometimes it may be necessary to place the connection points on a rectangle smaller or larger than the bounding box. For instance, this can happen when labels are displayed below or above nodes.

You can modify the position of the connection points of the links by providing a class that implements the ILinkConnectionBoxProvider interface. An example for the implementation of a link connection box provider is in *Link Connection Box*. To set a link connection box provider use the LinkConnectionBoxProvider property.

The link connection box provider specifies for each node a link connection box and a tangential shift offsets. The Hierarchical Layout uses the link connection box but does not use the tangential offsets.

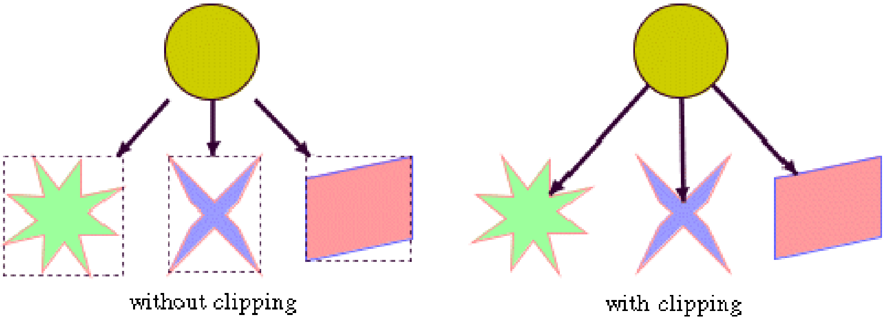
The following illustration shows the effects of customizing the connection box. On the left is the result without any connection box provider. The picture on the right shows the effect if the connection box provider returns the dashed rectangle for the blue node.



Using the Link Clipping

By default, the Hierarchical Layout places the connection points of links at the border of the bounding box of the nodes. If the node has a nonrectangular shape such as a triangle, rhombus, or circle, you may want the connection points to be placed exactly on the border of the shape. This can be achieved by using the link clipping feature. The link clipping corrects the calculated connection point so that it lies on the border of the shape. An example is shown the following illustration:

An example is shown the following illustration:

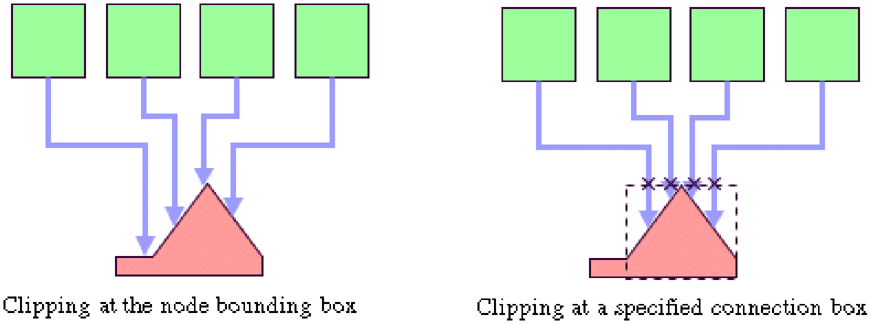


To enable the link clipping, use the property LinkClipping. The default value is **true**.

> ***Note:*** *Additionally to the link clip provider, the ShapeAnchor can be used. This special link anchor updates the clipped connection points automatically during interactive node movements.*

The connector style, the link connection box provider, and the link clip provider work together in the following way: by respecting the connector style, the proposed connection points are calculated on the rectangle obtained from the link connection box provider (or on the bounding box of the node, if no link connection box provider was specified). Then, the proposed connection point is passed to the link clip provider and the returned connection points are used to connect the link to the node.

An example of the combined effect is shown in the following illustration:

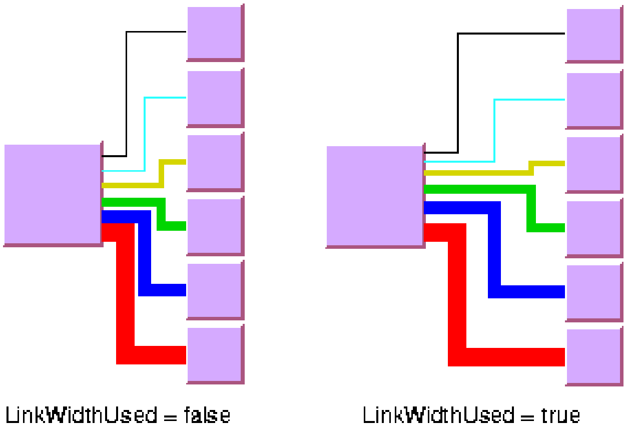

Clipping at the node bounding box
Clipping at a specified connection box

If the links are clipped at the red node (left picture), they appear unsymmetrical with respect to the node shape, because the relevant part of the node (here: the triangle) is not in the center of the bounding box of the node, but the proposed connection points are calculated with respect to the bounding box. This can be corrected by using a link connection box provider to explicitly specify a smaller connection box for the relevant part of the node (right picture) such that the proposed connection points are placed symmetrically at the triangle of the node.

For Experts: Thick Links

If the connector style is **HierarchicalLayoutConnectorStyle.EvenlySpaced**, the links can be evenly spaced with respect to the link center or with respect to the link border. The difference is only visible when links that connect to the same node have different widths. For instance, when the link width indicates the cost or capacity of a flow in the application, many different link widths may occur.

The following illustration shows the effect of using different link widths. In the drawing on the left, the center of the links are evenly distributed at the left node. Each link has the same space available at the node side. Therefore, the thick links appear closer to each other than do the thinner links and the offsets between the link borders are different. In the drawing on the right, the thick links have more space available than do the thinner links. The offset between the link border (at the segments that connect to the left node) is constant because the link width is considered in the calculation of the connection points.



To enable the connector calculation to respect the link width set the LinkWidthUsed property to **true**.

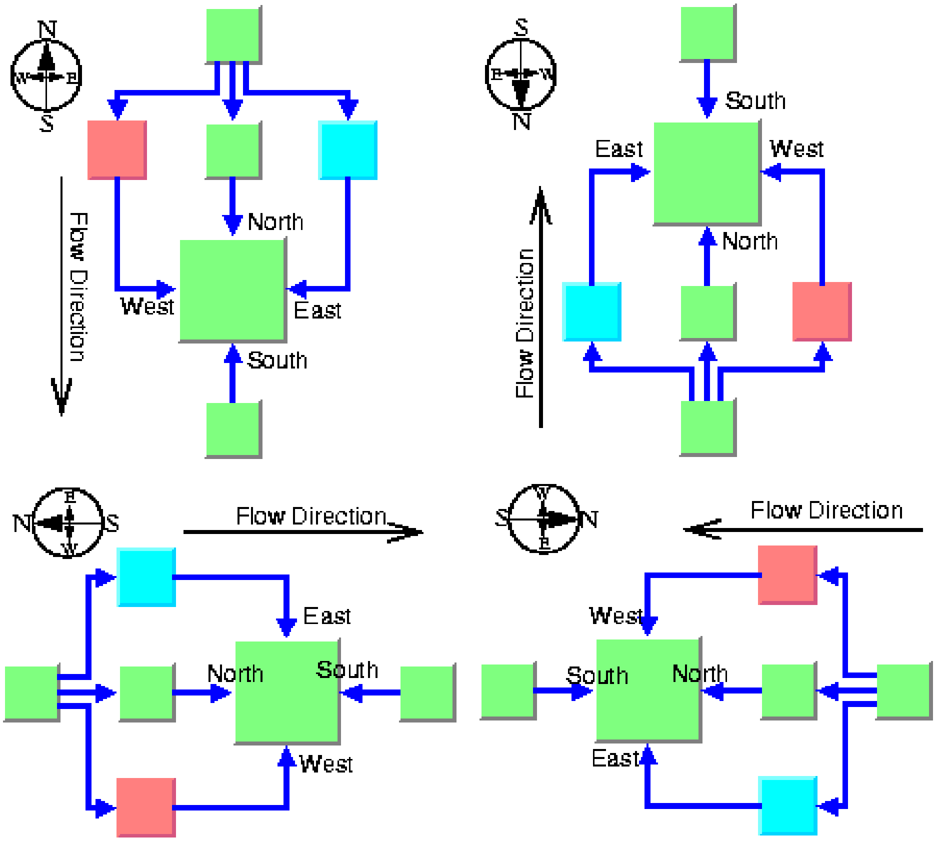
The link width setting is disabled by default. The link width has no effect if the connector styles **HierarchicalLayoutConnectorStyle.Centered** or **HierarchicalLayoutConnectorStyle.Clipped** are used.

Port Sides Parameter

The Hierarchical Layout algorithm produces a layout where the majority of the links flow in the same direction. If the flow direction is towards the bottom, usually the incoming links are connected to the top side of the node and the outgoing links are connected to the bottom side of the node. It is also possible to specify on which side a link connects to the node.

To simplify the explanations of the port sides, we use the compass directions north, south, east, and west. The specified link flow direction is always towards south and the first level is towards north. If the flow direction is towards bottom, north is at the top, south at the bottom, east on the right, and west on the left side of the drawing. If the flow direction is towards right, north is on the left, south on the right, east at the top, and west at the bottom.

The following illustration shows a drawing where the links connect to the larger middle node at the specified port sides. A compass icon shows the compass directions in these drawings.



To specify at which side the link connects to its source node, you can use the method SetFromPortSide(Object link, HierarchicalLayoutSide side). In a similar way, to specify at which side the link connects to its destination node, you can use the method SetToPortSide(Object link, HierarchicalLayoutSide side).

The valid values for side are defined by the HierarchicalLayoutSide enumeration:

- **HierarchicalLayoutSide.Unspecified** (the default)
- **HierarchicalLayoutSide.North**
- **HierarchicalLayoutSide.South**
- **HierarchicalLayoutSide.East**
- **HierarchicalLayoutSide.West**

To retrieve the current choice for a link, use the methods GetFromPortSide(Object link) and GetToPortSide(Object link).

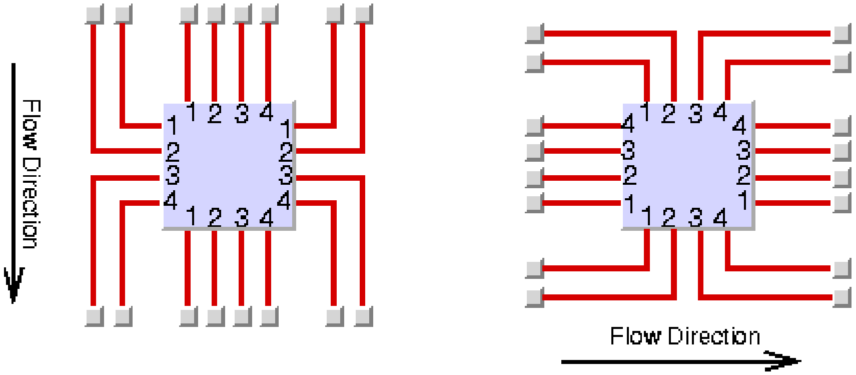
The port sides east and west work particularly well with the orthogonal link style. Polyline links with these port sides sometimes have unnecessary bends. Furthermore, if port sides are specified, the connector style **HierarchicalLayoutConnectorStyle.Clipped** should not be used.

Port Index Parameter

Instead of asking the layout algorithm to decide at which point a link connects to the node border, you can specify where the links connect to the node. You cannot specify the exact location, but you can specify the relative location compared to the connection points of the other links. This is done by using a port index. The illustration below shows a sample layout with ports at many nodes.

Links that have the same port index connect at the same point of the node. The ports are evenly distributed at the node sides, in a similar way as with the connector style **HierarchicalLayoutConnectorStyle.EvenlySpaced**. The ports are ordered according to their indices. On the north and south side of a node, the port indices increase toward the east. On the east and west sides of a node, the port indices increase toward the south. By using port indices in this way, it is easier to rotate a graph by simply changing the flow direction without needing to update all the port specifications.

The following illustration shows how the port indices depend on the flow direction.



Port numbers are normally used in combination with port sides. Therefore, you must specify how many ports are available on each side of a node. To specify the number of ports, use the method SetNumberOfPorts(Object node, HierarchicalLayoutSide side, int numberOfPorts).

For example, to use 4 ports on each side of a specific node, use the calls:

```
layout.SetNumberOfPorts(node, HierarchicalLayoutSide.East, 4);
layout.SetNumberOfPorts(node, HierarchicalLayoutSide.West, 4);
layout.SetNumberOfPorts(node, HierarchicalLayoutSide.North, 4);
layout.SetNumberOfPorts(node, HierarchicalLayoutSide.South, 4);
```

The node side is specified again by **HierarchicalLayoutSide** values. To retrieve the number of ports available at the node, use the method GetNumberOfPorts(Object node, HierarchicalLayoutSide side).

After the number of ports per side is specified, you can choose which port each link connects to. To choose the port side and the port index for a link:

To specify the connection at the source node, use the methods SetFromPortSide(Object link, HierarchicalLayoutSide portSide) and SetFromPortIndex(Object link, int portIndex).

To specify the connection at the destination node, use the methods SetToPortSide(Object link, HierarchicalLayoutSide portSide) and SetToPortIndex(Object link, int portIndex).

To obtain the current port index of a link, use the methods GetFromPortIndex(Object link) and GetToPortIndex(Object link).

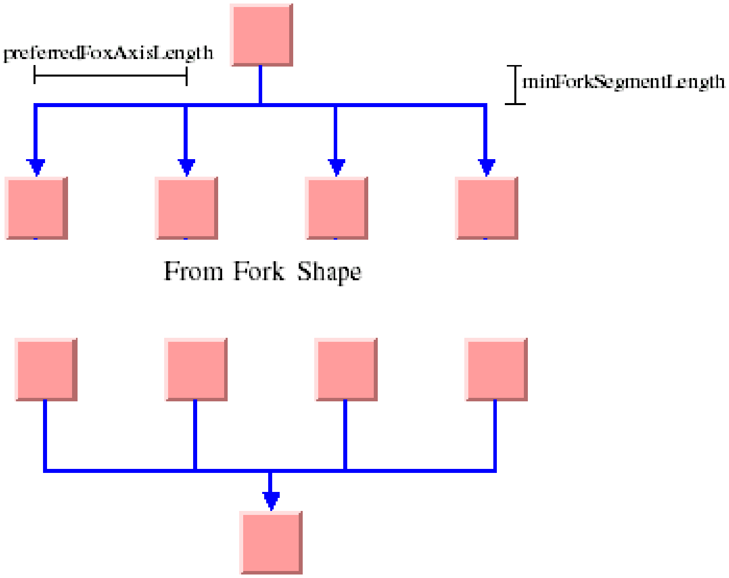
Using the port side and port index specifications are additional constraints for the layout algorithm. The more constraints are specified, the more difficult it is to calculate a layout. Therefore, if too many links have a specified port index, this resulting layout may have more link crossings and be less balanced.

Fork Link Shapes

If several links start at the same position and are orthogonally routed, it is sometimes preferred that the links share the first two link segments. The shape of a link bundle of this kind looks like a fork. To enable the fork shape mode for outgoing links, set the FromFork property to **true**.

To enable the fork shape mode for incoming links set the ToFork property to **true**.

These statements have an effect only if the links are routed orthogonally. The fork appears only at those links that start or end exactly at the same point. Setting the **FromFork** property to **true** by itself does not force the links to start at the same point. To force links to start or end at the same point, use the center connector style (see *Connector Style*) or specify the same port for the links (see *Port Index Parameter*).



There are two spacing properties for the fork shape:

- MinForkSegmentLength

  Specifies the minimal length of the segment that is directly adjacent to the node.

- PreferredForkAxisLength

  Specifies the preferred length of the fork axis per branch (the second segment adjacent to the node). If the fork has five branches, the entire axis has the preferred length five times the specified parameter. The preferred fork axis length is only a hint for the layout algorithm. If enough space is available, the algorithm will enlarge the fork axis to avoid unnecessary link bends. If there is not enough space, the algorithm may as well calculate a fork axis that is smaller than the preferred one.

Fork link shapes may sometimes look ambiguous, in particular when a link starts at the same point where another link ends, because in this case it is impossible to recognize whether the arrowhead belongs to one or the other link.

Link Priority Parameter

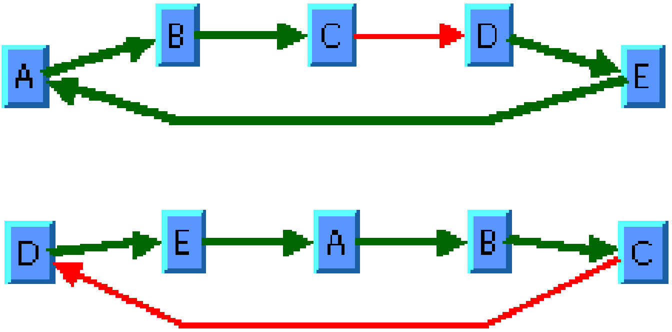
The layout algorithm tries to place the nodes such that all links are short, point in the flow direction, and do not cross each other. However, this is not always possible. Often, links cannot have the same length. If the graph has cycles, some links must be reversed against the flow direction. If the graph is a nonplanar graph, some links have to cross each other.

The link priority parameter controls which links should be selected if long, reversed, or crossing links are necessary. Links with a low priority are more likely to be selected than links with a high priority. This does not mean that low-priority links are always longer, reversed, or crossed, because the graph may have a structure such that no long, reversed or crossing links are necessary.

To set the link priority, use the methods SetLinkPriority(Object link, float priority) and GetLinkPriority(Object link).

The default value of the link priority is 1.0. Negative link priorities are not allowed.

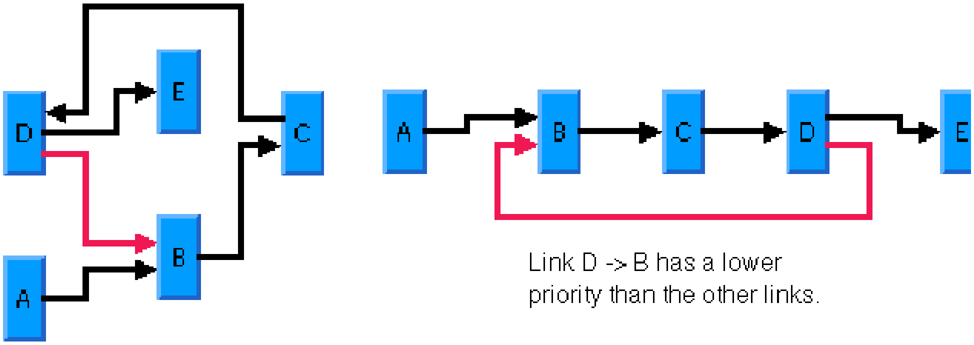
For an example of using the link priority, consider a cycle A->B->C->D->E->A. It is impossible to lay out this graph without reversing any link. Therefore, the layout algorithm selects one link to be reversed. To control which link is selected, you can give one link a lower priority than the others. This link will be reversed. In the following illustration, the bottom layout shows the use of the link priority. The link C->D was given the priority 0.5, while all the other links have the priority 1.0. Therefore C-D is reversed. The top layout in this illustration shows what happens when all links have the same priority. Link E->A is reversed.



All links have the same priority.

Link C -> D has a lower priority than the other links.

The use of link priorities is important in combination with ports. Links with "from" ports on the south side and "to" ports on the north side are preferably laid out opposite to the flow direction. Such a feedback link may cause parts of the drawing to tip over. The next illustration shows an example. The red link is a feedback link with port specifications. To obtain the correct result as shown in the drawing on the right side of this illustration, you would set the priority of the feedback link to a very low value.

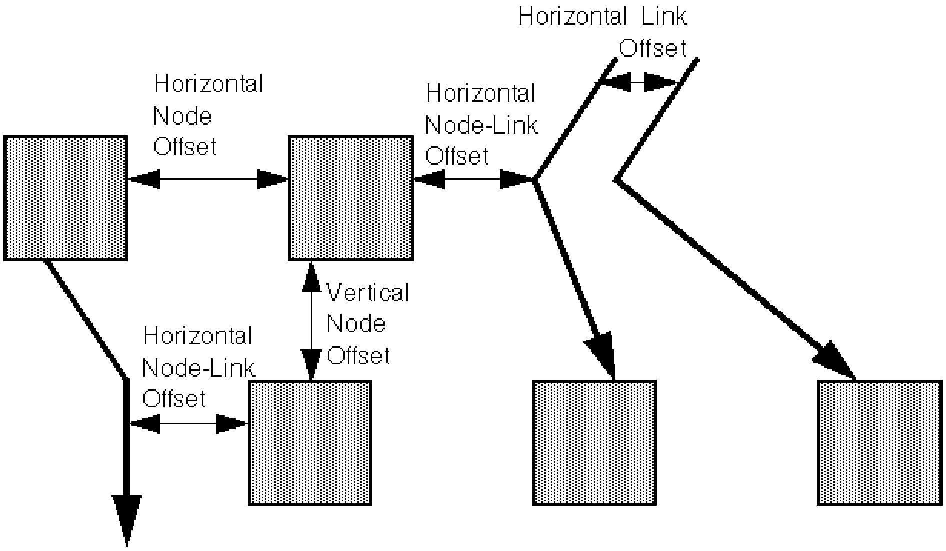


Link D -> B has a lower priority than the other links.

All links have the same priority.

Spacing Parameters

The spacing of the layout is controlled by three kinds of spacing parameters: the minimal offset between nodes, the minimal offset between parallel segments of links and the minimal offset between a node border and a bend point of a link or a link segment that is parallel to this border. The offset between parallel segments of links is at the same time the offset between bend points of links. All three kinds of parameters occur in both directions: horizontally and vertically.



To set the spacing parameters for the horizontal direction, use the properties:

- HorizontalNodeOffset
- HorizontalLinkOffset
- HorizontalNodeLinkOffset

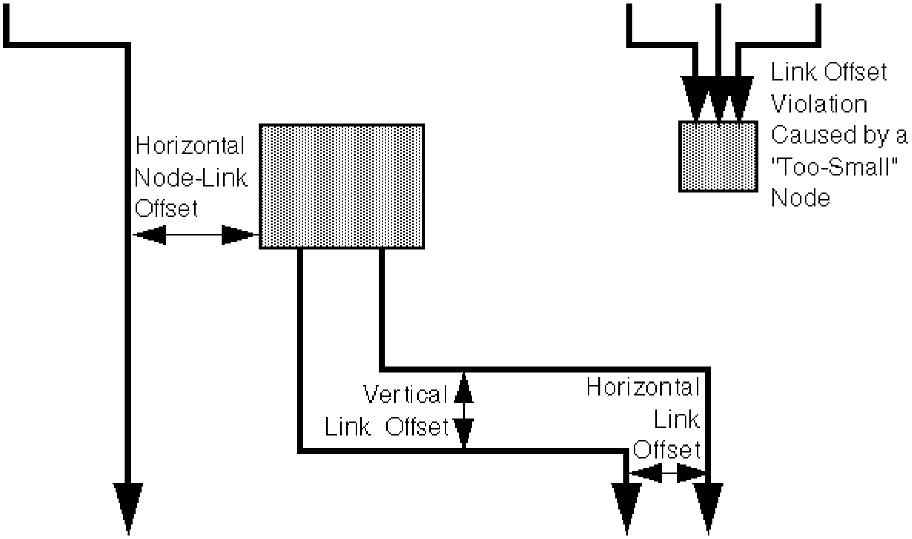
To set the spacing parameters for the vertical direction, use the properties:

- VerticalNodeOffset
- VerticalLinkOffset
- VerticalNodeLinkOffset

For a layout with horizontal levels (the flow direction is to the top or to the bottom), the horizontal node offset is the minimal distance between nodes of the same level. The vertical node offset is the minimal distance between nodes of different levels, that is, the minimal distance between the levels. For non-orthogonal link styles, the horizontal link offset is basically the minimal distance between bend points of links. The horizontal node-link offset is the minimal distance between the node border and the bend point of a link. For horizontal levels, the vertical link offset and the vertical node-link offset play a role only if the link shapes are orthogonal.

Similarly, for a layout with vertical levels (the flow direction is to the left or to the right), the vertical node offset controls node distances within the levels. The horizontal node offset is the minimal distance between the levels. In this case, the vertical link offset and the vertical node-link offset always play a role, while the horizontal link offset and the horizontal node-link offset affect the layout only with orthogonal links.

For orthogonal links, the horizontal link offset is the minimal distance between parallel, vertical link segments. The vertical link offset is the minimal distance between parallel, horizontal link segments. However, the layout algorithm cannot always satisfy these offset requirements. If a node is very small but has many incident links, it may be impossible to place the links orthogonally with the specified minimal link distance on the node border. In this case, the algorithm places some link segments closer than the specified link offset.


Horizontal Node-Link Offset
Link Offset Violation Caused by a "Too-Small" Node
Vertical Link Offset
Horizontal Link Offset

Incremental Mode

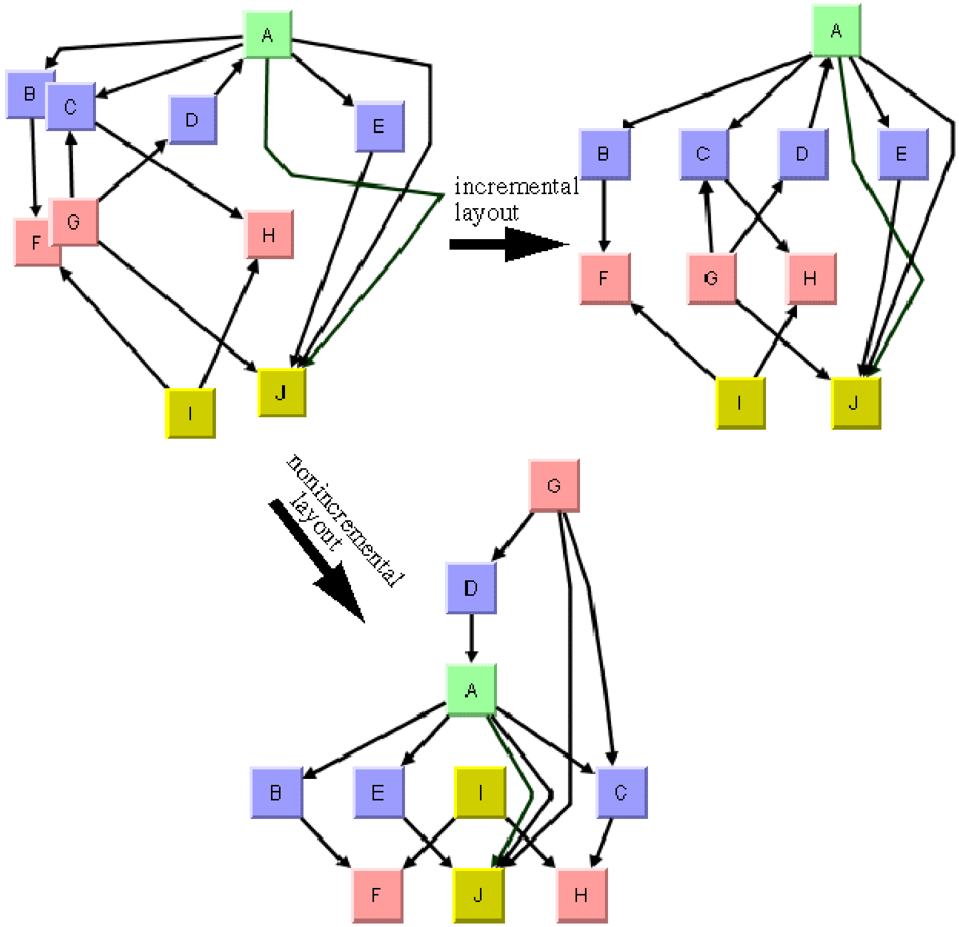
In some circumstances you may need to use a sequence of layouts on the same graph. For example:

- You work with graphs that have become out-of-date and you need to extend the graph. If you perform a layout on the extended graph, you probably want to identify the parts that were already laid out in the original graph. The layout should not change very much when compared with the layout of the original graph.
- The first layout results in a drawing with minor deficiencies. You want to solve these deficiencies manually and perform a second layout to clean up the drawing. The second layout probably should not greatly change the parts of the graph that were already acceptable after the first layout.

The Hierarchical Layout normally works non incrementally. It performs a layout from scratch and moves all nodes to new positions and reroutes all links. The previous positions of nodes have no influence on the result of the layout. Hence, even a small change can cause a large effect on the next layout.

But the Hierarchical Layout also supports incremental sequences of layout that "do not change very much." It can place the nodes close to their previous positions, so that you can more easily identify the parts that had already been laid out in the original graph. Incremental mode takes the previous positions of the nodes into account. In this mode the algorithm preserves the relative order of the levels and the nodes within the levels in the subsequent layout. It does not preserve the absolute positions of the nodes, but it tries to detect the structure of the previous layout by examining the node coordinates. For instance, if two nodes are in the same level, then they stay in the same level after an incremental layout. If a node is in a higher level than another node, it stays in the higher level.

The following illustration shows the difference between an incremental and nonincremental layout.



Incremental mode is disabled by default. To enable incremental mode, set the IncrementalMode property to **true**.

Phases of the Incremental Mode

The layout algorithm analyzes the drawing in incremental mode in the following way:

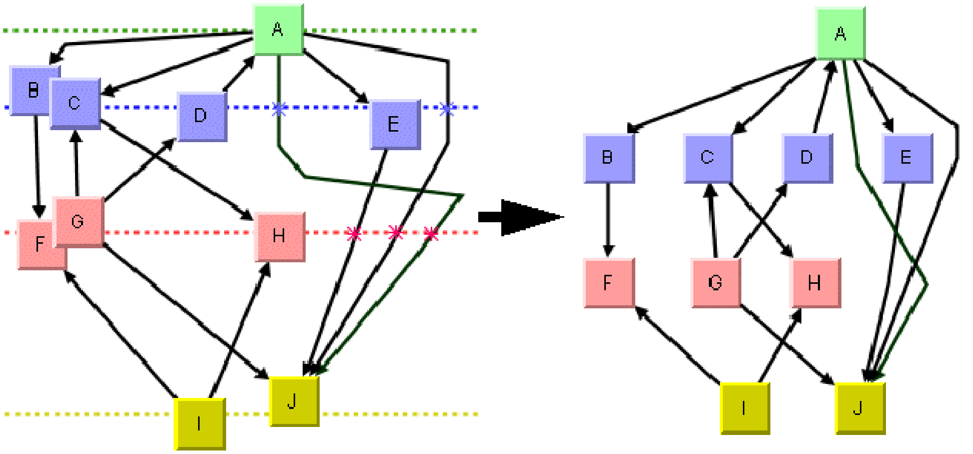
1. First, it determines from the node coordinates which nodes must belong to the same level. For instance, if the flow direction is towards the bottom, it tries to detect horizontal reference lines at those vertical positions where many nodes are placed along a line. The specified vertical node offset helps to detect these lines because the horizontal reference lines should be approximately the vertical node offset apart.
2. All nodes that touch the same reference line are assigned to the same level.

3. It determines the order of the nodes within each level by analyzing where the node touches the reference line. For instance, if the flow direction is towards the bottom, it determines from the x coordinate of the nodes how they are ordered within the levels.

4. If long links span several levels, the algorithm can preserve the shape of a long link. It determines the point where a link crosses the level reference line. It creates a bend point for the long link inside the level. It tries to preserve the order of the bend points in each level. For instance, if in a flow direction towards the bottom, a long link bypasses another node on the right side, then the incremental layout tries to find a similar shape of the link that bypasses the node on the right side, as illustrated in the following illustration.

5. Finally, the layout tries to calculate the absolute positions of the nodes that respect the levels and the ordering within the levels. It tries to balance the node positions. However, it also tries to place each node close to its previous position. Both criteria often compete with each other, because to get a perfect balance, nodes must sometimes move far from their original position. The Hierarchical Layout contains a parametrized heuristic to satisfy both criteria.

The following illustration shows incremental layout phases.



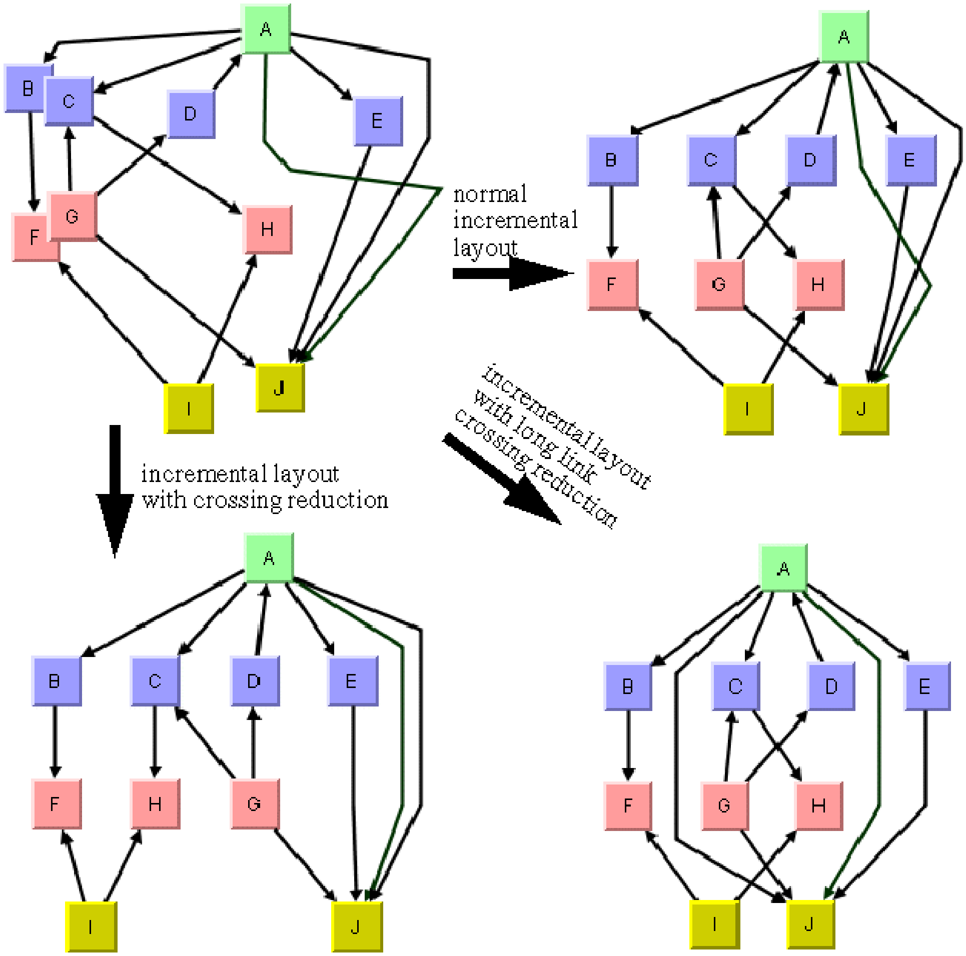
Expert Parameters of the Incremental Mode

Each phase of the incremental mode can be parameterized. These layout parameters have an effect only if incremental mode is switched on.

Minimizing Long Link Crossings

As discussed in *Phases of the Incremental Mode*, the incremental layout tries to preserve the shape of long links that cross several levels. This implies that link crossings between long links are not resolved. If crossings of long links are not desired, it may be better to reroute long links from scratch. The following illustration has four hierarchy trees, with the original layout at the upper left. The bottom right shows the result if long links are rerouted, and the top right shows the result if the shape of long links are preserved.

The following illustration shows crossing reduction during incremental layouts.


A
B
C
D
E
F
G
H
I
J
normal incremental layout
incremental layout with crossing reduction
incremental layout with long link crossing reduction

To reroute long links from scratch, you must enable the crossing reduction mechanism for long links, to do so, set the LongLinkCrossingReductionDuringIncremental property to **true**.

The crossing reduction of long links determines only the shape of the links. It does not influence the order of the other nodes within the levels.

Minimizing All Link Crossings

Optionally, you can apply a crossing reduction to all nodes within each level. In this case, the incremental layout determines from the node coordinates which nodes belong to the same level, but it may reorder the nodes within the levels completely to avoid link crossings. It also reorders the long links in this case. The illustration (bottom left) in *Minimizing Long Link Crossings* shows the result. Notice that the order of the nodes "F," "G," and "H" have changed to resolve the link crossings.

To enable the crossing reduction for all nodes set the CrossingReductionDuringIncremental property to **true**.

Setting Absolute Level Positioning

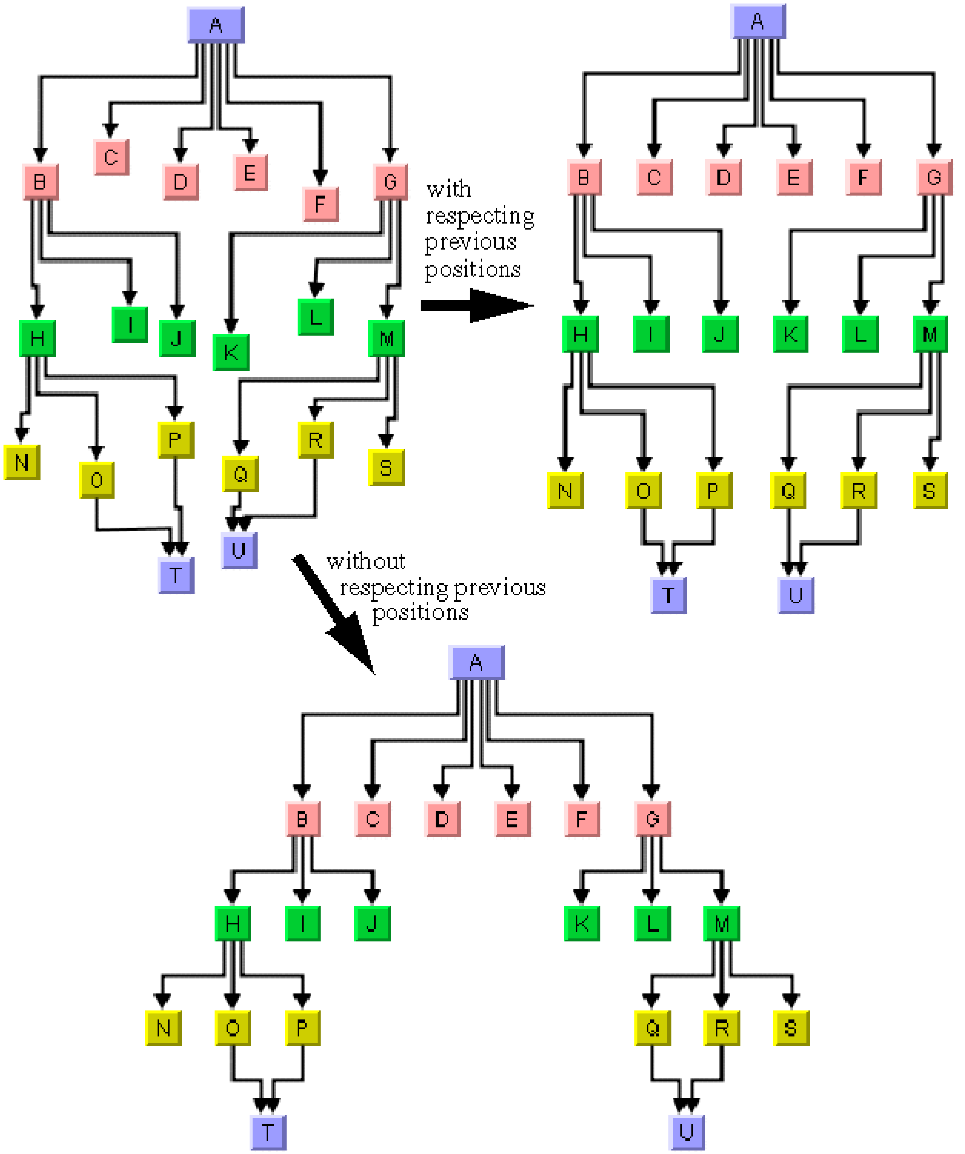
As discussed in *Phases of the Incremental Mode*, the incremental layout tries to place the nodes in absolute positions that are close to the previous positions. It tries to avoid nodes moving a large distance, because even if the relative order of the nodes within the levels does not change, large movement distances can be confusing for users. It is much easier to keep a mental map of the diagram if the nodes remain close to the previous positions.

The illustration below shows node repositioning with and without taking the previous positions into account. The incremental layout of the original graph at the top left results in the graph at the top right, which is easier to recognize as the same graph than the graph at the bottom.

The absolute level positioning feature is enabled by default, but it can be disabled. To do so set the IncrementalAbsoluteLevelPositioning property to **false**.

With this statement, the layout does not try to place the nodes close to the previous positions. It places the nodes such that the layout is balanced. However, to create a perfect balance, the layout may need to move a few nodes so far apart that you can no longer recognize the diagram after the layout from the node positions that were shown in the previous layout (see the bottom of the following illustration).

The following illustration shows the absolute positioning during incremental layouts.



Setting Absolute Level Position Range and Tendency

If absolute level positioning is enabled, it competes with the aesthetic criteria to create a balanced layout. Due to the fact that nodes must stay close to their previous positions, the diagram after incremental layout may be somewhat unbalanced and unsymmetrical. The Hierarchical Layout algorithm uses a heuristic that you can influence by two parameters, the absolute level position range and tendency.

The absolute level positioning feature is enabled by default, but it can be disabled. To do so use the IncrementalAbsoluteLevelPositionRange property.

```
layout.IncrementalAbsoluteLevelPositionRange = 100;
```

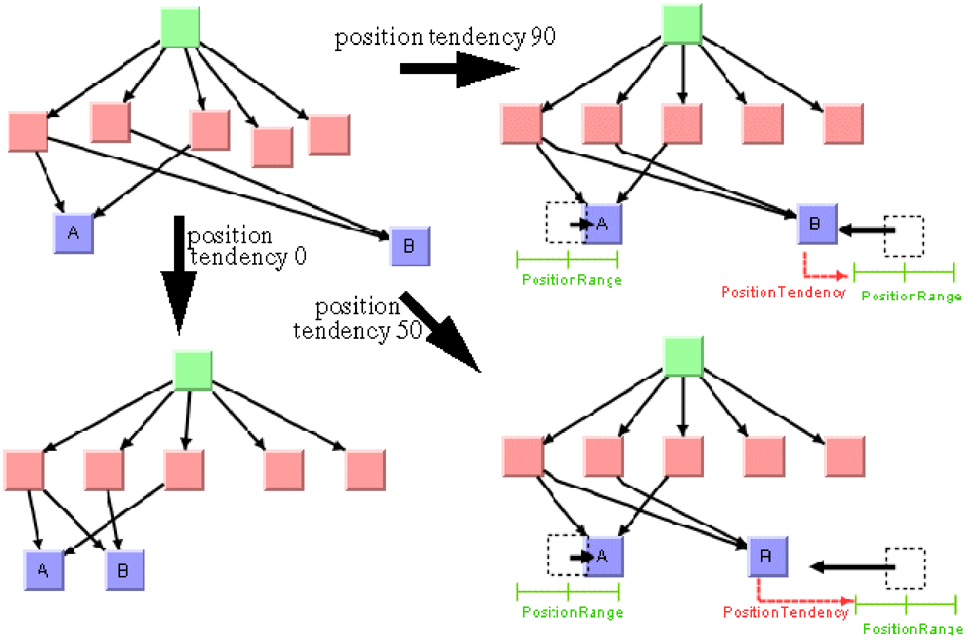
This statement specifies that within the range of 100 coordinate units from the old position of the node, the balance is the only criteria for the placement. This means that a node whose optimal position is less than 100 coordinate units away from its previous position is placed exactly at its optimal position. Nodes whose optimal position is farther away are placed at a position that is a compromise between previous position and optimal position. This is illustrated in the following illustration (on the right side).

To set the absolute level position tendency, use the IncrementalAbsoluteLevelPositionTendency property.

```
layout.IncrementalAbsoluteLevelPositionTendency = 70;
```

This statement specifies that positions of nodes whose optimal positions are far away from their previous position are 70% influenced by their previous position and 30% influenced by their optimal positions. Imagine a rubber band that tries to pull a node to its previous position, and another rubber band that tries to pull the same node to its optimally balanced position. The level position tendency 70 means that one rubber band pulls with 70% of the force towards the previous position, and the other rubber band pulls with 30% towards the optimal position. Increasing the tendency means that the node stays closer to its old position, decreasing it means that the node moves closer to its optimal position. If you set the tendency to 0%, this has the same effect as disabling the incremental absolute level positioning.

The following illustration shows the absolute positioning during incremental layouts.



Marking Nodes for Incremental Layout

Incremental layout normally treats all nodes and links of the drawing in the same way. However, you may have added nodes and links to the drawing programmatically, and the new nodes and links do not have meaningful coordinates yet. Perhaps you have placed them all at the origin (0, 0), or at random coordinates. In this case, you need an incremental layout that takes the coordinates of all nodes into account that were previously laid out, while it ignores the coordinates of all new nodes. The incremental mode of the Hierarchical Layout allows you to specify which nodes cannot be laid out incrementally by calling the method MarkForIncremental(nodeOrLink).

If you call this statement, the node or link is marked such that its coordinates are ignored during the next incremental layout. The positions of marked nodes and links are calculated from scratch. The mark is valid only until the next layout and is automatically cleared afterwards.

Layout Constraints

The Hierarchical Layout algorithm supports relative position constraints on nodes. Such a constraint is a rule on how a particular node (or a group of nodes) must be placed with respect to the other nodes. The constraints influence the relative positions. For example, you can force node A to be on the left side of node B, so the position of A is expressed relative to the position of B. It is theoretically possible to specify contradicting constraints: if you specify that node A must be on the left side of B and B must be on the left side of A, then these constraints are not solvable at the same time. If A is on the left side of B, then B must be on the right side of A. The Hierarchical Layout algorithm tries to detect and resolve constraint conflicts automatically. It ignores those constraints that are infeasible. Since the automatic constraint resolution is time consuming, it is recommended to specify nonconflicting constraints when possible.

Constraints should be used only if the incremental mode is switched off. In fact, the incremental mode is implemented by means of additional constraints that are added internally. Hence, if you use constraints during the incremental mode, it is very likely that the system detects so many constraint conflicts that you get unexpected results.

Constraints should be used carefully. The more constraints are specified, the more difficult it is to calculate a layout. Therefore, this resulting layout may have more link crossings and be less balanced than a graph with no constraints.

Each type of constraint is represented by a subclass of HierarchicalConstraint. The following constraint types are available:

HierarchicalLevelRangeConstraint	Forces a node into a range of certain levels.
HierarchicalSameLevelConstraint	Forces two nodes to the same level.
HierarchicalRelativeLevelConstraint	Forces a node to a lower/higher level than another node.
HierarchicalGroupSpreadConstraint	Forces a group of nodes on levels that are no more than a specified spread value apart.
HierarchicalRelativePositionConstraint	Forces a node to a lower/higher position than another node of the same level.
HierarchicalSideBySideConstraint	Forces two nodes of the same level to be placed side by side.
HierarchicalExtremityConstraint	Forces a node to the first or last level, or to the first or last position within a level.
HierarchicalSwimLaneConstraint	Forces a group of nodes into the same rectangular *swim lane* area.

The uses of these classes are explained in the following topics:

- *Adding and Removing Constraints*
- *Node Groups*
- *Level Range Constraints*
- *Level Index Parameter*

- *Same Level Constraints*
- *Group Spread Constraint*
- *Relative Level Constraint*
- *Position Index Parameter*
- *Relative Position Constraints*
- *Side-By-Side Constraints*
- *Extremity Constraints*
- *Swim Lane Constraint*
- *Constraint Priorities*
- *For Experts: Constraint Validation*

Adding and Removing Constraints

To add a constraint in the Hierarchical Layout, allocate a new constraint object and add it in the constraint collection of the HierarchicalLayout instance:

```
Layout.Constaints.Add(constraint)
```

You can add as many constraints as you want. The constraints will be respected during the subsequent calls of layout until you remove them.

To remove a specific constraint, use the method **Layout.Constraints.Remove(constraint)**. To remove all existing constraints, use the method **Layout.Constraints.Clear()**.

Node Groups

Some constraints affect single nodes. Other constraints affect groups of nodes. The class HierarchicalNodeGroup is a convenient way to specify a group of nodes. You can create a group of nodes in the following way:

```
HierarchicalNodeGroup group = new HierarchicalNodeGroup();
while (...) {
    group.Add(node);
}
Dim group As HierarchicalNodeGroup = New HierarchicalNodeGroup
While …
 group.Add(node)
End While
```

A node group is a collection. You can ask for the size and elements of the group, remove elements from the group, or check whether a node already belongs to the group. You can also convert a list of nodes into a group:

group.Add(node)	Adds a node to the group.
group.Remove(node)	Removes a node from the group.
group.Contains(node)	Checks whether a node is in the group.
group.Count	Returns the number of nodes in the group.
group.GetEnumerator()	Returns the nodes of the group as an Enumerator.
group = new HierarchicalNodeGroup(arrayList)	Creates a new group that contains the nodes stored in the array list.

Level Range Constraints

In Step 1 of the layout algorithm (the leveling phase), the nodes are partitioned into levels. These levels are indexed starting from 0. For instance, when the flow direction is to the bottom, the nodes of the level index 0 are placed at the topmost horizontal level line and the nodes with larger level index are placed at a position lower than the nodes with smaller level index. The layout algorithm calculates these level indices automatically.

You can choose how the levels are partitioned by specifying the range of the level index for some nodes. The nodes are placed in the levels whose index is in the specified range. You have to specify the minimal and maximal index of the level.

For example:

```
layout.Constraints.Add(new HierarchicalLevelRangeConstraint(node, 5, 7));
```

Notice that in this case, node contains the graphic node (subclass of GraphicObject). If you want to place the node exactly at level 5, call:

```
layout.Constraints.Add(new HierarchicalLevelRangeConstraint(node, 5, 5));
```

Alternatively, you can call

```
layout.SetSpecNodeLevelIndex(node, 5)
```

which has exactly the same meaning.

If you want to force the node to level 5 and above, set -1 as the maximal level:

```
layout.Constraints.Add(
  new HierarchicalLevelRangeConstraint(node, 5, -1));
```

If you want to force the node to level 5 and below (that is, level 0, ..., 5), set -1 as the minimal level:

```
layout.Constraints.Add(
  new HierarchicalLevelRangeConstraint(node, -1, 5));
```

In this particular case, you could also use zero (0) as the minimal level because the level indices start at 0.

You can apply the constraint to a group of several nodes at once. This has the same effect as specifying the constraint for each single node of the group, but it is more memory efficient and convenient. For instance, if you want to force the group of three nodes to the levels between 5 and 7:

Create a HierarchicalNodeGroup object (see *Node Groups*) of the three nodes and add it to the constraint in the following way:

```
layout.Constraints.Add(new HierarchicalLevelRangeConstraint(nodeGroup, 5, 7));
```

Level Index Parameter

The level index is a special case of a level range constraint (see *Level Range Constraints*). It forces the node to one particular level. For your convenience, you can specify the level index of a node directly by means of the method SetSpecNodeLevelIndex(Object node, int index).

You pass a single node as the first argument (not a node group). The default index value is -1. If the default value is used, or if a node is set to a negative level index, the level index is considered to be unspecified. In this case the layout algorithm automatically calculates an appropriate level index during the leveling phase of the algorithm.

To obtain the specified level index for a node, use the method GetSpecNodeLevelIndex(Object node).

However, this method returns the value that was set by **SetSpecNodeLevelIndex**. If the level index was specified by allocating a corresponding level range constraint that has the same meaning, **GetSpecNodeLevelIndex** still returns -1.

> ***Warning:*** *Using arbitrarily large level indices is not recommended. For instance, if you set the level index of a node to 100000, the layout algorithm creates 100,000 levels even if the graph has far fewer nodes. This causes the layout algorithm to become unnecessarily slow.*

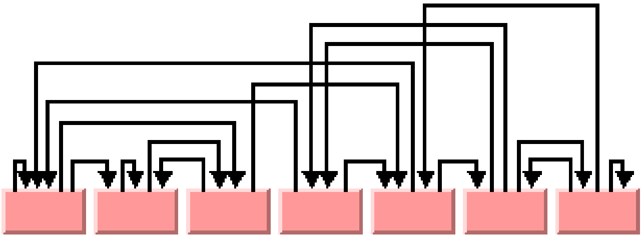
Same Level Constraints

If you want to force several nodes to the same level with fixed index, you can set the level index parameter of these nodes accordingly (see *Level Index Parameter*) or use a level range constraint (see *Level Range Constraints*). However, if you want to force several nodes to the same level without forcing them to a specific level index, you cannot use these mechanisms. You must use a same level constraint. To do so, call:

```
layout.Constraints.Add(new HierarchicalSameLevelConstraint(node1, node2));
```

This forces node1 and node2 to be placed into the same level, but it does not constrain them to any particular level.

The following illustration shows the placement of nodes on the same level.



Group Spread Constraint

An alternative way to force a group of nodes to the same level is by specifying a group spread constraint with a spread size of zero (0). In general, the group spread constraint forces a group of nodes to k+1 subsequent levels. The number k is the spread size. It does not select the lowest or highest level index of the group, but only requires that the nodes be placed no more than k levels apart. Hence, if k=0, all nodes of the group are placed at the same level.

The following example illustrates the general group spread constraint on nodes with ID "nodeA', "nodeB" and "nodeC":

To use the group spread constraint on graphic nodes (subclasses of GraphicObject), call:

```
HierarchicalNodeGroup nodeGroup = new HierarchicalNodeGroup ();
nodeGroup.Add(nodeA);
nodeGroup.Add(nodeB);
nodeGroup.Add(nodeC);
layout.Constraints.Add(new HierarchicalGroupSpreadConstraint(nodeGroup, 2));
Dim nodeGroup As HierarchicalNodeGroup = New HierarchicalNodeGroup
nodeGroup.Add(nodeA)
nodeGroup.Add(nodeB)
nodeGroup.Add(nodeC)
layout.Constraints.Add(New HierarchicalGroupSpreadConstraint(nodeGroup, 2))
```

The constraint is satisfied if the highest level index for nodeA, nodeB, and nodeC is no more than two levels apart from the smallest level index of the nodes. For instance, the constraint is satisfied if the level indices for nodeA, nodeB, and nodeC are 1, 2, 3; or if they are 7, 8, 9; or if they are 16, 14, 15. The constraint is also satisfied if all three nodes are placed at level 5, or if two of the nodes are placed at level 15 and the third node at level 13. The constraint is not satisfied if the level indices for nodeA, nodeB, and nodeC are 3, 5, 6, because in this case the highest index (6) is more than two levels away from the lowest index (3).

Relative Level Constraint

You can force a node into a higher level than another node. If the flow direction is towards the bottom, level 0 is topmost in the drawing. In this layout you can specify by relative level constraints that a node be above or below another node. If the flow direction is towards the right, level 0 is leftmost in the drawing. Here you can specify by relative level constraints that a node be left or right of another node.

For example:

```
layout.Constraints.Add(
    new HierarchicalRelativeLevelConstraint(nodeA, nodeB, priority));
```

This forces nodeA to be placed at a level with a smaller index than nodeB. Since relative level constraints compete with each other, you must specify the priority of the constraint. In fact, links also impose constraints on the system, and the link priority has the same impact as the constraint priority. A link with priority 10 forces its (usually) source node (unless ports are specified) into a lower level than its target node. To force the source node into a higher level than the target node, you need to create a constraint with a higher priority than the link. For instance, to ensure that the constraints are satisfied even if there are many links, you can use link priorities between 0 and 10 and constraint priorities between 1000 and 10,000.

You can also create a relative level constraint between groups of nodes.

```
layout.Constraints.Add(
    new HierarchicalRelativeLevelConstraint(nodeGroup1, nodeGroup2, priority));
```

Position Index Parameter

In Step 2 of the layout algorithm (the crossing reduction phase), the nodes are ordered within the levels. All nodes that belong to the same level get a position index starting from 0. For instance, when the flow direction is to the bottom, the node with the position index 0 is placed in the leftmost position within its level. The nodes with a larger position index are placed farther to the right than the nodes with a smaller position index in the same level. The nodes of different levels are independent. The node of the first level with the position index 0 is to the left of the node of the first level with the position index 1, but not necessarily to the left of a node of another level with position index 0. Note that long links crossing a level also obtain a position index. The layout algorithm calculates these position indices automatically.

You can affect how the nodes are positioned within each level by specifying the position index of some nodes. The nodes are placed at the specified position within their level.

To specify the position index of a node, use the method SetSpecNodePositionIndex(Object node, int index).

The default value is -1. If the default value is used, if a node is set to a negative position index, or if a node is set to a position index that is larger than the number of nodes of its level, the layout automatically calculates an appropriate position index during the crossing reduction step.

To obtain the current position index of a node, use the method GetSpecNodePositionIndex(Object node).

Relative Position Constraints

Working with absolute node position indices is inconvenient in certain situations. For instance, if two nodes belong to the same level, you may want to force one node to a position with a lower index than the other node without fixing the absolute positions of the nodes. You can achieve this by using a relative position constraint.

```
layout.Constraints.Add(
    new HierarchicalRelativePositionConstraint(nodeA, nodeB, priority));
```

This forces nodeA to a lower position than nodeB. If the flow direction is towards the bottom, the nodes are in horizontal levels; hence the constraint means that nodeA is placed at the left side of nodeB. If the flow direction is towards the right, the nodes are in vertical levels; hence the constraint means that nodeA is placed below nodeB.

The relative position constraint has an effect only if both nodes actually belong to the same level. To achieve this, you can, for instance, use a same level constraint in addition. There is no way to influence the relative position of nodes that belong to different levels.

Similar to the relative level constraint, the relative position constraint can be applied to node groups. These constraints also have priorities that indicate which constraints dominate if a constraint conflict occurs. The higher the priority, the more likely the constraint is satisfied when resolving constraint conflicts.

Side-By-Side Constraints

The relative position constraint forces a specific order upon the nodes of a level, but it does not specify which nodes are directly neighbored. For instance, a relative position constraint may force nodeA to be placed somewhere at a lower position than nodeB, but there may be many nodes between nodeA and nodeB.

To force nodes to be directly neighbored, use the side-by-side constraint.

You can create a side-by-side constraint on a group of type HierarchicalNodeGroup (*Node Groups*):

```
layout.Constraints.Add(
    new HierarchicalSideBySideConstraint(nodeGroup, priority));
```

If the node group consists of just two nodes, it forces the two nodes to be placed side by side. However, it does not specify which node is at the lower node position and which node is at the higher node position. If the group consists of more than two nodes, it forces the nodes to be placed at consecutive positions such that all nodes are clustered together. A node that does not belong to the group cannot be placed between the nodes of the group.

For instance, assume that the group contains the three nodes A, B, C. The constraint is satisfied if the position indices of A, B, and C are 3, 4, 5 or 9, 7, 8. However, if node D is placed between A and B (say, D has position 4, A has position 3, and C has position 5), then the constraint is not satisfied because D does not belong to the same group.

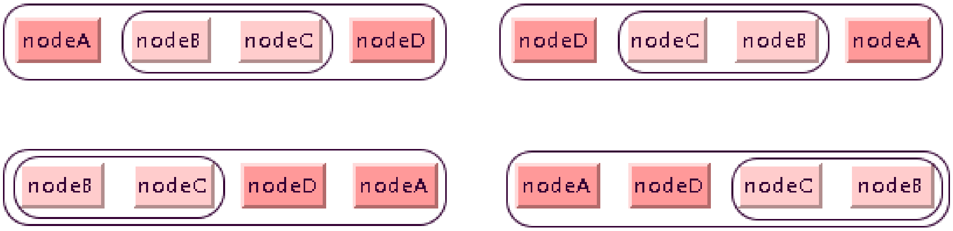
The side-by-side constraint has an effect only if the nodes actually belong to the same level. To achieve this, you can, for instance, use a same level constraint in addition.

Side-by-side constraints have priorities that decide how to resolve constraint conflicts. The higher the priority, the more likely the constraint is satisfied.

You can use side-by-side constraints to create nested clusters. For example:

```
HierarchicalNodeGroup group1 = new HierarchicalNodeGroup();
group1.Add(nodeA);
group1.Add(nodeB);
group1.Add(nodeC);
group1.Add(nodeD);
layout.Constraints.Add(
    new HierarchicalSideBySideConstraint(group1, 10.0f));
HierarchicalNodeGroup group2 = new HierarchicalNodeGroup();
group2.Add(nodeB);
group2.Add(nodeC);
layout.Constraints.Add(
    new HierarchicalSideBySideConstraint(group2, 10.0f));
Dim group1 As HierarchicalNodeGroup = New HierarchicalNodeGroup
group1.Add(nodeA)
group1.Add(nodeB)
group1.Add(nodeC)
group1.Add(nodeD)
layout.Constraints.Add(New HierarchicalSideBySideConstraint(group1, 10F))
Dim group2 As HierarchicalNodeGroup = New HierarchicalNodeGroup
group2.Add(nodeB)
group2.Add(nodeC)
layout.Constraints.Add(New HierarchicalSideBySideConstraint(group2, 10F))
```

The first constraint specifies that nodeA, nodeB, nodeC, and nodeD must be clustered. The second constraint specifies that nodeB and nodeC are clustered inside the larger cluster. This means that no other node can be placed between the four nodes and, furthermore, neither nodeA nor nodeD can be placed between nodeB and nodeC. The following illustration sketches four solutions that satisfy both constraints.



Extremity Constraints

To force a node to the first level, you can specify:

```
layout.SetSpecNodeLevelIndex(node, 0);
```

However, you cannot specify a level index for the last level because it is unknown at the beginning of layout how many levels will be created. It is unwise to specify:

```
layout.SetSpecNodeLevelIndex(node, int.MaxValue);
```

because this will create many empty levels between the levels actually used and the last one. Even though these empty levels are removed in postprocessing steps, this influences the speed and quality of the layout. (In fact, the algorithm will run out of memory if you set the specified level index unreasonably high.)

By using constraints you can achieve the same effect more efficiently. To force a node to the first level use:

```
layout.Constraints.Add(
    new HierarchicalExtremityConstraint(node, HierarchicalLayoutSide.North));
```

To force a node to the last level, use:

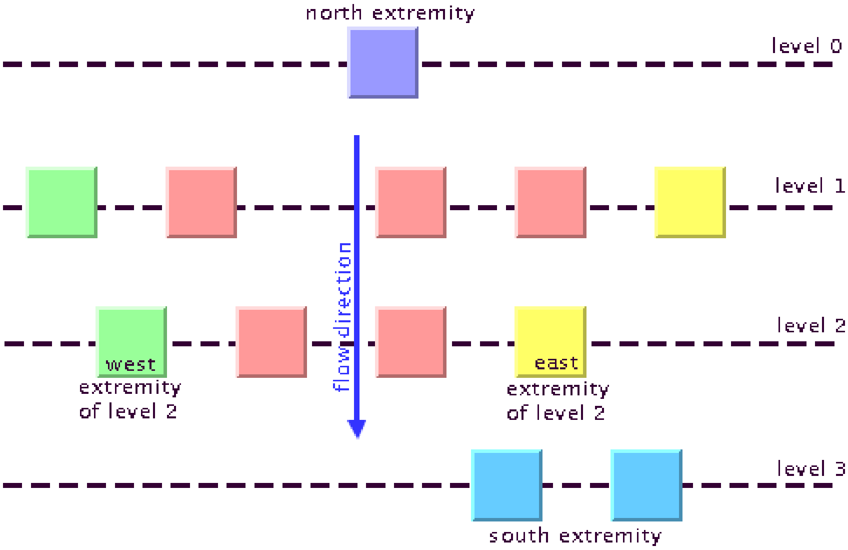
```
layout.Constraints.Add(
    new HierarchicalExtremityConstraint(node, HierarchicalLayoutSide.South));
```

With compass directions as a convenient reference (see *Port Sides Parameter*), the first level indicates the north pole and the last level indicates the south pole. You can also specify extremity constraints for the east and west sides:

```
layout.Constraints.Add(
    new HierarchicalExtremityConstraint(node1, HierarchicalLayoutSide.East));
layout.Constraints.Add(
    new HierarchicalExtremityConstraint(node2, HierarchicalLayoutSide.West));
```

The west extremity constraint forces the node to the lowest position index within its level, and the east extremity constraint forces the node to the highest position index within its level. The position indices specify the relative position within the level. For instance, a node with west extremity constraint will be the leftmost node within its level, if the flow direction is towards the bottom. However, this does not affect other levels; there may be a node in another level that is still placed farther to the left.

The following illustration shows some extremity constraints.



Swim Lane Constraint

Swim lanes are rectangular areas orthogonal to the levels.

If the link flow direction is towards the bottom or top, the levels are horizontal rows and the *swim lane*s are vertical columns.

If the flow direction is towards the left or right, the levels are vertical columns and the swim lanes are horizontal rows.

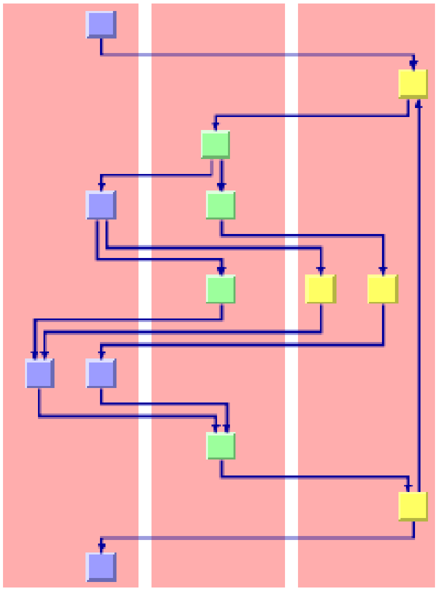
Swim lanes can be used if the nodes are partitioned into groups, to indicate which nodes belong to a certain group. The nodes of the same swim lane are placed so that it is possible to draw a surrounding rectangle around them. Swim lanes allow you to organize the graph in a table-like manner. For instance, you may have a workflow diagram where nodes represent actions; then the swim lanes could represent the departments that perform these actions. Each node can belong to only one swim lane.

To associate a group of nodes with the same swim lane, call:

```
layout.Constraints.Add(
     new HierarchicalSwimLaneConstraint(new HierarchicalNodeGroup(nodelist)));
```

All nodes of the node list will be placed in the same swim lane rectangle. If a graph has many swim lane rectangles, the relative order of these swim lanes is determined automatically. The size of the swim lane rectangle depends on the nodes that belong to the swim lane. However, you can specify the relative order, relative size, and the margins of the swim lane as well by using the constructor:

```
public HierarchicalSwimLaneConstraint(HierarchicalNodeGroup group,
float relativeSize,
int positionIndex,
float minMargin)
```



The red background rectangle indicates where the swim lane is.

The relative size indicates how large this swim lane is compared to the other swim lanes. Assume that the flow direction is towards the bottom. In this case, the relative size indicates the width of the swim lane. All swim lanes with the same relative size will have the same width. A swim lane with a relative size that is twice the value of another swim lane will have twice the width of the other swim lane. The actual number of this parameter does not matter, only how large the value is compared to the other swim lanes. If you do not want to restrict the size of the swim lane, set the value to 0. In this case, the width of the swim lane will be independent of the other swim lanes.

The minimal margin is the margin of the swim lane in absolute coordinates. If the flow direction is towards the bottom, then it is the minimal horizontal distance between the leftmost or rightmost node of the swim lane and the swim lane border.

The position index indicates the order of the swim lanes. Just as nodes have position indices, the swim lanes are placed sequentially at relative positions numbered from 0 to n. In a top-down layout, the swim lane with position 0 is the leftmost swim lane, and the swim lanes with higher position indices are placed farther to the right. If the swim lanes have the position index -1, the layout algorithm determines the appropriate position automatically.

A swim lane constraint is always evaluated, even if the incremental mode is enabled. The constraint has a higher priority than the relative position constraint and the side-by-side constraint. You can specify side-by-side constraints for a group of nodes that belong to the same swim lane, but side-by-side constraints of nodes of different swim lanes are ignored. You can specify relative position constraints between nodes of the same swim lane. You can also specify relative position constraints between one entire swim lane group and another swim lane group, which effectively orders the swim lanes. But relative position constraints are ignored if they would require breaking the swim lanes apart. The swim lane constraint dominates the specified position indices and the extremity constraints, that is, if a swim lane constraint is used, you cannot specify position indices or east/west extremity constraints for any node.

> ***Tip:*** *The automatic conflict resolution can handle conflicting constraints. However, to speed up the layout, it is recommended that you specify constraints in such a way that there are no conflicts.*

Constraint Priorities

A set of constraints may cause conflicts. This means that not all of the constraints can be satisfied at the same time. For instance, it is impossible to force two nodes into the same level by a HierarchicalSameLevelConstraint while at the same time forcing one of the nodes to a higher level than the other node by a HierarchicalRelativeLevelConstraint. In this case, one of the two constraints must be ignored during layout.

In general, constraint conflicts are resolved by ignoring the constraints with the lowest priority while the constraints with the highest priority get satisfied. The following rules explain the constraint priorities in detail.

- The constraints that influence into which level a node is placed are applied before the constraints that influence the position of the node within a level.
- The HierarchicalExtremityConstraint is translated into a sequence of constraints with high priority. For instance, the extremity constraint with the south side is translated into several same level constraints and several relative level constraints.
- The **HierarchicalSameLevelConstraint** and the HierarchicalGroupSpreadConstraint have the highest priority. They are never in conflict with each other. They dominate all other constraints, even the specified level index.
- The HierarchicalLevelRangeConstraint (and the direct level index specification) has the second highest priority. If two nodes are forced to the same level but have disjoint specified level ranges, then the level range is ignored. In the following example:

  ```
  layout.Constraints.Add(new HierarchicalSameLevelConstraint(node1, node2));
  layout.SetSpecNodeLevelIndex(node1, 5);
  layout.SetSpecNodeLevelIndex(node2, 10);
  ```

  both node1 and node2 will be placed at level 5. The conflicting specification layout.SetSpecNodeLevelIndex(node2, 10) is ignored.

- The **HierarchicalRelativeLevelConstraint** is dominated by the same level constraint, by the level range constraint, and by the direct specification of level indices. If several relative level constraints conflict each other, the one with the highest specified priority dominates. However, note that all links are implicitly considered relative level constraints as well. If links with high priority force a node to a certain level, then a relative level constraint with lower priority will be ignored.
- The HierarchicalSwimLaneConstraint is always evaluated, even if the incremental mode is enabled. The constraint has a higher priority than the relative position constraint and the side-by-side constraint. You can specify side-by-side constraints for a group of nodes that belong to the same swim lane, but side-by-side constraints of nodes of different swim lanes are ignored. You can specify relative position constraints between nodes of the same swim lane. You can also specify relative position constraints between one entire swim lane group and another swim lane group, which effectively orders the swim lanes. But relative position constraints are ignored if they would require breaking the swim lanes apart. The swim lane constraint dominates the specified position indices and the extremity constraints, that is, if a swim lane constraint is used, you cannot specify position indices or east/west extremity constraints for any node.
- The mHierarchicalSideBySideConstraint is evaluated only if the corresponding nodes belong to the same level. Typically you will use a same level constraint to force the nodes to the same level, and then a side-by-side constraint to force the nodes to a certain ordering. The side-by-side constraints dominate the relative position constraints. If several side-by-side constraints are conflicting, the one with the highest specified priority dominates the other constraints.
- The HierarchicalRelativePositionConstraint is also evaluated only if the corresponding nodes belong to the same level. It is dominated by the side-by-side constraint; however, conflicts with side-by-side constraints are rare. If several relative position constraints are conflicting, the one with the highest specified priority dominates the other constraints.

For Experts: Constraint Validation

Constraints may become invalid. For instance, if you add a constraint that node A must be to the left side of node B, but you remove A from the graph, then this constraint becomes invalid. It simply does not make sense any more, even though it does not conflict with any other constraint. The layout instance automatically removes invalid constraints from time to time because they are a waste of memory. The validation check is done during layout. Forcing a validation check is normally not needed but if you want to perform it, call:

```
layout.ValidateConstraints();
```

This removes all invalid constraints from the Hierarchical Layout and cleans up the memory. The constraint validation does not check which constraints have conflicts. The main effect of the validation is that the constraint system uses less memory afterwards.

> ***Note:*** *A constraint is valid if it is meaningful. Two valid constraints are conflicting if the system cannot satisfy them both at the same time. Invalid constraints cannot be conflicting because they are meaningless. Hence, constraint validation and constraint resolution are different phases. Constraint validation performs a quick local test. It removes invalid constraints from the layout instance completely. It does not affect conflicting constraints. Constraint resolution checks whether a set of valid constraints are in conflict with each other. Thus, constraint resolution is a complex process on a network of multiple related constraints. Constraint resolution decides which constraints can be solved and which cannot. But the constraint resolution does not remove conflicting constraints from the layout instance, it just delivers a solution that may ignore some constraints.*

For Experts: More Indices

The Hierarchical Layout allows you to specify the level index and the position index of a node.

You specify the level and position index of a graphic node in the following way:

```
layout.SetSpecNodeLevelIndex(node, 5);
layout.SetSpecNodePositionIndex(node, 33);
```

How these indices are used depends on the graph *topology* and the additional constraints. For example, the specified level index can be in conflict with some HierarchicalLevelRangeConstraint or HierarchicalSameLevelConstraint. In this case, the constraint priorities determine how the conflict is resolved (see *Constraint Priorities*). If the incremental mode is switched on, the specified node level and position index are ignored, since the incremental mode tries to preserve old node positions. It is also possible to obtain the indices of nodes that were calculated during layout.

Calculated Level Index

The layout algorithm allows you to access the level index that was calculated for a node by a previous layout. To do this, use the method GetCalcNodeLevelIndex(Object node).

If the node was never laid out, this method returns -1. Otherwise, it returns the previous level index of the node.

The method can be used to specify the level index for the next layout in the following way:

```
int index = layout.GetCalcNodeLevelIndex(node);
layout.SetSpecNodeLevelIndex(node, index);
```

When this is done, it ensures that the node is placed at the same level as in the previous layout.

If the graph is detached from the layout algorithm, the calculated level index of a node is set back to -1.

> ***Note:*** *You should be aware of the difference between the methods **GetCalcNodeLevelIndex** and GetSpecNodeLevelIndex. The first one returns the level index calculated by the previous layout. The second one returns the specified level index, even if there was no previous layout. For instance, consider two nodes A and B. Node A has no specified level index and node B has a specified level index 5. Before the first layout, the method **GetCalcNodeLevelIndex** returns -1 for both nodes because the levels have not been calculated yet. However, **GetSpecNodeLevelIndex** returns -1 for A and 5 for B. After the first layout, node A may be placed at level 4. Now, **GetCalcNodeLevelIndex** returns 4 for node A and 5 for node B and **GetSpecNodeLevelIndex** still returns -1 for A and 5 for B.*

Calculated Position Index

The layout algorithm allows you to access the position index within a level that was calculated for a node by a previous layout. To do this, use the method GetCalcNodePositionIndex(Object node).

If the node was never laid out, this method returns -1. Otherwise, it returns the previous position index of the node within its level.

To ensure that the node is placed at the same level at the same relative position as in the previous layout, use the following line of code:

```
layout.SetSpecNodeLevelIndex(node,layout.GetCalcNodeLevelIndex(node));
```

This example code works only if the generic connected component layout is disabled and the port sides East or West are not used in the layout.

If the graph is detached from the layout algorithm, the calculated position index of a node is set back to -1.

> ***Note:*** *You should be aware of the difference between the methods **GetCalcNodePositionIndex** and GetSpecNodePositionIndex. The first one returns the position index calculated by the previous layout and -1 if there was no previous layout. The second one returns the specified position index even if there was no previous layout. This behavior is similar to the behavior of the specified and calculated level index (see Calculated Level Index)*

Tree Layout

In this section, use the following topics to learn about the Tree Layout algorithm (class TreeLayout).

In This Section

Samples

Provides some sample drawings produced by the algorithm.

What Types of Graphs?

Explains the type of graph on which you can use the algorithm.

Application Domains

Explains the application domains for the algorithm.

Features

Lists the features of the algorithm.

Limitations

Explains the limitations of the algorithm.

Brief Description of the Algorithm

Provides a short description of the tree layout algorithm.

Code Sample

Provides a sample of code that shows how to use the algorithm.

Generic Features and Parameters

Explains what generic features of the graph layout library are supported and how.

Specific Parameters (All Tree Layout Modes)

Presents the parameters of the tree layout algorithm.

Free Layout Mode

Explains the free layout mode.

Level Layout Mode

Explains the level layout mode.

Radial Layout Mode

Explains the radial layout mode.

Tip-Over Layout Modes

Explains the tip-over layout modes.

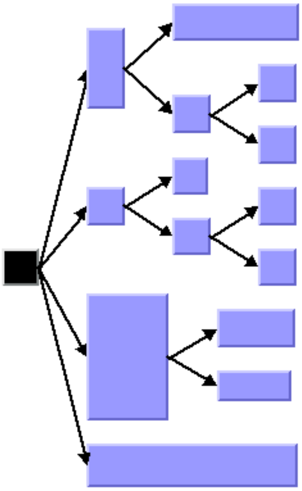
For Experts: Additional Tips and Tricks

Provide additional information for layout experts

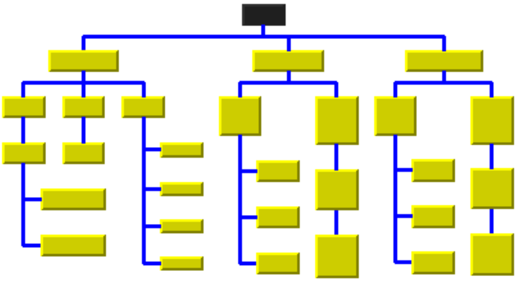
Samples

Here are some sample drawings produced with the Tree Layout.

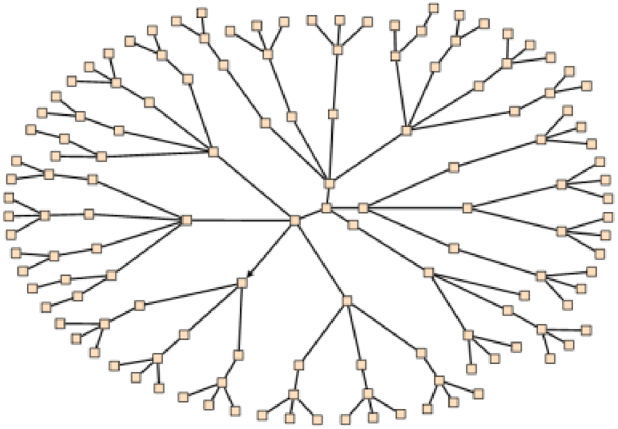
The following illustration shows a tree layout in Free Layout mode with the center alignment and flow direction to the right.



The following illustration shows a tree layout with the flow direction to the bottom, orthogonal link style, and tip-over alignment at some leaf nodes.



The following illustration shows a tree layout in Radial Layout Mode with aspect ratio 1.5.



What Types of Graphs?

- Primarily designed for pure trees, the tree layout can also be used for non-trees, that is, for cyclic graphs. In this case, the algorithm computes and uses a *spanning tree* of the graph, ignoring all links that do not belong to the spanning tree.
- Directed and undirected trees. If the links are directed, the algorithm automatically chooses the canonical root node. If the links are undirected, you can choose a root node.
- Connected and disconnected graphs. If the graph is not connected, the layout algorithm treats each *connected component* separately. Each component has exactly one root node. In this case, a forest of trees is laid out.

Application Domains

Application domains of the Tree Layout include:

- Business processing (organizational charts)
- Software management/software (re-)engineering (UML diagrams, call graphs)
- Database and knowledge engineering (decision trees)
- The World Wide Web (Web site maps)

Features

- Takes into account the size of the nodes so that no overlapping occurs.
- Optionally, reshapes the links to give them an orthogonal form (alternating horizontal and vertical line segments).

- Various layout modes: free, levels, radial, or automatic tip-over.
  - In the free layout mode, arranges the children of each node, starting recursively from the root, so that the links flow uniformly in the same direction.
  - In the level layout mode, partitions the nodes into levels, and arranges the levels horizontally or vertically.
  - In radial layout mode, partitions the nodes into levels, and arranges the levels in circles or ellipses around the root.
  - In the tip-over modes, arranges the nodes in a similar way to the free layout mode, but tries to tip over children automatically to better fit the layout to the given aspect ratio.
- Provides several alignment and offset options.
- Allows you to specify nodes that must be directly neighbored.
- Provides incremental and nonincremental modes. Incremental mode takes the previous position of nodes into account and positions the nodes without changing the relative order of the nodes in the tree so that the layout is stable on incremental changes of the graph.
- Very efficient, scalable algorithm. Produces a nice layout quickly even if the number of nodes is huge.

Limitations

- If the orthogonal setting is not specified as the link style (see *Link Style*), some links may in rare cases overlap nodes depending on the size of the nodes, the alignment parameters, and the offset parameters.
- The layout algorithm first determines a spanning tree of the graph. If the graph is not a pure tree, some links will not be included as part of the spanning tree. These links are ignored. For this reason, they may cross other links or overlap nodes in the final layout.
- For stability in incremental mode, the algorithm tries to preserve the relative order of the children of each node. It uses a heuristic to calculate the relative order from the previous positions of the nodes. The heuristic may fail if children overlap their old positions or are not aligned horizontally or vertically.
- Despite preserving the relative order of the children, in rare cases the layout is not perfectly stable in incremental radial layouts. Subsequent layouts may rotate the nodes around the root, although the relative circular order of the nodes within their circular levels is still preserved.
- The tip-over layout modes will perform several trial layouts with different tip-over alignment options according to various heuristics. From these trial layouts, the algorithm picks the layout that best fits the given aspect ratio. This may not be the optimal layout

for the aspect ratio, but it is the best layout among the trials. Calculating the absolute best-fitting layout is not computationally feasible (it is generally an NP-complete problem).

Brief Description of the Algorithm

The core algorithm for the free, level, and radial layout modes works in just two steps and is very fast.

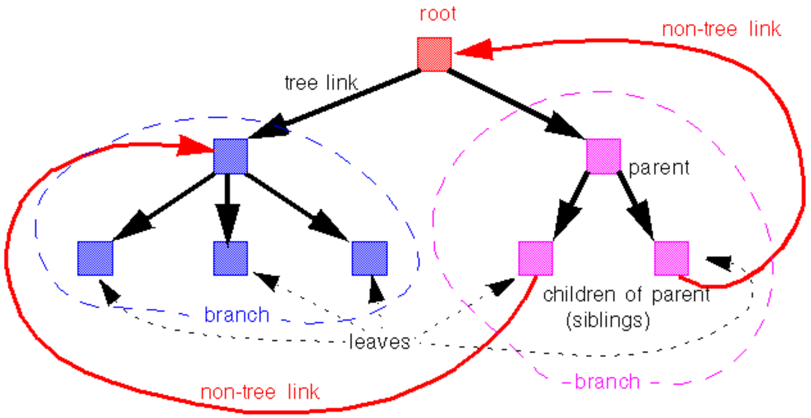
The variations of the tip-over layout mode perform the second step several times and pick the layout result that best fits the given aspect ratio (the ratio between width and height of the drawing area). For this reason, the tip-over layout modes are slower.

Step 1: Calculating the Spanning Tree

If the graph is disconnected, the layout algorithm chooses a root node for each connected component. Starting from the root node, it traverses the graph to choose the links of the spanning tree. If the graph is a pure tree, all links are chosen. If the graph has cycles, some links will not be included as part of the spanning tree. These links are called non-tree links, while the links of the spanning tree are called tree links. The non-tree links are ignored in step 2 of the algorithm.

In the spanning tree, each node except the root has a parent node. All nodes that have the same parent are called children with respect to the parent and siblings with respect to themselves. Nodes without children are called leaves. Each child at a node starts a subtree (also called a branch of the tree).

The following illustration shows a spanning tree.



Step 2: Calculating Node Positions and Link Shapes

The layout algorithm arranges the nodes according to the layout mode and the offset and alignment options. In the free mode and level mode, the nodes are arranged horizontally or vertically so that all tree links flow roughly in the same direction. In the radial layout modes, the nodes are arranged in circles or ellipses around the root so that all tree links flow radially away from the root. Finally, the link shapes are calculated according to the link style and alignment options.

Code Sample

Below is a code sample that uses the TreeLayout class. This code sample shows how to perform a Tree Layout on a Group object directly.

```
using System;
using ILOG.Diagrammer;
using ILOG.Diagrammer.Graphic;
using ILOG.Diagrammer.GraphLayout;


…
Group group = new Group();

// Fill the group with nodes and links here
// Suppose we have added rootNode as an object in the group

TreeLayout layout = new TreeLayout();

group.GraphLayout = layout;
group.PerformGraphLayout();
Imports System
Imports ILOG.Diagrammer
Imports ILOG.Diagrammer.Graphic
Imports ILOG.Diagrammer.GraphLayout

Dim group As Group = New Group

' Fill the group with nodes and links here
' Suppose we have added rootNode as an object in the group

Dim layout As TreeLayout = New TreeLayout

group.GraphLayout = layout
group.PerformGraphLayout()
```

Generic Features and Parameters

The TreeLayout class supports the following generic features defined in the GraphLayout class. (See also *Layout Parameters and Features in the GraphLayout Base Class*):

- *Allowed Time*
- *Layout of Connected Components*

- *Link Clipping*
- *Link Connection Box*
- *Percentage of Completion Calculation*
- *Preserve Fixed Links*
- *Preserve Fixed Nodes*
- *Stop Immediately*

The following subsections describe the particular way in which these features are used by the subclass **TreeLayout**.

Allowed Time

The layout algorithm stops if the allowed time setting has elapsed. (For a description of this layout parameter in the **GraphLayout** class, see *Allowed Time*.)

If the layout stops early because the allowed time has elapsed, the nodes and links are not moved from their positions before the layout call and the result code in the layout report is **GraphLayoutReportCode.StoppedAndInvalid**.

Layout of Connected Components

The layout algorithm can utilize the generic mechanism to layout connected components. (For more information about this mechanism, see *Layout of Connected Components*). It has, however, a specialized internal mechanism to layout connected components and, therefore, the generic mechanism is switched off by default.

The generic connected component layout mechanism has the disadvantage that it moves connected components completely. Fixed nodes within a component do not preserve their old position, and the resulting layout may be unstable on incremental changes, depending on which layout instance is used for the component layout.

If the generic connected component layout mechanism is disabled, the algorithm uses its own specialized internal mechanism instead of the generic mechanism to lay out each component as a separate tree. This is usually faster and more stable on incremental changes than the generic mechanism. Furthermore, it enables the user to set the position of the layout.

Link Clipping

The layout algorithm can use a link clip interface to clip the connection points of a link. (See *Link Clipping*)

This is useful if the nodes have a nonrectangular shape such as a triangle, rhombus, or circle. If no link clip interface is used, the links are normally connected to the bounding boxes of the nodes, not to the border of the node shapes. See *Using the Link Clipping* for details of the link clipping mechanism.

Link Connection Box

The layout algorithm can use a link connection box provider (see *Link Connection Box*) in combination with the link clip interface. If no link clip interface is used, the link connection box provider has no effect. For details see *Using a Link Connection Box Interface*.

Percentage of Completion Calculation

The layout algorithm calculates the estimated percentage of completion. This value can be obtained from the layout report during the run of layout. (For a detailed description of this feature, see *Percentage of Completion Calculation* and *Graph Layout Event Handlers*.)

Preserve Fixed Links

The layout algorithm does not reshape the links that are specified as fixed. For more information on link parameters in the GraphLayout class, see *Preserve Fixed Links* and *Link Style*.

Preserve Fixed Nodes

The layout algorithm does not move the nodes that are specified as fixed. (For more information on node parameters in the **GraphLayout** class, see *Preserve Fixed Nodes*.) Moreover, the layout algorithm ignores fixed nodes completely and also does not route the links that are incident to the fixed nodes. This can result in unwanted overlapping nodes and link crossings. However, this feature is useful for individual, disconnected components that can be laid out independently.

Stop Immediately

The layout algorithm stops after cleanup if the method StopImmediately is called. (For a description of this method in the **GraphLayout** class, see *Stop Immediately*.) If the layout stops early because the allowed time has elapsed, the nodes and links are not moved from their positions before the layout call, and the result code in the layout report is GraphLayoutReport.StoppedAndInvalid.

Specific Parameters (All Tree Layout Modes)

The following parameters are specific to the TreeLayout class. They apply to all layout modes.

Root Node

The final layout is influenced mainly by the choice of the root node.

The root node is placed in a prominent position. For instance, in a top-down drawing with free layout mode, it is placed at the top of the tree. With the radial layout mode, it is placed at the center of the tree.

The spanning tree is calculated starting from the root node. If the graph is disconnected, the layout algorithm needs one root node for each connected component.

The layout algorithm automatically selects a root node when needed. It uses a heuristic that calculates preferences for all nodes to become a root. It chooses the node with the highest preference. The heuristic gives nodes without incoming links the highest preference and leaf nodes without outgoing links the lowest preference. Hence, in a directed tree, the canonical root is always chosen automatically.

It is possible to influence the choice of the root node. To set a node explicitly as the root use the method:

```
void SetRoot(object node)
```

In this case, the node argument must be a graphic node (subclass of GraphicObject).

This gives the node the maximal preference to become the root during layout. If only one node is specified this way, the algorithm selects this node. If several nodes of the same connected component are specified this way, the layout algorithm chooses one of them as the root.

For Experts: Additional Options for Root Nodes

The layout algorithm manages a list of the root nodes that have been specified by the SetRoot method. To obtain the nodes in this list, use the method:

```
ICollection GetSpecRoots();
```

After layout, you can also retrieve the list of root nodes that were actually used by the algorithm. This list is not necessarily the same as the list of specified roots. For instance, it contains the chosen root nodes if none were specified or if too many were specified. To obtain the root nodes that were used by the algorithm, use the method:

```
ICollection GetCalcRoots()
```

This example shows how to iterate over the calculated root nodes and print the root node preferences:

```
foreach (Object node in  layout.GetCalcRoots()) {
   Console.WriteLine("Preference:" + layout.GetRootPreference(node));
}
For Each node As Object In layout.GetCalcRoots
 Console.WriteLine("Preference:" + layout.GetRootPreference(node))
Next
```

To directly manipulate the root node preference value of an individual node use the method:

```
SetRootPreference(Object node, int preference);
```

In this case, the layout uses the specified value instead of the heuristically calculated preference for the node. The normal preference value should be between `0` and `10000`. Specifying a root node explicitly corresponds to setting the preference value to `10000`. If you want to prohibit a node from becoming the root, specify a preference value of zero (`0`).

A negative preference value indicates that the layout algorithm should recalculate the root node preference using the heuristic. If a root was specified by the SetRoot method but this node should no longer be the root in subsequent layouts, use the following call to clear the root node setting:

```
layout.SetRootPreference(node, -1);
```

This call also removes the node from the list of specified roots.

Position Parameters

To set the position of the top left corner of the layout to (10, 10) use the method:

```
layout.SetPosition(new Point2D(10, 10), false);
```

If the graph consists of only a single tree, it is often more convenient to set the position of the root node instead. To do this, use the same method and pass true instead of false:

```
layout.SetPosition(point, true);
```

If no position is specified, the layout keeps the root node at its previous position.

Using Compass Directions for Positional Layout Parameters

To simplify the explanations of the layout parameters, we use the compass directions north, south, east, and west. The center of the root node of a tree is considered the north pole.

In the nonradial layout modes, the link flow direction always corresponds to south. If the root node is placed at the top of the drawing, north is at the top, south at the bottom, east to the right, and west to the left. If the root node is placed at the left border of the drawing, north is to the left, south to the right, east at the top, and west at the bottom.

In the radial layout modes, the root node is placed in the center of the drawing. The meaning of north and south depends on the position relative to the root: the north side of the node is the side closer to the root and the south side is the side further away from the root. The east direction is counterclockwise around the root and the west direction is clockwise around the root. This is similar to a cartographic map of a real globe that shows the area of the north pole as if you were looking down at the top of the globe.

Compass directions are used to provide uniform naming conventions for certain layout options. They occur in the alignment options, the level alignment option, and the east-west neighboring feature, which are explained later.

Layout Modes

The tree layout algorithm has several layout modes. To specify the layout mode use the LayoutMode property.

The available layout modes are defined by the TreeLayoutMode enumeration.

- **TreeLayoutMode.Free** (the default)
- **TreeLayoutMode.Level**

- **TreeLayoutMode.Radial**
- **TreeLayoutMode.AlternatingRadial**
- **TreeLayoutMode.TipOver**
- **TreeLayoutMode.TipRootsOver**
- **TreeLayoutMode.TipLeavesOver**
- **TreeLayoutMode.TipRootsAndLeavesOver**

The following sections describe the characteristics and the layout parameters of each layout mode.

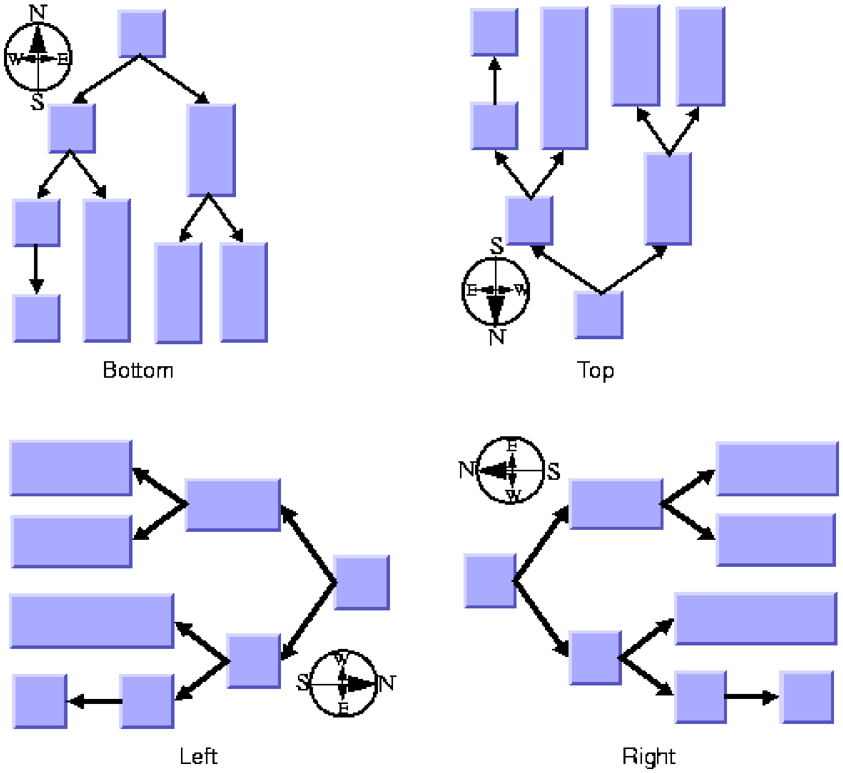
Free Layout Mode

The free layout mode arranges the children of each node starting recursively from the root so that the links flow roughly in the same direction. For instance, if the link flow direction is top-down, the root node is placed at the top of the drawing. Siblings (nodes that have the same parent) are justified at their top borders, but nodes of different tree branches (nodes with different parents) are not justified. To set the free layout mode use

```
layout.LayoutMode = TreeLayoutMode.Free;
```

Flow Direction

The flow direction parameter specifies the direction of the tree links. The compass icons show the compass directions in these layouts.



Bottom Top

Left Right

If the flow direction is to the bottom, the root node is placed topmost. Each parent node is placed above its children, which are normally arranged horizontally. (This tip-over alignment is an exception.)

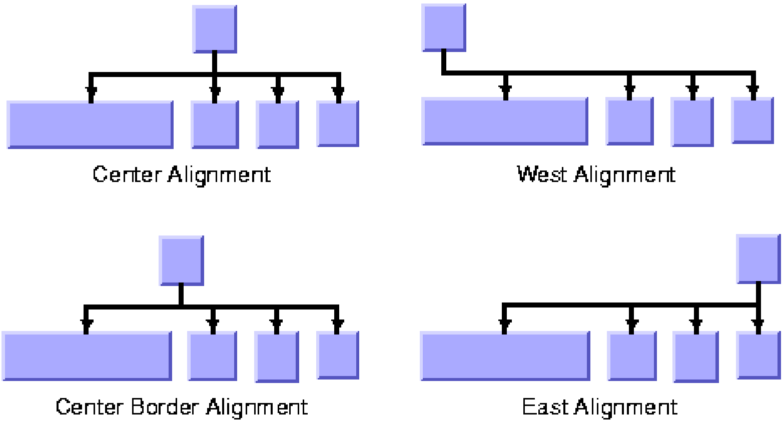
If the flow direction is to the right, the root node is placed leftmost. Each parent node is placed to the left of its children, which are normally arranged vertically.

To specify the flow direction use the FlowDirection property. The valid values for the flow direction are defined by the LayoutFlowDirection enumeration:

- **LayoutFlowDirection.Right** (the default)
- **LayoutFlowDirection.Left**
- **LayoutFlowDirection.Bottom**
- **LayoutFlowDirection.Top**

Alignment Parameter

The alignment option controls how a parent is placed relative to its children. The alignment can be set globally, in which case all nodes are aligned in the same way, or locally on each node, with the result that different alignments occur in the same drawing.



Center Alignment

West Alignment

Center Border Alignment

East Alignment

Global Alignment Parameters

To set the global alignment use the Alignment property. The valid values for the alignment are:

- **TreeLayoutAlignment.Center** (the default)

  The parent is centered over its children, taking the center of the children into account.

- **TreeLayoutAlignment.BorderCenter**

  The parent is centered over its children, taking the border of the children into account. If the size of the first and the last child varies, the border center alignment places the parent closer to the larger child than to the default center alignment.

- **TreeLayoutAlignment.East**

  The parent is aligned with the border of its easternmost child. For instance, if the flow direction is to the bottom, east is the direction to the right. If the flow direction is to the top, east is the direction to the left. See *Using Compass Directions for Positional Layout Parameters* for details.

- **TreeLayoutAlignment.West**

  The parent is aligned with the border of its westernmost child. For instance, if the flow direction is to the bottom, west is the direction to the left. If the flow direction is to the right, west is the direction to the bottom. See *Using Compass Directions for Positional Layout Parameters* for details.

- **TreeLayoutAlignment.TipOver**

  The children are arranged sequentially instead of in parallel, and the parent node is placed with an offset to the children. For details see *Tip-over Alignment*.

- **TreeLayoutAlignment.TipOverBothSides**

  The children are arranged sequentially instead of in parallel. Whereas the alignment **TipOver** arranges all children at the same side of the parent, this alignment arranges the children at both sides of the parent. For details see *Tip-over Alignment*.

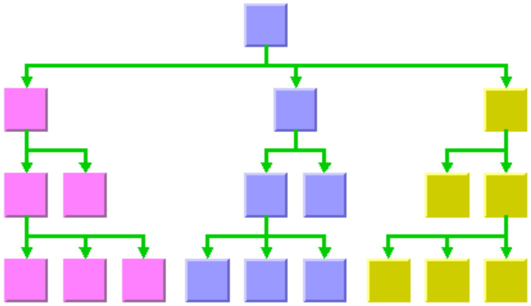
- **TreeLayoutAlignment.Mixed**

  Each parent node can have a different alignment. The alignment of each individual node can be set with the result that different alignments can occur in the same graph.

Alignment of Individual Nodes

All nodes have the same alignment unless the global alignment is set to **TreeLayoutAlignment.Mixed**. Only when the global alignment is set to **TreeLayoutAlignment.Mixed** on the TreeLayout, can each node have an individual alignment style.

The following illustration shows different alignments mixed in the same drawing.



To specify the alignment of an individual node use the methods:

```
void SetAlignment(Object node, TreeLayoutAlignment alignment)
TreeLayoutAlignment GetAlignment(Object node)
```

The valid values for alignment are:

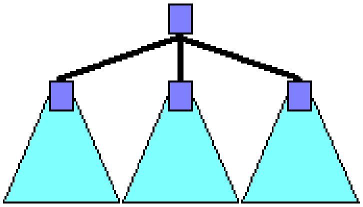
- **TreeLayoutAlignment.Center** (the default)
- **TreeLayoutAlignment.BorderCenter**
- **TreeLayoutAlignment.East**
- **TreeLayoutAlignment.West**
- **TreeLayoutAlignment.TipOver**

- **TreeLayoutAlignment.TipOverBothSides**

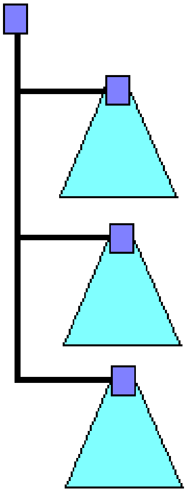
Tip-over Alignment

Normally, the children of a node are placed in a parallel arrangement with siblings directly neighbored to each other. Tip-over alignment means a sequential arrangement of the children instead.

The following illustration shows normal and tip-over alignment.



Normal Alignment



Tip-Over Alignment

Tip-over alignment is useful when the tree has many leaves. With normal alignment, a tree with many leaves would result in the layout being very wide. If the global alignment style is set to tip-over, the drawing is very tall rather than wide. To balance the width and height of the drawing, you can set the global alignment to mixed, for example:

```
layout.Alignment = TreeLayoutAlignment.Mixed;
```

Also, you can set the individual alignment to tip-over for some parents with a high number of children as follows:

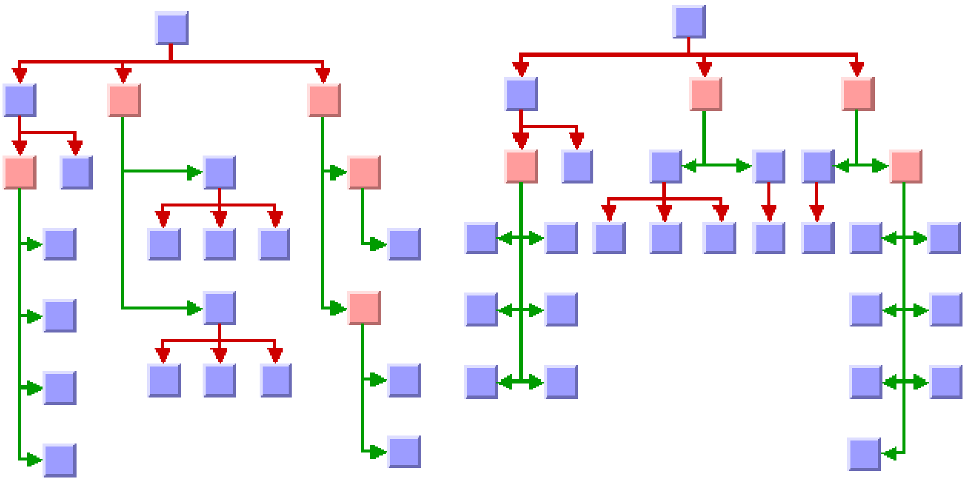
```
layout.SetAlignment(parent, TreeLayoutAlignment.TipOver);
```

Tip-over alignment can be specified explicitly for some or all of the nodes. Furthermore, the Tree Layout offers layout modes that automatically determine when to tip over, yielding a drawing that fits into a given aspect ratio. These layout modes are described in *Tip-Over Layout Modes*.

Besides the normal tip-over alignment, there is also a variant that distributes the subtrees on both sides of the center line that starts at the parent. You can specify this variant at a parent node with a high number of children by the following code:

```
layout.SetAlignment(parent, TreeLayoutAlignment.TipOverBothSides);
```

The difference between normal tip-over alignment and tip-over at both sides is shown in the illustration below. Tip-over alignment works very well with the orthogonal link style (see *Link Style*).



Link Style

The links can be straight or have a specific shape with intermediate points. You can specify that the links be reshaped into an orthogonal form. You can set the link style globally, in which case all links have the same kind of shape, or locally on each link, in which case different link shapes occur in the same drawing.

Global Link Style

To specify the global link style use the LinkStyle property:

```
TreeLayoutLinkStyle LinkStyle
```

The valid values for style are defined by the TreeLayoutLinkStyle enumeration:

- **TreeLayoutLinkStyle.NoReshape**

  None of the links is reshaped in any manner.

- **TreeLayoutLinkStyle.StraightLine**

  All the intermediate points of the links (if any) are removed. This is the default value.

- **TreeLayoutLinkStyle.Orthogonal**

The links are reshaped in an orthogonal form (alternating horizontal and vertical segments). See *Tip-over Alignment* as an example.

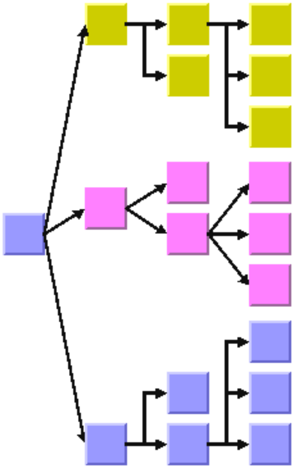
- **TreeLayoutLinkStyle.Mixed**

  Each link can have a different link style. The style of each individual link can be set to have different link shapes occurring on the same graph.

Individual Link Style

All links have the same style of shape unless the **LinkStyle** property to **TreeLayoutLinkStyle.Mixed**. Only when the link style is set to **TreeLayoutLinkStyle.Mixed** can each link have an individual link style.

The following illustration shows different link styles in the same drawing:



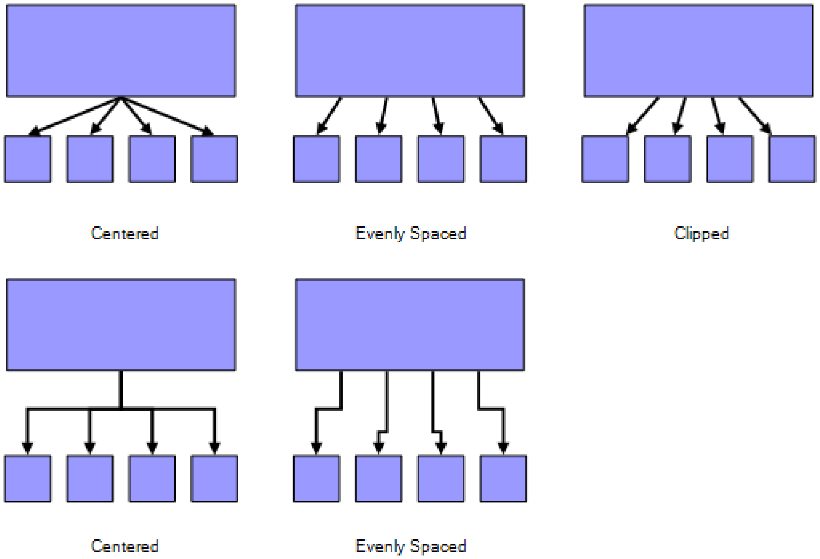
To specify the style of an individual link use the methods:

```
void SetLinkStyle(Object link, TreeLayoutLinkStyle style)
TreeLayoutLinkStyle GetLinkStyle(Object link)
```

Connector Style

The layout algorithm automatically positions the connection points of links at the nodes. The connector style parameter specifies how these connection points are calculated for the outgoing links at the parent node.

The following illustration shows various connector styles:



To specify the connector style use the ConnectorStyle property.

The valid values for style are defined by the TreeLayoutConnectorStyle enumeration.

- **TreeLayoutConnectorStyle.Centered**

  The connection points of the links are placed in the center of the border where the links are attached.

- **TreeLayoutConnectorStyle.Clipped**

  Each link pointing to the center of the node is clipped at the node border. The connection points are placed at the points on the border where the links are clipped. This style affects straight links. It behaves like centered connector pins for orthogonal links.

- **TreeLayoutConnectorStyle.EvenlySpaced**

  The connection points are evenly distributed along the node border. This style works for straight and orthogonal links.

- **TreeLayoutConnectorStyle.Automatic**

  The connector style is selected automatically depending on the link style and the layout mode. In the nonradial modes, the algorithm always chooses centered. In the radial layout modes, it chooses clipped.

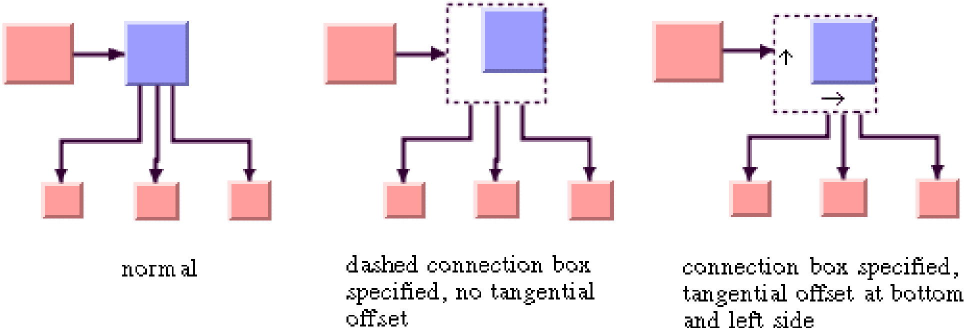
Using a Link Connection Box Interface

By default, the connector style determines how the connection points of the links are distributed on the border of the bounding box of the nodes, symmetrically with respect to the middle of each side. Sometimes it may be necessary to place the connection points on a rectangle smaller or larger than the bounding box, eventually in a nonsymmetric way. For instance, this can happen when labels are displayed below or above nodes.

You can modify the position of the connection points of the links by providing a class that implements the ILinkConnectionBoxProvider. An example for the implementation of a link connection box provider is in *Link Connection Box*. To set a link connection box provider use the LinkConnectionBoxProvider property.

The link connection box provider provides each node with a link connection box and a tangential shift offset that defines how much the connection points are "shifted" tangentially depending on which side the links connect.

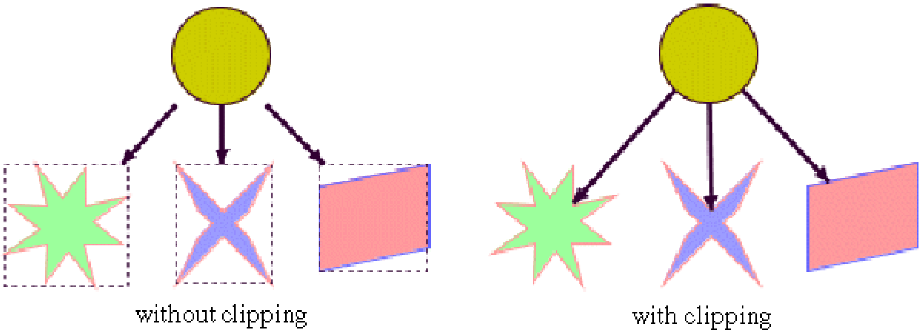
The following illustration shows the effects of customizing the connection box when the connector style is evenly spaced.


normal
dashed connection box specified, no tangential offset
connection box specified, tangential offset at bottom and left side

On the left is the result without any connection box provider. The middle picture shows the effect if the connection box provider returns the dashed rectangle for the blue node but the tangential offset at all sides of the node is 0. Note that the outgoing links are spaced according to the dashed rectangle, which appears too wide for the blue node in this situation. The picture on the right shows the effect of the connection box provider if, in addition, a positive tangential offset was specified for the bottom side and a negative offset was specified for the left side of the blue node.

Using the Link Clipping

By default, the Tree Layout places the connection points of links at the border of the bounding box of the nodes. If the node has a nonrectangular shape such as a triangle, rhombus, or circle, you may want the connection points to be placed exactly on the border of the shape. This can be achieved by using the link clipping feature. The link clipping corrects the calculated connection point so that it lies on the border of the shape. An example is shown in the following illustration:

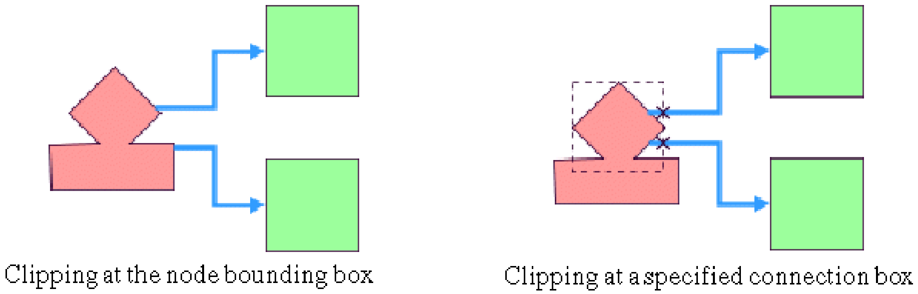


To enable the link clipping, use the property LinkClipping. The default value is **true**.

> ***Note:*** *Additionally to the link clip provider, the ShapeAnchor can be used. This special link anchor updates the clipped connection points automatically during interactive node movements.*

The connector style, the link connection box provider, and the link clipping work together in the following way: by respecting the connector style, the proposed connection points are calculated on the rectangle obtained from the link connection box provider (or on the bounding box of the node, if no link connection box provider was specified). Then, the proposed connection point is corrected by the link clipping and the clipped points are used to connect the link to the node.

An example of the combined effect is shown in the following illustration.


Clipping at the node bounding box
Clipping at a specified connection box

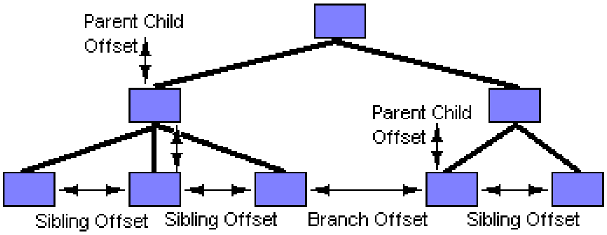
If the links are clipped at the red node (left picture), they appear unsymmetrical with respect to the node shape, because the relevant part of the node (here: the upper rhombus) is not in the center of the bounding box of the node, but the proposed connection points are calculated with respect to the bounding box. This can be corrected by using a link connection box provider to explicitly specify a smaller connection box for the relevant part of the node (right picture) such that the proposed connection points are placed symmetrically at the upper rhombus of the node.

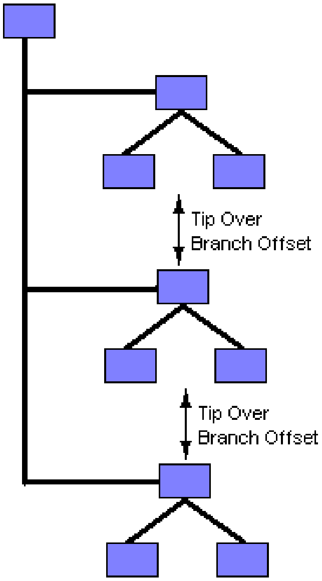
Spacing Parameters

The spacing of the layout is controlled mainly by three spacing parameters: the distance between a parent and its children, the minimal distance between siblings, and the minimal distance between nodes of different branches. For instance, if the flow direction is to the top or bottom, the offset between parent and children is vertical, while the sibling offset and the branch offset are horizontal.

For tip-over alignment, an additional spacing parameter is provided: the minimal distance between branches starting at a node with tip-over alignment. This offset is always orthogonal to the normal branch offset. If the flow direction is to the top or bottom, the tip-over branch offset is vertical.

The following illustration shows the various spacing parameters:





To specify the spacing parameters use the following properties:

- float ParentChildOffset
- float SiblingOffset
- float BranchOffset
- float TipOverBranchOffset

For Experts: Additional Spacing Parameters

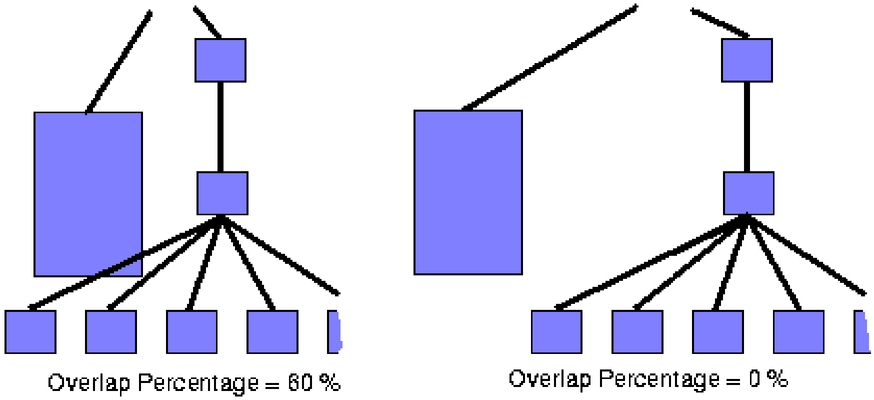
The spacing parameters normally specify the minimal offsets between the node borders. Hence, the layout algorithm places the nodes such that they do not overlap. You can also specify that the layout should ignore the node sizes. To do so, set the RespectNodeSizes property to **false.**

In this case, the spacing parameters are interpreted as the minimum distances between the node centers, and the node sides are not taken into account during the layout. However, if the specified offset parameters are now smaller than the node size, the nodes and links will overlap. This often happens with orthogonal links in particular. It makes sense to use this option only if all nodes have approximately the same size, all links are straight, and the spacing parameters are larger than the largest node.

If the link style is orthogonal, the shape of the links from the parent to its children looks like a fork. The position of the bend points in this shape can be influenced by the orthogonal fork percentage, a value between 0 and 100. This is a percentage of the parent child offset. If the orthogonal fork percentage is 0, the link shape forks directly at the parent node. If the percentage is 100, the link shape forks at the child node. A good choice is between 25 and 75. This percentage can be specified with the OrthForkPercentage property.

If the link style is not orthogonal, links may overlap neighboring nodes. This happens only in a very few cases, for instance, when a link starts at a very small node that is neighbored by a very large node. This deficiency can be fixed by increasing the branch offset. However, this influences the layout globally, affecting nodes without that deficiency. To avoid a global change, you can change the overlap percentage instead, which is a value between 0 and 100. This value is used by an internal heuristic of the layout algorithm that considers a node to be smaller by this percentage. The default percentage is 30. This usually results in better usage of the space. However, if very small nodes are neighbored to very large nodes, it is recommended to decrease the overlap percentage or to set it to 0 to switch this heuristic off to avoid links overlapping nodes. The overlap percentage can be set through the OverlapPercentage property.

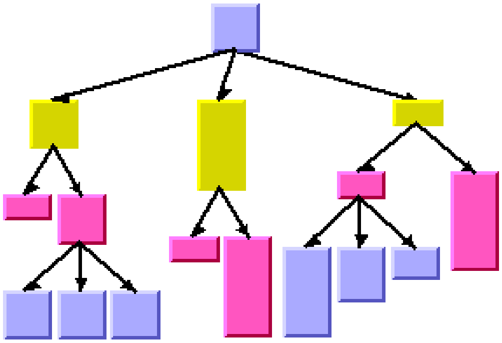
Note: *It is recommended that you always set the orthogonal fork percentage to a value larger than the value of the overlap percentage.*



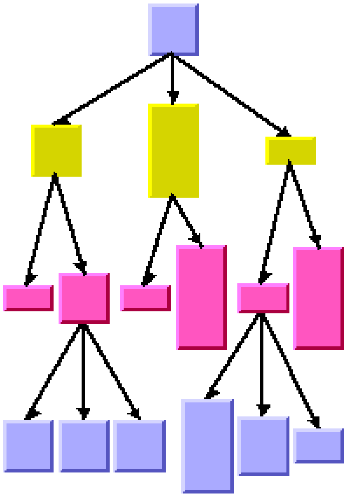
Level Layout Mode

The level layout mode partitions the node into levels and arranges the levels horizontally or vertically. The root is placed at level 0, its children at level 1, the children of those children at level 2, and so on. In contrast to the free layout mode, in level layout mode the nodes of the same level are justified with each other even if they are not siblings (that is, they do not have the same parent). To set the level layout mode set the LayoutMode property to **TreeLayoutMode.Level**.

The following illustration shows the same graph in free layout mode and in level layout mode:



Free Layout Mode



Level Layout Mode

General Parameters

Most layout parameters that work for the free layout mode work as well for the level layout mode. You can set the flow direction, the spacing offsets, the global or individual link style, and the global or individual alignment. See *Free Layout Mode* for details.

The differences from the free layout mode are:

- The tip-over alignment does not work in level layout mode.

  The parent-child offset parameter controls the spacing between the levels. In level layout mode, it is the minimal distance between parent and its children, while in free layout mode, it is the exact distance between parent and its children.

- The overlap percentage has no effect in level layout mode.

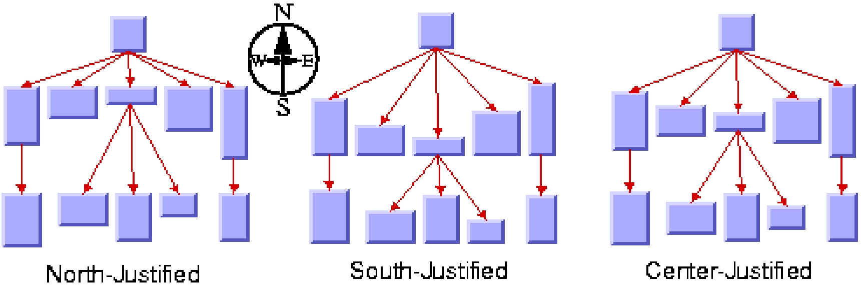
Level Alignment

In level layout mode with flow direction to the top or bottom, the nodes are organized in horizontal levels such that the nodes of the same level are placed approximately at the same y-coordinate. The nodes can be justified, depending on whether the top border, the bottom border, or the center of all nodes of the same level should have the same y-coordinate.

In flow direction to the left or right, the nodes are organized in vertical levels approximately at the same x-coordinate. The nodes of the same level can be justified at the left border, at the right border, or at the center.

To distinguish the level alignment independently from the flow direction, we use the directions north and south (see *Using Compass Directions for Positional Layout*

Parameters). The north border of a node is the border that is closer to the level where its parent is placed, and the south border of a node is the border that is closer to the level where its children are placed. If the flow direction is to the bottom, the level alignment north means that the nodes are justified at the top border, and south means that the nodes are justified at the bottom border. If the flow direction is to the top, it is converse: north means the bottom border and south means the top border. If the flow direction is to the right, then north means the left border and south means the right border.



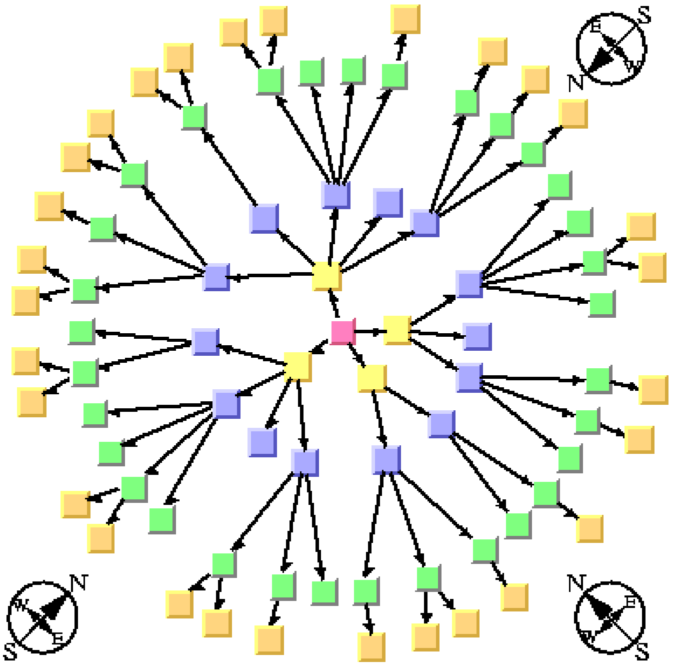
To specify the level alignment use the LevelAlignment property.

The valid values for alignment are defined by the TreeLayoutLevelAlignment enumeration:

- **TreeLayoutLevelAlignment.Center** (the default)
- **TreeLayoutLevelAlignment.North**
- **TreeLayoutLevelAlignment.South**

Radial Layout Mode

The radial layout mode partitions the node into levels and arranges the levels in circles around the root node. The following illustration shows an example of the radial layout mode. The compass icons show the compass directions in this drawing.



To set the radial layout mode set the LayoutMode property to **TreeLayoutMode.Radial**.

General Parameters

Most layout parameters that work for the free and level layout mode work as well for the radial layout mode. You can set the spacing offsets, the level alignment, the global or individual link style, and the global or individual alignment. See *Free Layout Mode* and *Level Layout Mode* for details.

The radial layout mode differs from the other layout modes as follows:

- The tip-over alignment does not work in radial layout mode.
- The orthogonal link style does not work in radial layout mode.
- The clipped connector style is always used.
- The parent-child offset parameter controls the minimal distance between the circular levels. However, it is sometimes necessary to increase the offset between circular levels to obtain enough space on the circle to place all nodes of a level.
- The level alignment north indicates alignment at the inner border of the circular level (that is, towards the root), and the level alignment south indicates alignment at the outer border of the circular level (that is, away from the root).

- The level alignments north and south sometimes result in overlapping nodes.
- The overlap percentage has no effect in radial layout mode.

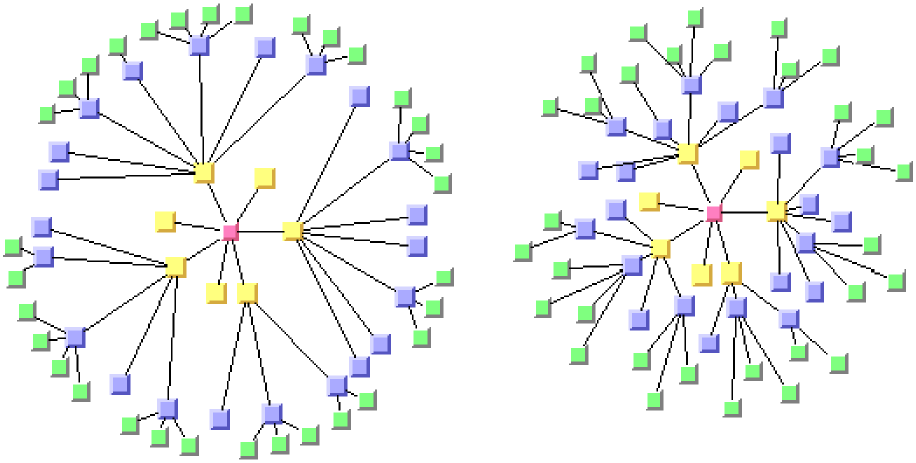
Alternating Radial Mode

If levels of the graph contain many nodes, it is sometimes necessary to increase the radius of the circular level to provide enough space on the circumference of the circle for all the nodes. This may result in a considerable distance from the previous level. To avoid this, there is an alternating radial mode. The alternating radial mode places the nodes of a level alternating between two circles instead of one circle, resulting better usage of the space of the layout.

The alternating radial mode uses two circles only when necessary. For many small and light trees, there will be no difference from the normal radial mode. Only for large graphs with a large number of children will the alternating radial mode have an effect.

To set the alternating radial layout mode set the **LayoutMode** property to **TreeLayoutMode.AlternatingRadial**.

The following illustration shows the radial (left) and alternating radial (right) mode on the same graph



Aspect Ratio

If the drawing area is not a square, arranging the levels as circles is not always the best choice. You can specify the aspect ratio of the drawing area to better fit the layout to the drawing area. In this case, the algorithm uses ellipses instead of circles.

When specifying the aspect ratio, there are several possibilities. If the drawing area is a view (a subclass of IDiagramView), you can use the method:

```
void SetAspectRatio(IDiagramView view);
```

If the drawing area is given only as a rectangle, use the following:

```
void SetAspectRatio(Rectangle2D rect);
```

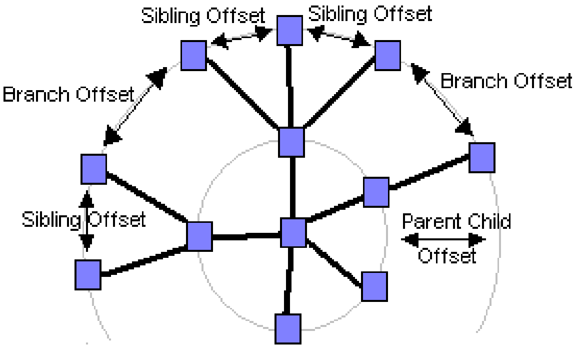
If neither a view nor a rectangle is specified, you can calculate the aspect ratio from the width and height of the drawing area as aspectRatio = width/height and use the AspectRatio property:

Spacing Parameters

The spacing parameters of the radial layout mode are controlled by the same properties as used for the free and level layout modes: ParentChildOffset, SiblingOffset and BranchOffset.

> ***Note:*** *The sibling and branch offsets are minimal distances tangential to the circles or ellipses, while the parent-child offset is a minimal distance radial to the circles or ellipses.*

The following illustration shows the spacing parameters in radial layout mode.

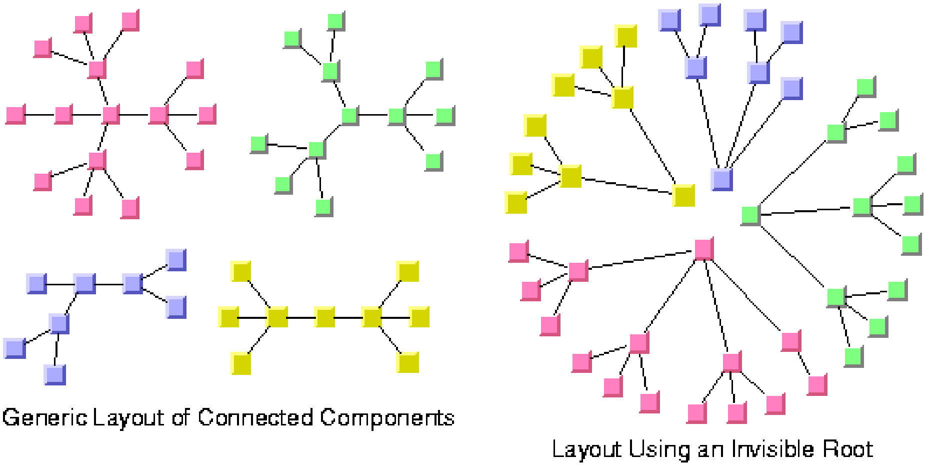


For Experts: Adding an Invisible Root to the Layout

If the graph contains several trees that are disconnected from each other, the layout places them individually next to each other. Each connected component has its own radial structure with circular layers. However, sometimes it is appropriate to fit all connected components into a single circular layer structure. Conceptually, this is done by adding an invisible root at the center and connecting all disconnected trees to this root. This works only if the generic mechanism to lay out connected components is switched off. To add an invisible root to the layout use the following properties:

```
layout.LayoutOfConnectedComponentsEnabled=false;
layout.InvisibleRootUsed=true;
```

The following illustration shows the result of using an invisible root:


Generic Layout of Connected Components

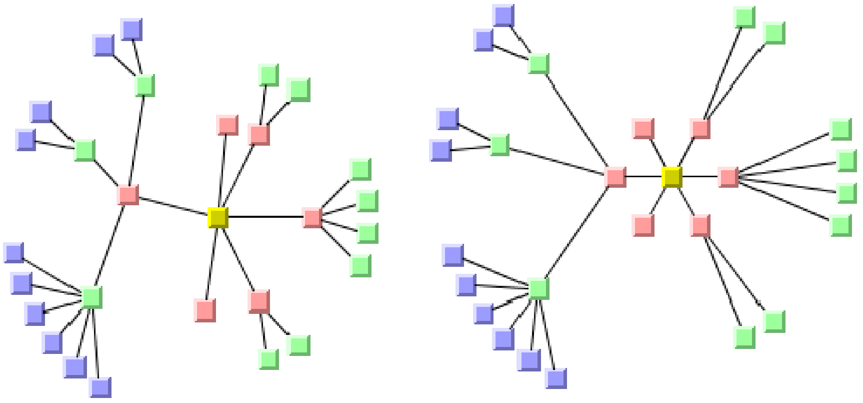
Layout Using an Invisible Root

For Experts: Even Spacing for the First Circle

The radial mode is designed to optimize the space such that the circles have a small radius and the overall space for the entire layout is small. To achieve this, the layout algorithm may create larger gaps on the inner circles for better space usage of the outer circles. This may produce unevenly spaced circles, most notably for the first circle where all nodes have the same parent node.

To avoid this effect, you can force the nodes to be evenly spaced on the entire first circle. Depending on the structure of the graph, this may cause the overall layout to waste more space on the other circles but may produce a more pleasing graph. To enable even spacing set the FirstCircleEvenlySpacing property to **true**.

The following illustration shows the same graph with evenly and non-evenly spaced first circle.



Unevenly spaced first (red) circle

Evenly spaced first (red) circle

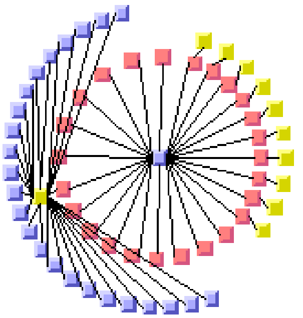
For Experts: Forcing All Levels To Alternating

When the layout mode **TreeLayoutMode.AlternatingRadial** is used, the layout checks whether the alternating node arrangement of a level saves space. If it doesn't save space, it uses the normal radial arrangement. Hence, for many sparse graphs, radial and alternating radial mode yield the same result because the alternating arrangement does not save space for any level. It is possible to disable the space check, that is, to perform an alternating arrangement for all levels even if this results in waste of space. To do so, set the AllLevelsAlternating property to **true**.

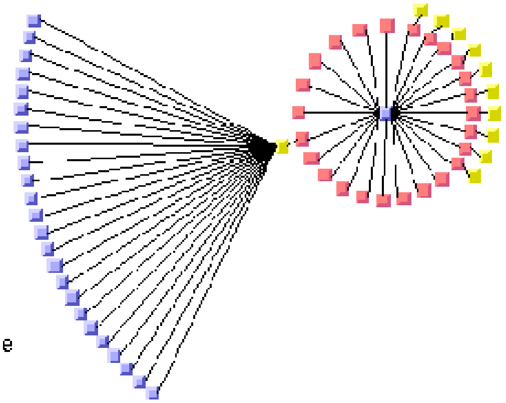
For Experts: Setting a Maximal Children Angle

If a node has a lot of children, they may extend over a major portion of the circle and, therefore, are placed nearly 360 degrees around the node. This can result in links overlapping some nodes. The deficiency can be fixed by increasing the offset between parent and children. However, this affects the layout globally which means that nodes without the deficiency are also affected. To avoid a global change such as this, you can limit the maximal angle between the two rays from the parent (if it is not the root) to its two outermost children. This increases the offset between parent and children only where necessary.

In the following illustration, you can see in the layout on the left that many of the links overlap other nodes. In the layout on the right, you can see how this problem was solved by setting a maximal children angle between two rays from a parent to the two outermost children.



Unrestricted Maximal Children Angle



Maximal Children Angle = 90 degrees

To set an angle in degrees use the MaxChildrenAngle property.

Recommended values are between 30 and 180. Setting the value to 0 means the angle is unrestricted. The calculation of the angle is not very precise above 180 degrees or if the aspect ratio is not 1.0.

Tip-Over Layout Modes

As in radial layout mode, drawing in free layout mode can be adjusted to the aspect ratio of the drawing area. Free layout mode can also use tip-over alignment to balance the drawing between the height and depth.

While tip-over alignment can be specified explicitly for individual nodes, the Tree Layout algorithm also has layout modes that automatically use tip-over alignment when needed. These are the tip-over layout modes.

The tip-over layout modes work as follows: Several trial layouts are performed in free layout mode. For each trial, certain tip-over alignments are set for individual nodes, while the specified alignment of all other nodes is preserved. The algorithm picks the trial layout that best fits the specified aspect ratio of the drawing area.

The aspect ratio can be specified by one of the methods or property (see *Aspect Ratio* in the Radial Layout Mode):

```
void SetAspectRatio(IDiagramView view)
void SetAspectRatio(Rectangle2D rect)
float AspectRatio
```

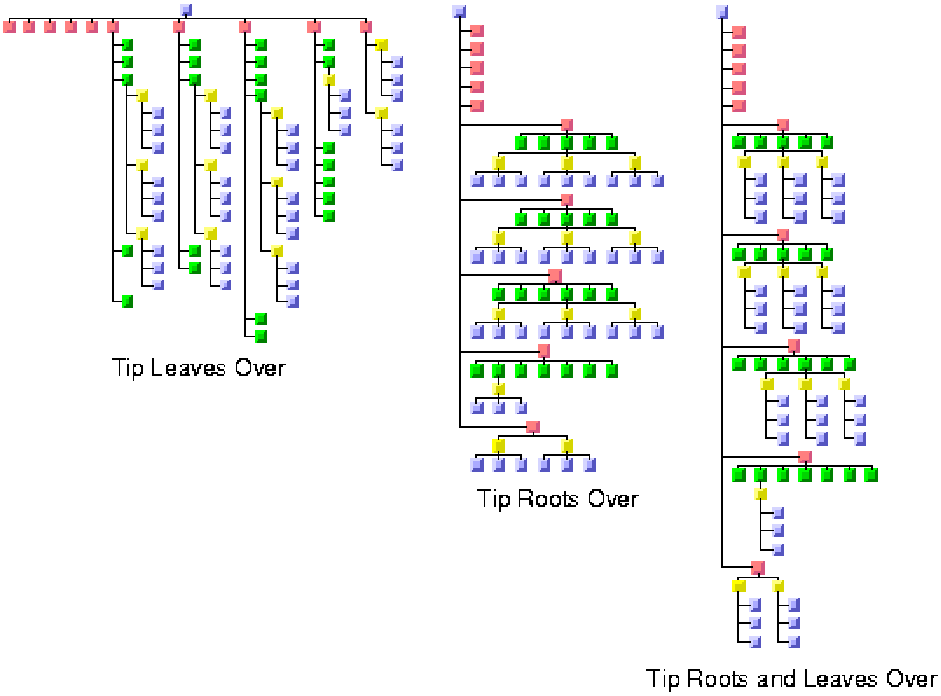
The tip-over modes are slightly more time-consuming than the other layout modes. For very large trees, it is recommended that you set the allowed layout time to a high value (for instance, 60 seconds) when using the tip-over modes. To set this mode:

```
layout.AllowedTime=60000;
```

By using this call, you avoid running short of time for sufficient iterations of the layout algorithm. Because it would be too time-consuming to check all possibilities of tip-over alignment use, there are heuristics that check only certain trials according to the following different strategies, illustrated in Tip-Over Strategies:

- *Tip Leaves Over*
- *Tip Roots Over*
- *Tip Roots and Leaves Over*
- *Tip Over Fast*

The following illustration shows the result of various tip-over strategies:



Tip Leaves Over

Tip Roots Over

Tip Roots and Leaves Over

Tip Leaves Over

To use this tip-over strategy, set the layout mode as follows:

```
layout.LayoutMode = TreeLayoutMode.TipLeavesOver;
```

The heuristic first tries the layout without any additional tip-over options. Then it tries to tip over the leaves, then the leaves and their parents, then additionally the parents of these parents, and so on. As a result, the nodes closest to the root use normal alignment and the nodes closest to the leaves use tip-over alignment.

Tip Roots Over

To use this tip-over strategy, set the layout mode as follows:

```
layout.LayoutMode = TreeLayoutMode.TipRootsOver;
```

The heuristic first tries the layout without any additional tip-over options. Then it tries to tip over the root node, then the root and its children, then additionally the children of these children, and so on. As a result, the nodes closer to the leaves use normal alignment and the nodes closer to the root use tip-over alignment.

Tip Roots and Leaves Over

To use this tip-over strategy, set the layout mode as follows:

```
layout.LayoutMode = TreeLayoutMode.TipRootsAndLeavesOver;
```

The heuristic first tries the layout without any additional tip-over options. Then it tries to tip over the root node and the leaves simultaneously; then the root and its children, and the leaves and its parent; then additionally the children of these children and the parents of these parents, and so on. As result, the nodes in the middle of the tree use normal alignment and the nodes closest to the root or leaves use the tip-over alignment.

This is the slowest strategy because it includes all trials of the strategy "tip leaves over" as well as all tries of the strategy "tip roots over."

Tip Over Fast

The fast tip-over provides a compromise among all other strategies. The heuristic tries a small selection of the other strategies, not all possibilities. Therefore, it is the fastest strategy for large graphs.

To use this strategy, set the layout mode as follows:

```
layout.LayoutMode = TreeLayoutMode.TipOver;
```

It is possible that all four strategies yield the same result because the strategies are not disjoint; that is, certain trials are performed in all four strategies. In addition, the tip-over modes do not necessarily produce the optimal layout that gives the best possible fit to the aspect ratio. The reason is that some unusual configurations of tip-over alignment are never tried because doing so would cause the running time to be too high.

For Experts: Additional Tips and Tricks

In this section, you are going to find the following topics:

- *Specifying East-West Neighbors*
- *Retrieving Link Categories*

- *Sequences of Layouts with Incremental Changes*
- *Interactive Editing*
- *Specifying the Order of Children*

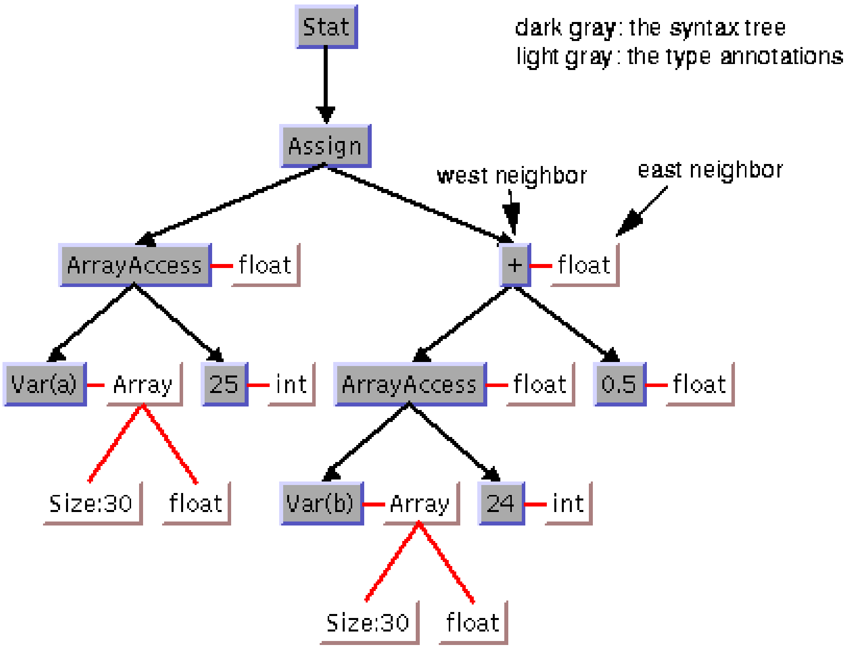
Specifying East-West Neighbors

You can specify that two unrelated nodes must be directly neighbored in a direction perpendicular to the flow direction. In the level and radial layout modes, both nodes are placed in the same level next to each other. In the free layout and tip-over modes, both nodes are placed aligned at the north border. Such nodes are called east-west neighbors because one node is placed as the direct neighbor on the east side of the other node. The other node becomes the direct neighbor on the west side of the first node. (See also *Using Compass Directions for Positional Layout Parameters*).

Technically, both nodes are treated as parent and child, even if there may be no link in between. Therefore, one of the two nodes can have a real parent, but the other node should not because its virtual parent is its east-west neighbor.

The east-west neighbor feature can be used, for example, for annotating nodes in a typed syntax tree occurring in compiler construction.

The following illustration shows an example of such a tree.



To specify that two nodes are east-west neighbors, use the method:

```
void SetEastWestNeighboring(object eastNode, object westNode);
```

You can also use the following method, which is identical except for the reversed parameter order:

```
void SetWestEastNeighboring(Object westNode, Object eastNode);
```

If the flow direction is to the bottom, the latter method may be easier to remember because, in this case, west is to the left of east in the layout, which is similar to the text flow of the parameters.

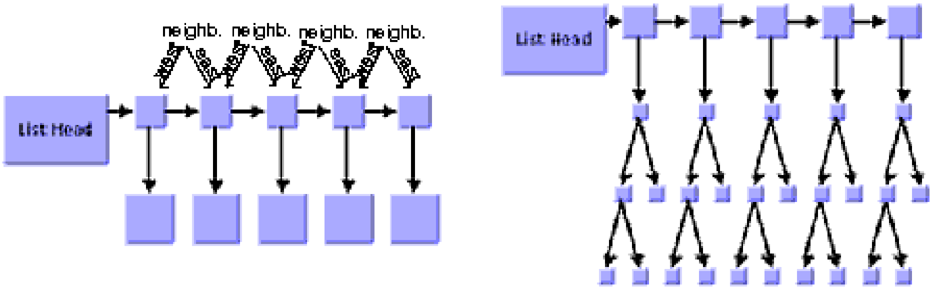
To obtain the node that is the east or west neighbor of a node, use the calls:

```
object GetEastNeighbor(object node);
object GetWestNeighbor(object node);
```

Note that each node can have maximally one east neighbor and one west neighbor because they are directly neighbored. If more than one direct neighbor is specified, it is partially ignored. Cyclic specifications can cause conflict as well. For instance, if node B is the east neighbor of node A and node C is the east neighbor of B, then node A cannot be the east neighbor of C. (Strictly speaking, such cycles could be technically possible in some situations in the radial layout mode, but nonetheless they are not allowed in any layout mode.)

If B is the east neighbor of A, then A is automatically the west neighbor of B. On the other hand, the east neighbor of A can itself have another east neighbor. This allows creating chains of east-west neighbors, which is a common way to visualize lists of trees.

Two examples are shown in the following illustration.

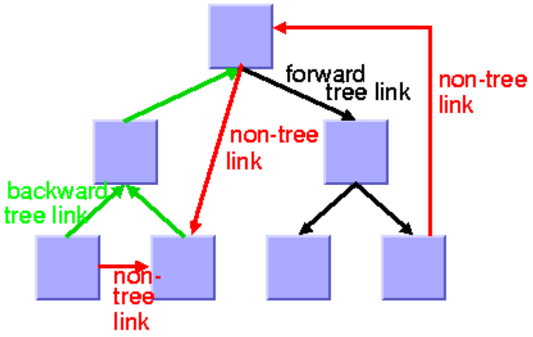


Retrieving Link Categories

The Tree Layout algorithm works on a spanning tree, as mentioned in a *Brief Description of the Algorithm*. If the graph to be laid out is not a pure tree, the algorithm ignores some links. To perform special treatment of such links, you can obtain a list of non-tree links.

Because there are parents and children in the spanning tree, we distinguish the following link categories:

- A forward tree link is a link from a parent to its child.
- A backward tree link is a link from a child to its parent. If the link is drawn as a directed arrow, the arrow will point in the opposite direction to the flow direction.
- A non-tree link is a link between two unrelated nodes; neither one is a child of the other. The following illustration shows link categories.



The layout algorithm uses these link categories internally but does not store them permanently for the sake of conserving time and memory efficiency. If you want to perform special treatment on some link categories (for example, to call the Link Layout on the non-tree links), you must specify before the layout that you want to access the link categories after the layout. To do this, set the CategorizingLinks property to **true**.

After the layout is performed, the link categories can be obtained by the methods:

```
ICollection GetCalcForwardTreeLinks();
ICollection GetCalcBackwardTreeLinks();
ICollection GetCalcNonTreeLinks();
```

The link category data gets filled each time the layout is called, unless you set the property **CategorizingLinks** back to **false**.

Sequences of Layouts with Incremental Changes

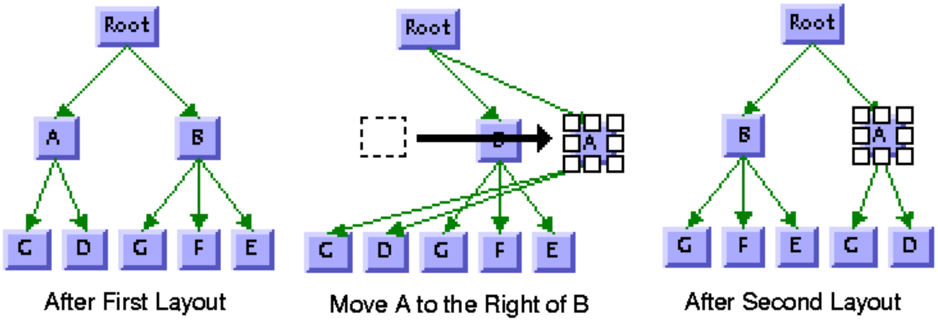
You can work with trees that have become out-of-date, for example, those that need to be extended with more children. If you perform a layout after an extension, you probably want to identify the parts that had already been laid out in the original graph. The Tree Layout algorithm supports these incremental changes in incremental mode because it takes the previous positions of the nodes into account. It preserves the relative order of the children in the subsequent layout.

In nonincremental mode, the Tree Layout algorithm calculates the order of the children from the node order given by the attached graph model. In this case, the layout is independent from the positions of the nodes before layout. It does not preserve the relative order of the children in subsequent layouts.

The incremental mode is enabled by default. To disable the incremental mode set the IncrementalMode property to **false**.

Interactive Editing

The fact that the relative order of the layout is preserved is particularly useful during interactive editing. It allows you to correct the layout easily. For instance, if the first layout places a node A left to its sibling node B but you need to reverse the order, you can simply move node A to the right of node B and start a new layout to clean up the drawing. In the second layout, A remains to the right of B, and the subtree of A will follow node A.



Specifying the Order of Children

Some applications require a specific relative order of the children in the tree. This means that, for instance, when the flow direction is to the bottom, which child must be placed to the left of another child. Even if the graph has never been laid out, you can use the coordinates to specify a certain order of the children at a node. You can use the following:

- First, make sure that the incremental mode is enabled.
- In free and level layout modes with flow direction to the bottom or top, determine the maximal width W of all nodes. Simply move the child that should be in the leftmost position to the coordinate (0, 0), and the child that should get the ith relative position (in order from left to right) to the coordinate ((W+1)*i, 0).
- If the flow direction is to the left or to the right, determine the maximal height H of all nodes. Move the child that should be in the topmost position to the coordinate (0, 0) and the child that should get the ith relative position (in the order from top to bottom) to coordinate (0, (H+1)*i).

- In the radial layout modes, determine the maximal diagonal D = W2 + H of all nodes. If the position of the parent is (x, y) before the layout, move the child that should be the first in the circular order to the coordinate (x, y+D) and the child that should get the ith relative position in the circular order to coordinate (x+D*i, y+D).
- If you want to specify a relative order for all nodes in radial layout mode, you must do this for the parents before you do it for the children. In this case, moving the children can be performed easily during a depth-first traversal from the root to the leaves.

The layout that is performed after moving the children arranges the children with the relative order.

Force-Directed Layout

Use the following topics to learn about the Force-Directed Layout algorithm (class ForceDirectedLayout).

In This Section

Samples

Provides some sample drawings produced by the algorithm.

What Types of Graphs?

Explains the type of graph on which you can use the algorithm.

Application Domains

Explains the application domains for the algorithm.

Features

Lists the features of the algorithm.

Limitations

Explains the limitations of the algorithm.

Brief Description of the Algorithm

Provides a short description of the Force-Directed Layout algorithm.

Code Sample

Provides a sample of code that shows how to use the algorithm.

Generic Features and Parameters

Explains what generic features of the graph layout library are supported and how.

Specific Parameters

Presents the parameters of the Force-Directed Layout algorithm.

For Experts: Additional Features

Provides additional information for layout experts.

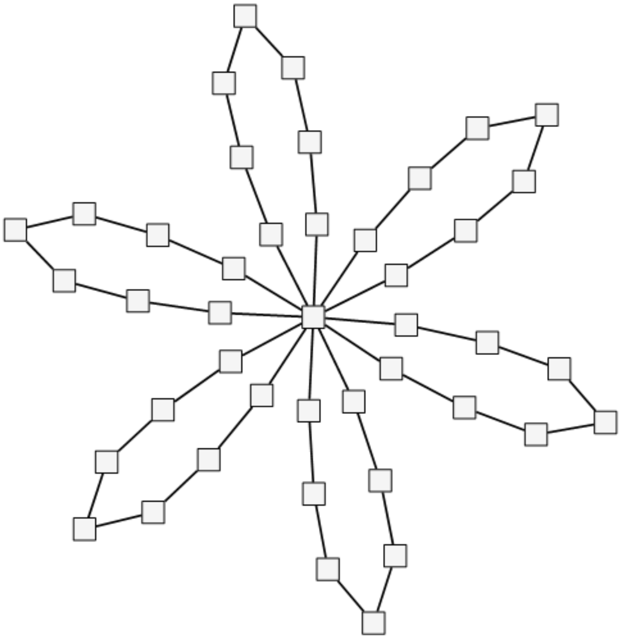
Using the Link Clipping

Explains how to use the link clipping provider.

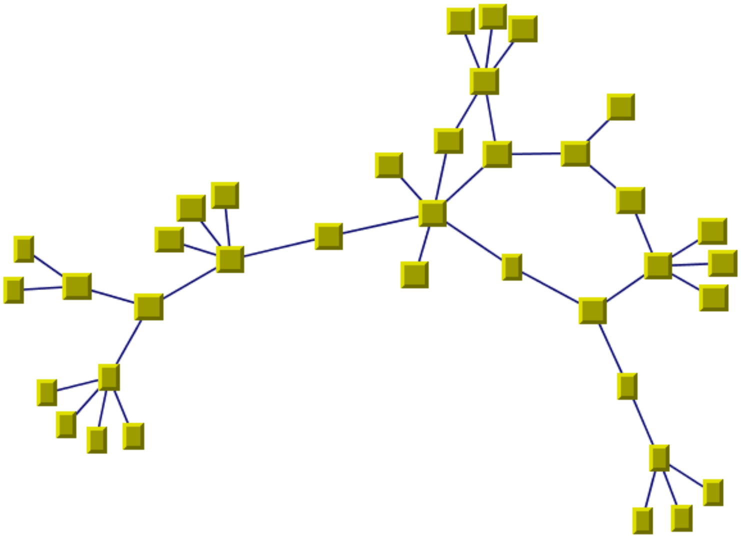
Samples

Here are sample drawings produced with the Force-Directed Layout.

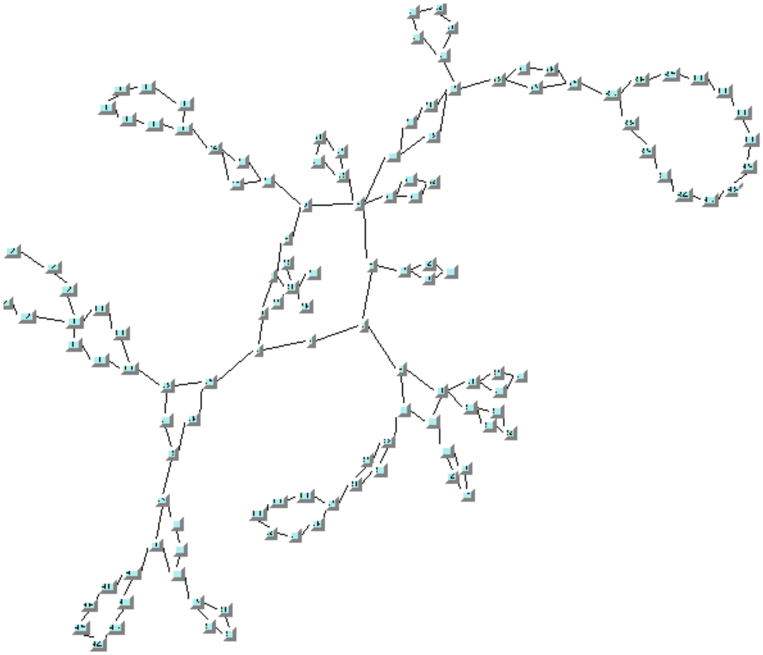
The following illustration shows small cyclic graph drawing produced with the Force-Directed Layout.



The following illustration shows medium graph drawing (combination of cycles and trees) produced with the Force-Directed Layout.



The following illustration shows large graph drawing (combination of cycles and trees) produced with the Force-Directed Layout.



What Types of Graphs?

Any type of graph:

- Connected and disconnected graphs
- Planar and nonplanar graphs

Application Domains

Application domains of the Force-Directed Layout include:

- Telecom and networking (WAN diagrams)
- Software management/software (re-)engineering (call graphs)
- CASE tools (dependency diagrams)
- Database and knowledge engineering (semantic networks, database query graphs, qualitative reasoning and other artificial intelligence diagrams, and so on)
- World Wide Web (Web hyperlink neighborhood)

Features

- Often provides a drawing without any or with only a few link crossings and with approximately equal length links for small- and medium-size graphs having a small number of cycles. The maximum number of nodes for which you can use the algorithm depends on the connectivity of the graph and is difficult to predict.
- On demand, the algorithm can take into account the size (width and height) of the nodes. Otherwise, they are more efficiently considered as points.
- It is possible to specify the length for each link individually.
- Because it takes into account the initial configuration (coordinates) of the nodes, the algorithm allows an incremental layout methodology. Sometimes the layout algorithm is not able to find the optimal layout because it has been trapped by a local minimum of the cost function. In this case, the user can perform the layout once, then move by hand one or more nodes to help the layout algorithm skip from the local minimum, and then perform the layout again. However, the success of this procedure depends on the choice of the nodes to be moved. In many cases, the choice is quite intuitive.

Limitations

- The algorithm is not appropriate for all graphs. In particular, it will produce bad results on some highly connected *cyclic graph*s for which a planar drawing with equal-length links may simply not exist.

- The computation time required to obtain an appropriate drawing grows relatively quickly with the size of the graph (that is, the number of nodes and links) and the layout process may become time-consuming for large graphs.
- Overlapping nodes cannot always be avoided. Nevertheless, the layout algorithm often produces a drawing with no overlapping nodes.

Brief Description of the Algorithm

This layout algorithm iteratively searches for a configuration of the graph where the length of the links is close to a user-defined or a default value.

Code Sample

Below is a code sample using the ForceDirectedLayout class. This code sample shows how to perform a Force-Directed Layout on a Group:

```
using System;
using ILOG.Diagrammer;
using ILOG.Diagrammer.Graphic;
using ILOG.Diagrammer.GraphLayout;


…
Group group = new Group();

// Fill the group with nodes and links here

ForceDirectedLayout layout = new ForceDirectedLayout ();

group.GraphLayout = layout;
group.PerformGraphLayout();
Imports System
Imports ILOG.Diagrammer
Imports ILOG.Diagrammer.Graphic
Imports ILOG.Diagrammer.GraphLayout

Dim group As Group = New Group

' Fill the group with nodes and links here

Dim layout As ForceDirectedLayout = New ForceDirectedLayout

group.GraphLayout = layout
group.PerformGraphLayout()
```

Generic Features and Parameters

The ForceDirectedLayout class supports the following generic parameters defined in the GraphLayout class (see *Layout Parameters and Features in the GraphLayout Base Class*):

- *Allowed Time*

- *Layout of Connected Components*
- *Layout Region*
- *Link Clipping*
- *Link Connection Box*
- *Preserve Fixed Links*
- *Preserve Fixed Nodes*
- *Stop Immediately*

The following comments describe the particular way in which these parameters are used by this subclass.

Allowed Time

The layout algorithm stops if the allowed time setting has elapsed. (See *Allowed Time*)

Layout of Connected Components

The layout algorithm can utilize the generic mechanism to lay out connected components. For more information about this mechanism, see *Layout of Connected Components*.

Layout Region

The layout algorithm can use the layout region setting (either your own or the default setting) to control the size and the position of the graph drawing. All three ways to specify the layout region are available for this subclass. (See *Layout Region*)

Note that by default the Force-Directed Layout algorithm does not use the layout region. (For details see also *Force Fit to Layout Region*.)

Remember that if you are using the default settings, the visible area of the diagram view (an instance of IDiagramView) attached to the container is used as a layout region. If several diagram views are attached, the first attached view is used. If no diagram view is attached, the layout region is automatically estimated on the basis of the number and size of the nodes.

Link Clipping

The layout algorithm can use a link clip provider to clip the connection points of a link. (See *Link Clipping*)

This is useful if the nodes have a nonrectangular shape such as a triangle, rhombus, or circle. If no link clip provider is used, the links are normally connected to the bounding boxes of the nodes, not to the border of the node shapes. See *Using the Link Clipping* for details of the link clipping mechanism.

Link Connection Box

The layout algorithm can use a link connection box interface (see *Link Connection Box*) in combination with the link clip interface. If no link clip interface is used, the link connection box interface has no effect. For details see *Using the Link Clipping*.

Preserve Fixed Links

The layout algorithm does not reshape the links that are specified as fixed. (See *Preserve Fixed Links* and *Link Style*.)

Preserve Fixed Nodes

The layout algorithm does not move the nodes that are specified as fixed. Moreover, the algorithm takes into account the fixed nodes when computing the position of the nonfixed nodes. (See *Preserve Fixed Nodes*.)

Stop Immediately

The layout algorithm stops after cleanup if the method StopImmediately is called. (For a description of this method in the **GraphLayout** class, see *Stop Immediately*.) If the layout stops early because the allowed time has elapsed, the result code in the layout report is GraphLayoutReportCode.StoppedAndInvalid.

Specific Parameters

The following parameters are specific to the ForceDirectedLayout class.

Link Style

When the layout algorithm moves the nodes, straight-line links will automatically follow the new positions of their end nodes. If the diagram contains other types of links (for example orthogonal links), the shape of the link may not be appropriate because the intermediate points of the link will not be moved. In this case, you can ask the layout algorithm to automatically remove all the intermediate points of the links (if any).

To do this, you use the LinkStyle property.

The valid values for style are:

- **ForceDirectedLayoutLinkStyle.NoReshape**

  None of the links is reshaped in any manner.

- **ForceDirectedLayoutLinkStyle.StraightLine**

  All the intermediate points of the links (if any) are removed. This is the default value.

Number of Iterations

The iterative computation of the layout algorithm is stopped if the time exceeds the allowed time (see *Allowed Time*) or if the number of iterations exceeds the allowed number of iterations. To specify this number, use the AllowedNumberOfIterations property.

Preferred Length

The main objective of this layout algorithm is to obtain a layout where all the links have a given length. This is called the preferred length.

To specify the preferred length for all the links, use the PreferredLinksLength property. The default value is 60. To specify the length for a specific link, use the following methods:

```
void SetPreferredLength(object link, float length)
float GetPreferredLength(object link)
```

If a specific length is not specified for a link, the global settings are used.

Respect Node Sizes

By default, the layout algorithm respects the size (width and height) of the nodes to support graphs with heterogeneous node sizes. For efficiency reasons, you can turn this option off so that the nodes are approximated with points placed in the center of the bounding box of the nodes. In this mode, when dealing with large nodes, the preferred length parameter can be increased in such a way that the nodes do not overlap.

To change this mode use the RespectNodeSizes property. The default value is **true**.

Force Fit to Layout Region

For this layout algorithm, it is more difficult than for others to choose an appropriate size for the layout region. If the specified layout region is too small for a given graph, the resulting layout will not be the best. For this reason, by default, the Force-Directed Layout algorithm does not use the layout region parameter. It can use as much space as it needs to lay out the graph.

To specify whether the layout algorithm must use the layout region use the ForceFitToLayoutRegion property. The default value of the property is **false**.

For Experts: Additional Features

Expert users can also try and use the following parameters.

Maximum Allowed Move Per Iteration

At each iteration, the layout algorithm moves the nodes a relatively small amount. This amount should not be too large; otherwise the algorithm may not converge. But it should not be too small either, otherwise the number of necessary iterations increases and the running time does also.

The maximum amount of movement at each iteration is controlled by the MaxAllowedMovePerIteration property.

Typical values for this setting are 1 to 30, but it depends on the value of the PreferredLinksLength property. For example, if the setting for the **PreferredLinksLength** property is 1000, then a value of 100 for the **MaxAllowedMovePerIteration** property is still meaningful.

Link Length Weight

The layout algorithm is based on the computation of attraction and repulsion forces for each of the nodes and the iterative search of an equilibrium configuration. One of these forces is related to the objective of obtaining a link length close to the specified preferred length. The weight of this force, representing the total amount of forces, is controlled by the LinkLengthWeight property.

The default value is 1. Increasing this parameter can help obtain link lengths closer to the specified length, but increasing too much can increase the number of link crossings.

Additional Node Repulsion Weight

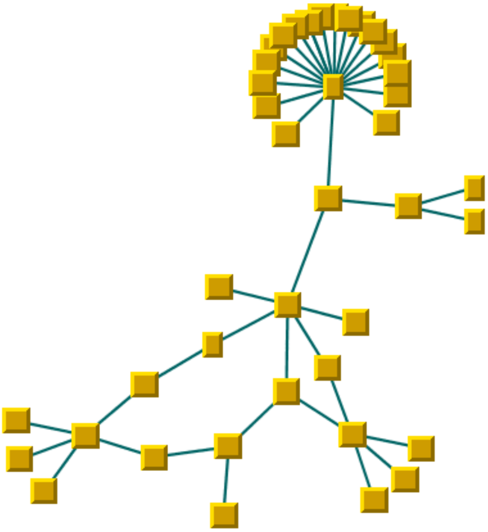
An additional repulsion force can be computed between nodes that are not connected by a link. The weight of this force, representing the total amount of forces, is controlled by the AdditionalNodeRepulsionWeight property.

The default value of this parameter is 0.2.

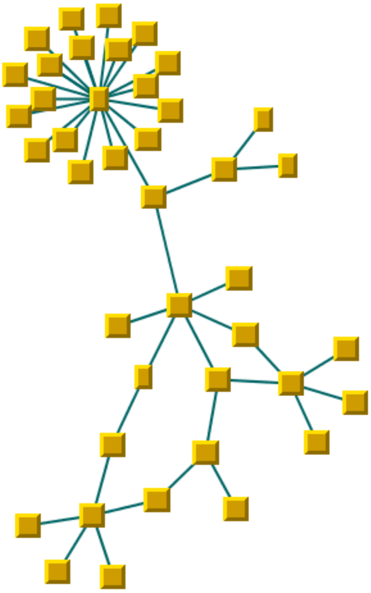
Setting this weight to some small values, for instance 0.2 (that is one-fifth of the default link length weight), can help avoid overlaps between nodes that are not connected by links. However, increasing this value can slow down the convergence of the algorithm.

The following illustrations enable you to compare the same graph laid out with additional repulsion disabled (value of 0) and then enabled. You can see that the star configuration, where many nodes are connected to the same central node, is better displayed when additional repulsion is enabled.

The following illustration shows additional repulsion disabled, produced with the Force-Directed Layout.



The following illustration shows additional repulsion enabled, produced with the Force-Directed Layout.



Node Distance Threshold

The additional repulsion force between two nodes not connected by a link is computed only when their distance is smaller than a predefined distance. To set this distance, use the NodeDistanceThreshold property.

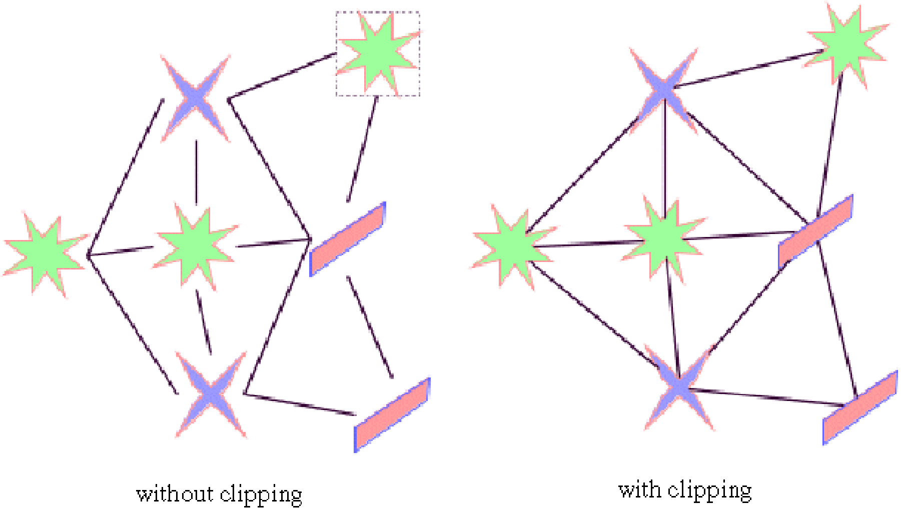
Note that this additional force is computed only if the additional node repulsion weight is set to a value larger than the default value 0.

It is recommended that this threshold be set to a value smaller than the preferred length of the links.

Using the Link Clipping

By default, the Force-Directed Layout does not place the connection points of links. It relies on the anchors of the nodes to determine the connection points. If no anchors are installed at the nodes, the default behavior is to connect to a point at the border of the bounding box of the nodes. If the node has a nonrectangular shape such as a triangle, rhombus, or circle, you may want the connection points to be placed exactly on the border of the shape. This can be achieved by using the link clipping feature. The link clipping corrects the calculated connection point so that it lies on the border of the shape.

The following illustration shows the effect of the link clipping.



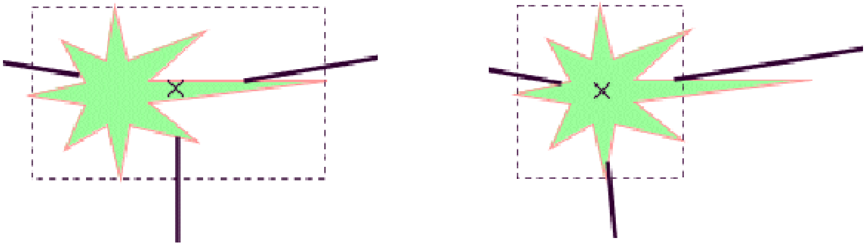
To enable the link clipping, use the property LinkClipping. The default value is **true**.

> ***Note:*** *Additionally to the link clip provider, the ShapeAnchor can be used. This special link anchor updates the clipped connection points automatically during interactive node movements.*

If a node has an irregular shape, the clipped links sometimes should not point towards the center of the node bounding box, but to a virtual center inside the node. You can achieve this by additionally using the link connection box provider. For details, see *Link Connection Box*.

The link connection box provider is used only when link clipping is enabled. Otherwise, the link connection box provider has no effect.

An example of the combined effect is shown in the following illustration.



Clipping at the node bounding box

Clipping at a specified connection box

If the links are clipped at the green irregular star node (left picture), they do not point towards the center of the star, but towards the center of the bounding box of the node. This can be corrected by specifying a link connection box provider that returns a smaller node box than the bounding box (right picture). Alternatively, the problem could be corrected by specifying a link connection box provider that returns the bounding box as the node box but with additional tangential offsets that shift the virtual center of the node.

Link Layout

Use the following topics to learn about the Link Layout algorithms (classes ShortLinkLayout and LongLinkLayout).

In This Section

Samples

Provides some sample drawings produced by the algorithm.

What Types of Graphs?

Explains the type of graph on which you can use the algorithm.

Application Domains

Explains the application domains for the algorithm.

Features

Lists the features of the algorithm.

Limitations

Explains the limitations of the algorithm.

Brief Description of the Algorithm

Provides a short description of the Link Layout algorithm.

Code Sample

Provides a sample of code that shows how to use the algorithm.

Generic Features and Parameters

Explains what generic features of the graph layout library are supported and how.

Specific Parameters for Both Short and Long Layout Layout

Presents the parameters of the Short and Long Link Layout algorithm.

Spacing Parameters in Short Link Layout

Describes the parameters necessary to control the spacing in the Short Link Layout.

Spacing Parameters in Long Link Layout

Describes the parameters necessary to control the spacing in the Long Link Layout.

For Experts: Additional Features

Provides additional information for layout experts.

For Experts: Special Options of the Short Link Layout

Provides special options for the Short Link Layout experts.

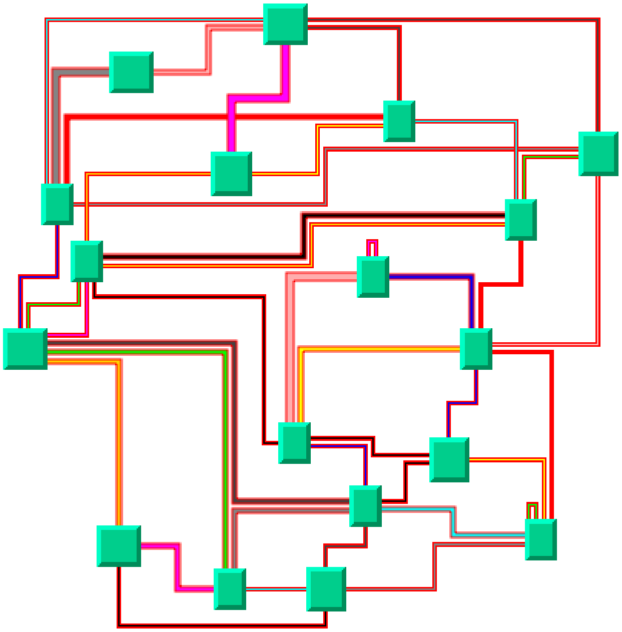
For Experts: Special Options of the Long Link Layout

Provides special options for the Long Link Layout experts.

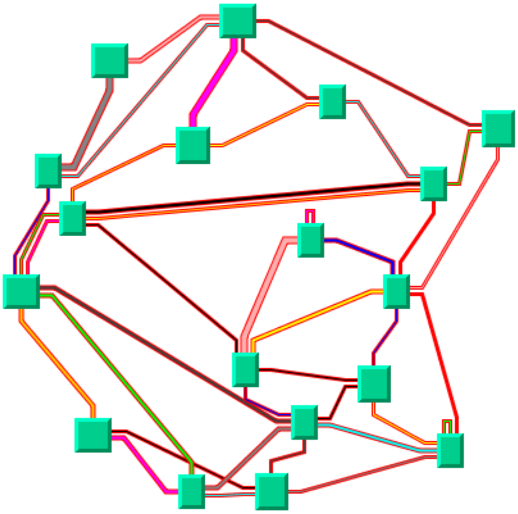
Samples

In this section you can find sample drawings produced with the Short Link Layout.

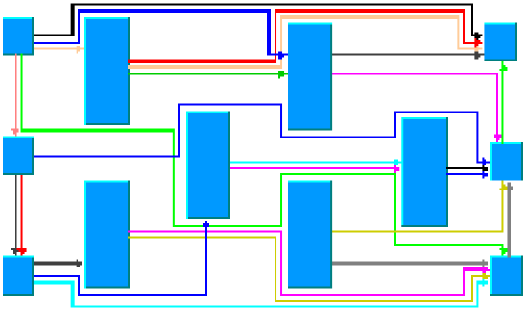
The following illustration shows a Short Link Layout with orthogonal links.



The following illustration shows the same graph in Short Link Layout with direct links.



The following illustration shows a Long Link Layout with orthogonal links.



What Types of Graphs?

Any type of graph where nodes are fixed and links need to be routed:

- Connected and disconnected graphs
- Planar and nonplanar graphs
- Nested graphs with intergraph links

Application Domains

Application domains of the Link Layout include:

- Electrical engineering (circuit block diagrams)
- Industrial engineering (schematic design diagrams, equipment/resource control charts)
- Business processing (entity relation diagrams)
- Software management/software (re-)engineering (data inspector diagrams)
- Database and knowledge engineering (sociology, genealogy)
- CASE tools (design diagrams)

Features

- Reshapes the links of a graph in either an orthogonal or a direct style, without moving the nodes. Orthogonal and direct style links can be combined in the same layout.
- Allows you to specify which side of the node (top, bottom, left, or right) a link can be connected to, or to preserve the existing connection points of the links.
- Supports self-links (that is, links with the same origin and destination node).
- Supports multiple links (that is, more than one link between the same origin and destination nodes).
- Allows you to specify pinned (fixed) links that the layout algorithm cannot reshape.
- Supports intergraph links of nested graphs. An intergraph link is a link whose end nodes belong to different subgraphs of a nested graph.
- Supports an incremental mode: If new links are added to a drawing, the next layout takes the shapes of the old links into account.
- Two link layout algorithms: Short Link Layout with a limited number of bends or Long Link Layout with unlimited number of bends.

Short Link Layout

- Links are placed freely in the space.
- Link-to-link and link-to-node crossings are reduced, if this is possible with link shapes that have a maximum of 4 bends.
- Links of different width are supported.
- Link bundles between the same pair of nodes are supported. Optionally, the algorithm can ensure that multiple links are bundled together by giving them parallel shapes.
- Automatically arranges the final segments of the links (the segments near the origin or destination node) to obtain a bundle of parallel links.
- Provides two optional shapes for the *self-link*s.
- Very fast algorithm with low memory footprint.

Long Link Layout

- Links are placed on a grid.
- Link-to-node crossings of orthogonal links are avoided, even if this introduces many bends.
- Orthogonal link segments do not overlap.
- Does not bundle the final segments. Instead, it distributes the links on the border of each end node according to which border has more free space.
- Fast algorithm: speed and memory footprint depend on the grid spacing.

Limitations

- When routing intergraph links, the incremental mode cannot be used. Due to the complexity of intergraph link routing, more crossings and overlappings may occur than when routing normal links.
- In Short Link Layout, crossings and overlapping of links with other links and nodes cannot always be avoided because the algorithm uses link shapes with a limited number of bends. This happens in particular when there are many obstacles between the connection points of a link.
- In Long Link Layout, link crossings cannot always be avoided. Segment overlappings of orthogonal links are always avoided unless there is no free space remaining on the border of the end nodes. Any overlapping of nodes and links is always avoided unless one end nodes is inside an enclave. An enclave is an area that is surrounded by other nodes such that the area cannot be reached from the other end node.
- In Long Link Layout, segment overlapping or overlapping between nodes and links cannot always be avoided if the direct link style is used.
- The Long Link Layout is slower and uses more memory if the grid spacing is very small.

Brief Description of the Algorithm

The Link Layout algorithms use two layout classes:

- ShortLinkLayout
- LongLinkLayout

Both implement different strategies to find the link shapes.

Short Link Layout Algorithm

The Short Link Layout algorithm is based on a combinatorial optimization that chooses the "optimal" shape of the links to minimize a cost function. This cost function is proportional to the number of link-to-link and link-to-node crossings.

For efficiency reasons, the basic shape of each link is chosen from a set of predefined shapes. These shapes are different for each link style option. For the orthogonal link style, the links are reshaped to a polygonal line of up to five alternating horizontal and vertical segments. For the direct link style, the links are reshaped to a polygonal line composed of three segments: a straight-line segment that starts and ends with small horizontal or vertical segments.

The shape of a link also depends on the relative position of the origin and destination nodes. For instance, when two nodes are very close or they overlap, the shape of the link is chosen to provide the best visibility of the link.

The exact shape of a link is computed taking into account additional constraints. The layout algorithm tries to do the following:

- Minimize the number of crossings between the links incident to a given side of a node.
- Space the final segments of the links incident to a given side of a node equally on the node border.

Long Link Layout Algorithm

The Long Link Layout algorithm first treats each link individually. For each link, it first calculates the connection points at the end nodes that are on the grid and orders them according to a penalty value. Connection points on used grid points have a very high penalty and, therefore, are very unlikely to be used.

For the orthogonal links, the Long Link Layout algorithm then uses a grid traversal to search a route over free grid points from the start connection point to the end connection point. Therefore, in contrast to the short link layout, orthogonal links can have any shape with a large number of bends if this is necessary to bypass obstacle nodes to avoid overlappings. For the direct links, it shortens the search by using a direct segment between the connection points.

After all links are placed, a crossing reduction phase examines pairs of links and eliminates link crossings by exchanging parts of the routes between both links.

The Long Link Layout algorithm relies on the fact that links fit to the grid spacing and parts of the routes between different links can be exchanged. Therefore, the Long Link Layout algorithm does not take the link width into account because it would be too difficult to find the parts of two links that can be exchanged. It is recommended to set the grid spacing larger than the largest link width.

Choosing the Appropriate Link Layout Algorithm

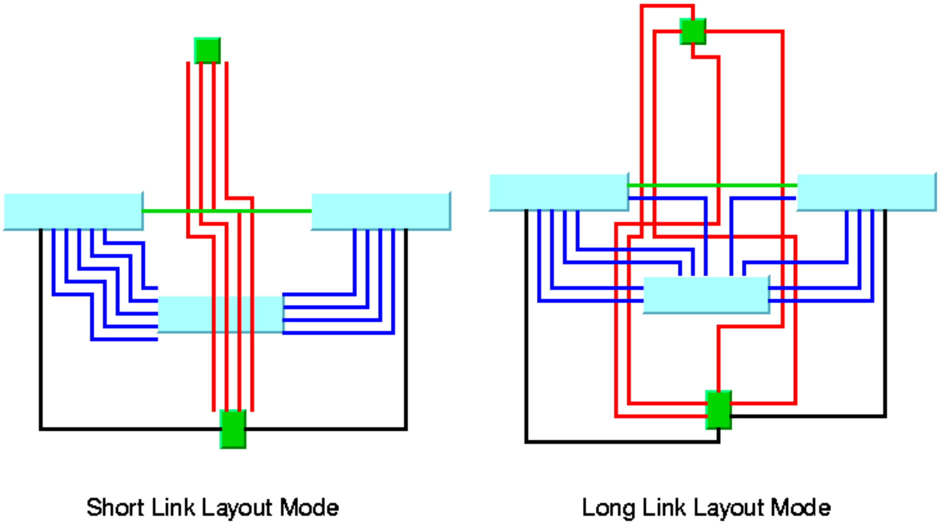
The short link layout should be used if any of the following conditions apply:

- The majority of links is short and it is not fatal if long links overlap obstacles.
- The link routes must be placed freely and cannot be restricted to a grid.
- It is important to limit the number of bends.

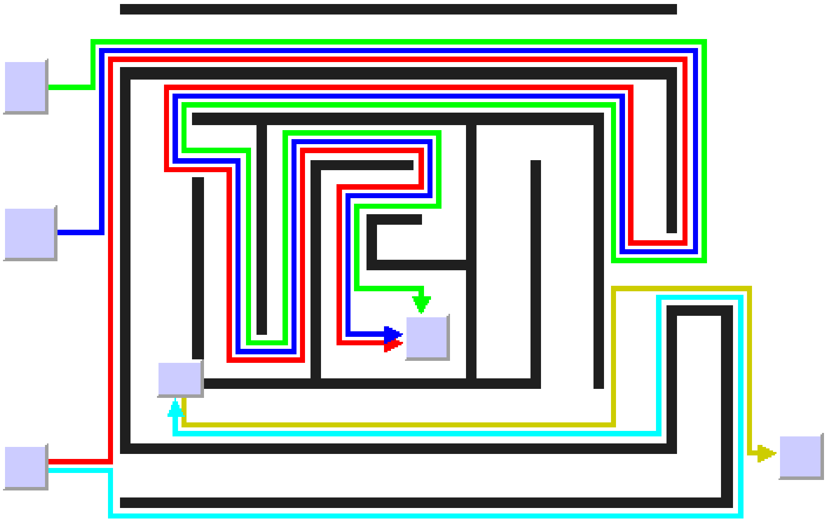
The long link layout should be used if any of the following conditions apply:

- Many links are long and it is important that long links do not overlap obstacles.
- There is a preferred routing because the nodes are already placed on the grid.
- It is important to have a guaranteed a minimal distance between link segments.
- An increasing number of bends is acceptable if it avoids any overlappings.

The illustration below shows a small sample graph in short and long link layout. The short link layout bundles the links very well. However, due to the bundling, some red links appear to be unconnected to the green nodes. Furthermore, the algorithm cannot find a route for the long red links without overlapping some nodes or without overlapping the green link. The long link layout works on a grid. It is specialized for long links and avoids overlapping any nodes or link segments. It can connect to the green nodes by choosing connection points on different sides of the end nodes. This advantage, however, is paid for by a less regular structure that does not bundle the links and a larger number of link crossings.



The following illustration shows how the long link layout can be used to find an orthogonal route without overlappings in a labyrinth of node obstacles.



Code Sample

The following example shows how to perform a Short Link Layout on a Group object by using the ShortLinkLayout class:

```
using System;
using ILOG.Diagrammer;
using ILOG.Diagrammer.Graphic;
using ILOG.Diagrammer.GraphLayout;


…
Group group = new Group();

// Fill the group with nodes and links here

ShortLinkLayout layout = new ShortLinkLayout();

group.LinkLayout = layout;
group.PerformGraphLayout();
Imports System
Imports ILOG.Diagrammer
Imports ILOG.Diagrammer.Graphic
Imports ILOG.Diagrammer.GraphLayout

Dim group As Group = New Group

' Fill the group with nodes and links here

Dim layout As TreeLayout = New TreeLayout
group.LinkLayout = layout
group.PerformGraphLayout()
```

Generic Features and Parameters

The ShortLinkLayout and LongLinkLayout classes support the following generic parameters as defined in the class GraphLayout (see *Layout Parameters and Features in the GraphLayout Base Class*).

- *Allowed Time*
- *Automatic Layout*
- *Preserve Fixed Links*
- *Stop Immediately*

The following comments describe the particular way in which these parameters are used by these subclasses.

Allowed Time

The layout algorithm stops if the allowed time setting has elapsed. For a description of this layout parameter in the **GraphLayout** class, see *Allowed Time*. If the layout stops early because the allowed time has elapsed, some links may not be routed in the best possible way. The result code in the layout report is **GraphLayoutReportCode.StoppedAndInvalid**.

Automatic Layout

The link layout routes the links so that they bypass the nodes and cross each other as few times as possible. It does not position any nodes. However, if the user moves, adds, or resizes nodes, or adds or removes links, the layout usually becomes invalid; that is, the links no longer look orthogonal, overlap the moved nodes, or cross other links.

Using the automatic layout feature of the **GraphLayout** class, the layout is performed again whenever a change of the graph occurs. For a description of this layout parameter in the **GraphLayout** class, see *Automatic Layout*.

Preserve Fixed Links

The layout algorithm does not reshape the links that are specified as fixed. (See *Preserve Fixed Links*) The fixed links are taken into account when computing the optimal layout of the nonfixed links.

Stop Immediately

The layout algorithm stops if the method StopImmediately is called. If the layout stops early, some links may not be routed in the best possible way. The result code in the layout report is **GraphLayoutReportCode.StoppedAndInvalid**.

Specific Parameters for Both Short and Long Layout Layout

The following parameters are specific to both the ShortLinkLayout and LongLinkLayout classes.

Link Style

The link layout algorithms provide two link styles. You can set the link style globally, in which case all links have the same kind of shape, or locally on each link, in which case different link shapes occur in the same drawing.

Global Link Style

To set the global link style, use the ShortLinkLayout.LinkStyle and LongLinkLayout.LinkStyle properties.

The link styles are defined by the LinkLayoutLinkStyle enumeration:

- **LinkLayoutLinkStyle.Orthogonal** (the default)

  The links are reshaped in an orthogonal form (alternating horizontal and vertical segments).

- **LinkLayoutLinkStyle.Direct**

  The links are reshaped to a polygonal line composed of three segments: a straight-line segment that starts and ends with a small horizontal or vertical segment.

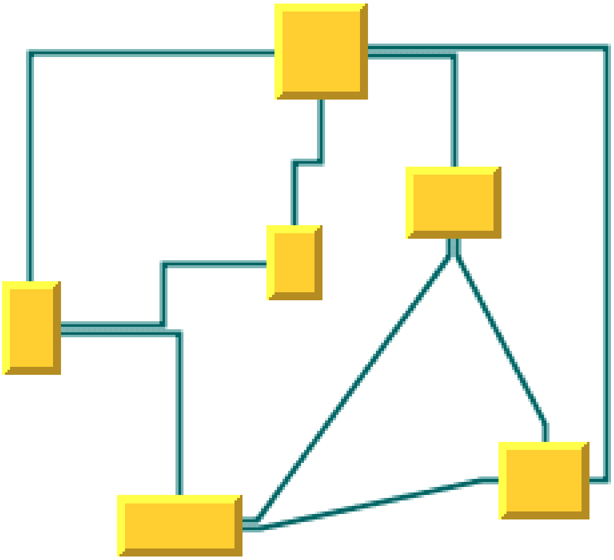
- **LinkLayoutLinkStyle.Mixed**

  Each link can have a different link style. The style of each individual link can be set to have different link shapes occurring on the same graph.

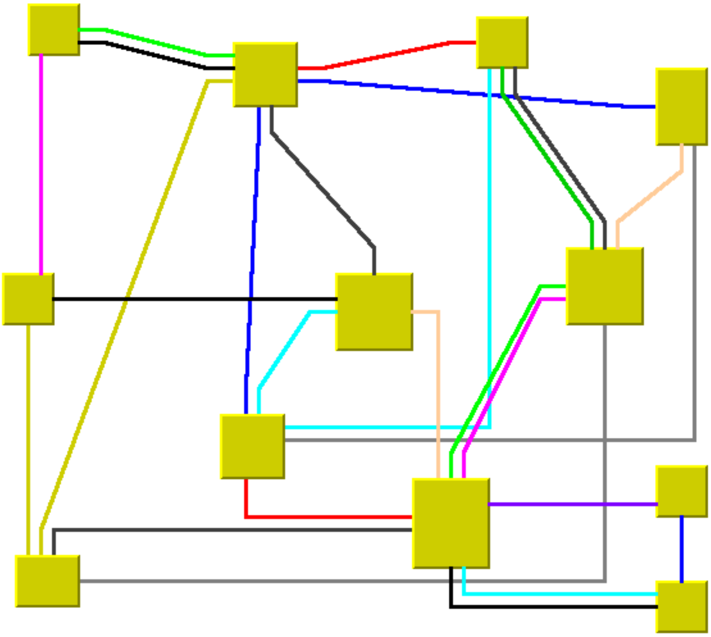
Individual Link Style

All links have the same style of shape unless the global link style is **LinkLayoutLinkStyle.Mixed**. Only when the global link style is set to **LinkLayoutLinkStyle.Mixed** can each link have an individual link style.

The following illustration shows different link styles mixed in the same drawing (Short Link Layout).



The following illustration shows different link styles mixed in the same drawing (Long Link Layout).



To set and retrieve the style of an individual link use the following methods SetLinkStyle(Object link, LinkLayoutLinkStyle style) and GetLinkStyle(Object link).

Connection Points Mode

Normally, the layout algorithm is free to choose the termination points of each link. However, if fixed, non-moveable anchors, the user can specify that the current fixed termination anchor of a link should be used.

The layout algorithm provides two connection point modes. You can set the connection point mode globally, in which case all connection points have the same mode, or locally on each link, in which case different connection point modes occur in the same drawing.

Global Connection Point Mode

To set the global connection point mode, use the following properties:

- ShortLinkLayout.OriginPointMode
- ShortLinkLayout.DestinationPointMode
- LongLinkLayout.OriginPointMode
- LongLinkLayout.DestinationPointMode

The connection point modes are defined by the enumeration ConnectionPointMode:

- **ConnectionPointMode.Free** (the default)

  The layout is free to choose the appropriate position of the connection point on the origin/destination node.

- **ConnectionPointMode.Fixed**

  The layout must keep the current position of the connection point on the origin/ destination node.

- **ConnectionPointMode.Mixed**

  Each link can have a different connection point mode.

The connection points are automatically considered as fixed if they are connected to anchors for which the property Anchor.CanMove is false.

Individual Connection Point Mode

All links have the same connection point mode unless the global connection point mode is **ConnectionPointMode.Mixed**. Only when the global connection point mode is set to **ConnectionPointMode.Mixed** can each link have an individual connection point mode.

Use the following methods to set the mode of an individual link:

To set the mode of an individual link, use the following methods:

- ShortLinkLayout.SetOriginPointMode(Object link, ConnectionPointMode mode)
- ShortLinkLayout.GetOriginPointMode(Object link)
- LongLinkLayout.SetDestinationPointMode(Object link, ConnectionPointMode mode)
- LongLinkLayout.GetDestinationPointMode(Object link)

Incremental Mode

The link layout algorithms normally route all links from scratch. If the graph changes incrementally because you add or remove links or nodes, the subsequent layout may differ considerably from the previous layout. To avoid this effect and to help the user to retain a mental map of the graph, the algorithm has an incremental mode.

To enable the incremental mode, use the properties ShortLinkLayout.IncrementalMode and LongLinkLayout.IncrementalMode.

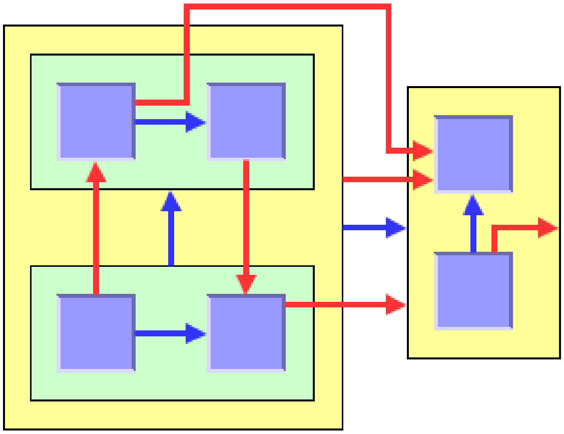
In incremental mode, the layout tries to minimize the changes to the layout. A link is only rerouted if it is new, if a link bend moved, if its layout parameters have changed, or if a node was moved such that it overlaps the link.

In Short Link Layout, if the next layout is incremental, the links preserve the connection side and the general shape calculated by a previous layout, except if one of their end nodes has been moved or resized.

In Long Link Layout, a new route is searched for the links that are no longer on the grid or that overlap with nodes. The shape and the connection side of the rerouted links can change completely. However, links that are already on the grid and do not overlap nodes or other links are not rerouted in incremental mode. It is also possible to specify which link must be rerouted by the next incremental layout even though the layout has not changed. To select an individual link to be used for incremental rerouting, use the method MarkForIncremental(Object link).

Intergraph Link Routing

A nested graph is a graph with nodes that are subgraphs. In a nested graph, normal links and intergraph links can occur. Normally, both end nodes of a link belong to the same subgraph. Intergraph links are those links whose end nodes belong to different subgraphs. Intergraph links belong to the lowest common graph in the nesting structure that contains both end nodes. The following illustration shows a nested graph with blue normal links and red intergraph links.



By default, the link layouts route both the normal links and the intergraph links. In order to route only normal links, disable intergraph link routing using the properties ShortLinkLayout.InterGraphLinksMode and LongLinkLayout.InterGraphLinksMode.

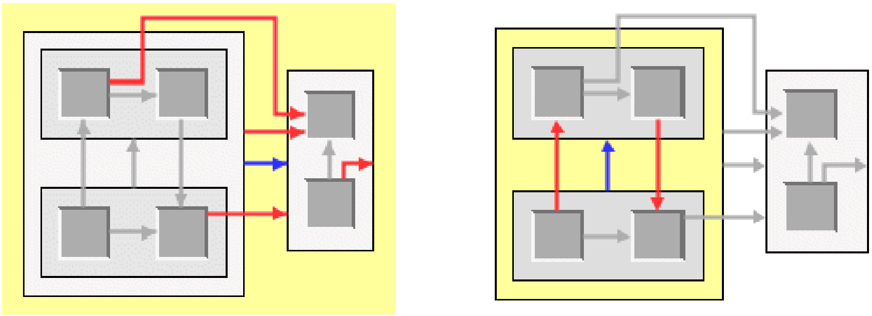
If the intergraph links mode is enabled, you can select whether only the intergraph links are routed, or whether the intergraph links and the normal links are routed at the same time using the properties ShortLinkLayout.CombinedInterGraphLinksMode and LongLinkLayout.CombinedInterGraphLinksMode.

If this property is set to **false**, the next layout routes the intergraph links but does not reshape any normal links. If the property is set to **true**, the next layout routes both the normal links and the intergraph links.

When the intergraph links mode is enabled, the layout cannot route the links incrementally (see *Incremental Mode*).

Note that the layout routes only those links that belong to the attached graph. In a nested graph, each subgraph is attached to a different layout instance. Therefore, when starting a normal (nonrecursive) layout for the top-level graph of, not all links are routed that are shown in that figure, but only those links that belong to the top-level graph.

The following illustration shows two situations: the yellow subgraph indicates the subgraph where the nonrecursive layout is currently applied, and the color of the links indicates which links are currently routed. Depending on the intergraph links mode, the red and/or blue links are routed, but the grey links are not reshaped.



To route all links of a nested graph, you need to apply the Link Layout recursively. Details of the recursive layout mechanism are explained in *What is Recursive Layout?*. For instance:

```
layout.InterGraphLinksMode = true;
layout.PerformLayout(force, redraw, true);
```

routes the intergraph links recursively in all subgraphs. If you use a layout provider (a class that implements the interface ILayoutProvider), you need to set the intergraph links mode for all subgraphs explicitly:

```
ILayoutProvider layoutProvider = ...
// first, set the intergraph mode for all layouts
IEnumerator layouts =
GraphLayoutUtil.GetLayouts(graphModel, layoutProvider, true);

while (layouts.MoveNext()) {
    GraphLayout layout = (GraphLayout)layouts.Current;
    if (layout is ShortLinkLayout)
        ((ShortLinkLayout)layout).InterGraphLinksMode = true;
    if (layout is LongLinkLayout)
        ((LongLinkLayout)layout).InterGraphLinksMode = true;

}
// then perform layout recursively using the provider
GraphLayout.PerformLayout(graphModel, layoutProvider, force, true);
```

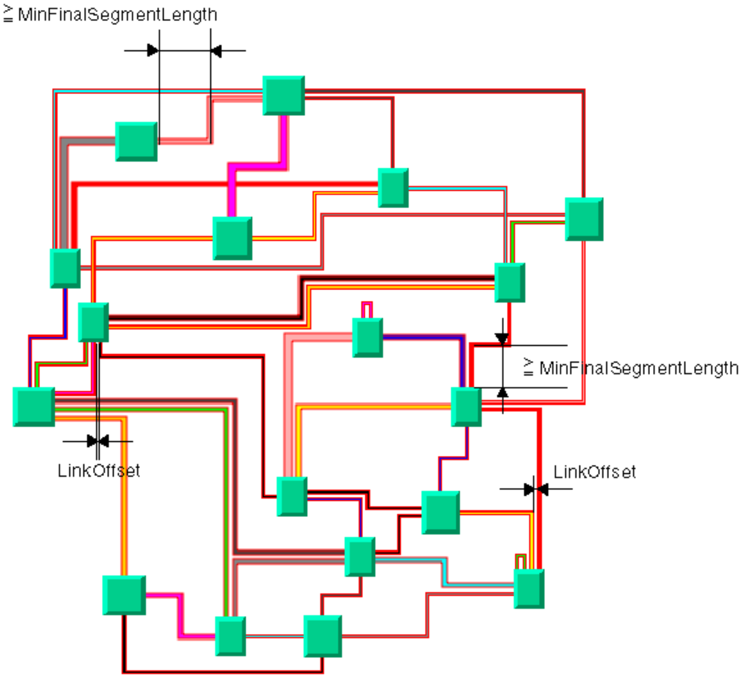
```
Dim layoutProvider As ILayoutProvider = ...
' first, set the intergraph mode for all layouts
Dim layouts As IEnumerator = _
 GraphLayoutUtil.GetLayouts(graphModel, layoutProvider, True)
While layouts.MoveNext
 Dim layout As GraphLayout = CType(layouts.Current, GraphLayout)
 If TypeOf layout Is ShortLinkLayout Then
   CType(layout, ShortLinkLayout).InterGraphLinksMode = True
 End If
 If TypeOf layout Is LongLinkLayout Then
   CType(layout, LongLinkLayout).InterGraphLinksMode = True
 End If
End While
' then perform layout recursively using the provider
GraphLayout.PerformLayout(graphModel, layoutProvider, force, True)
```

If you want to recursively perform the intergraph link routing in combination with a layout that places the nodes or that arranges labels, we recommend that you use an instance of the class (XREF) Multiple Layout to encapsulate the link layout and the other layouts, and then perform the Multiple Layout recursively all at once. For details, see XREF Recursive Layout.

Spacing Parameters in Short Link Layout

Since the Short Link Layout places the links freely in the space, only two parameters are necessary to control the spacing: the minimal distance between links and the minimal length of the final segment.

The following illustration shows the spacing parameters used in the Short Link Layout.



Link Offset

The layout algorithm computes the final connecting segments of the links (that is, the segments near the origin and destination nodes) to obtain parallel lines spaced at a user-defined distance. The Short Link Layout takes into account the width of the links when computing the offset. To specify the offset, use the LinkOffset property.

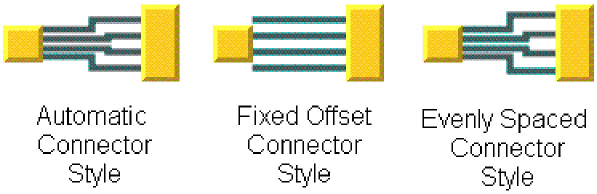
The offset is measured from the border of one link to the nearest border of the other link. Therefore, if the specified offset is zero, the border of a link touches the border of its neighbor link.

Minimum Final Segment Length

You can specify a minimum value for the length of the final connecting segments of the links (that is, the segments near the origin and destination nodes). To do so, use the property MinFinalSegmentLength.

Connector Style

The layout algorithm positions the connection points of links (the connection points) at the nodes automatically. The connector style parameter specifies how these connection points are calculated.


Automatic Connector Style
Fixed Offset Connector Style
Evenly Spaced Connector Style

The layout algorithm provides two connector styles. You can set the connector style globally, in which case all the nodes (hence, all the links) have the same kind of connector style, or locally on each node (that is, for all the links connected to the node), in which case different connector styles occur in the same drawing.

Global Connector Style

To specify the global connector style, use the property ConnectorStyle.

The styles are defined by the ShortLinkLayoutConnectorStyle enumeration:

- **ShortLinkLayoutConnectorStyle.FixedOffset**

  The connection points are spaced along the node border at a distance equal to the link offset parameter.

- **ShortLinkLayoutConnectorStyle.EvenlySpaced**

  The connector pins are evenly spaced along the node border, preserving a margin which is determined by the EvenlySpacedConnectorMarginRatio property.

- **ShortLinkLayoutConnectorStyle.Automatic** (the default)

  Uses the connector style **FixedOffset** except if this pushes a connection point outside the border the link is attached to, in which case it uses the connector style **EvenlySpaced**.

- **ShortLinkLayoutConnectorStyle.Mixed**

  Each node can have a different connector style. The style of each individual node can be set to have different connector styles occurring on the same graph.

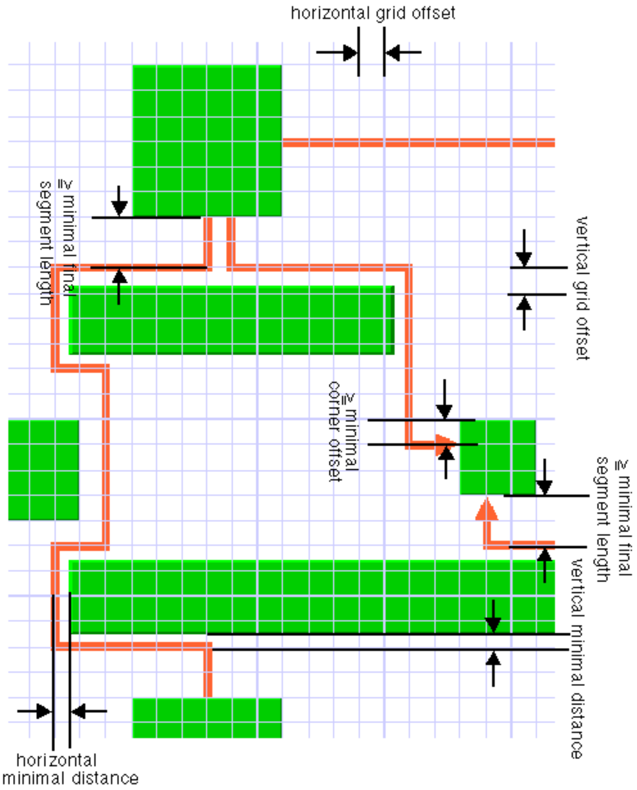
Individual Connector Style

All nodes have the same connector style unless the global connector style is **ShortLinkLayoutConnectorStyle.Mixed**. Only when the global connector style is set to **Mixed** can each node have an individual connector style. To specify the connector style of an individual node, use the following methods SetConnectorStyle(Object node, ShortLinkLayoutConnectorStyle style) and GetConnectorStyle(Object node).

The default value is 10.

Spacing Parameters in Long Link Layout

The Long Link Layout places the links on a grid. Four parameters control the grid offsets and five parameters control the spacing of links in relation to other objects. The following illustration shows the spacing parameters used in the Long Link Layout.


horizontal grid offset
≥ minimal final segment length
vertical grid offset
≥ minimal corner offset
≥ minimal final segment length
vertical minimal distance
horizontal minimal distance

Grid Offset Parameters

The grid offset parameters control the spacing between grid lines. Links are routed such that the center of the orthogonal link segments is on the grid lines. The grid offsets should be set to a value larger than the largest link width value to avoid links that visually overlap.

To set the horizontal and vertical grid offset, use the properties HorizontalGridOffset and VerticalGridOffset.

The grid offset is the critical parameter for the Long Link Layout. If the grid offset is too large, there may be no grid lines between nodes even though some free space exists between the nodes. In this case, the link routings cannot use the free space. However, if the grid offset is too small, the algorithm needs a long time to traverse the grid.

Grid Base Parameters

Sometimes it is necessary to shift the whole grid by a small amount because the nodes are not aligned on the grid. For instance, to have grid lines at positions 3, 13, 23, 33, and so on, you can set the grid offset to 10 and the grid base to 3. To adjust the grid base, use the properties HorizontalGridBase and VerticalGridBase.

Minimal Distance Parameters

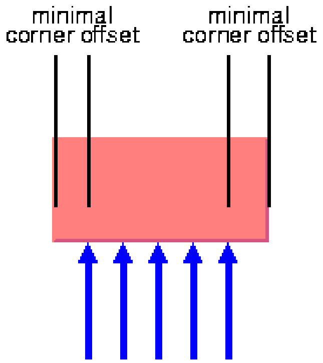
The minimal distance controls how closely a link can be placed to the border of a node that needs to be bypassed. If the node border is not aligned to the grid, the minimal distance specifies the next grid line close to the border that can be used. For instance, if a node covers the x-coordinates 25 to 65 on a grid with offset 10 and base 0, the next grid lines used to bypass the node would normally be at 20 and 70. If we specify a minimal distance of 8, these grid lines are too close to the node and then the grid lines at 10 and 80 would be used.

To set the minimal distance, use the properties HorizontalMinOffset and VerticalMinOffset.

Minimal Node Corner Offset Parameter

The minimal corner offset is the minimal distance between a node corner and a link that connects to the node. This parameter is used to avoid having a link that connects exactly to the corner or outside the border of the node.

To set the minimal corner offset, use the property MinNodeCornerOffset.



Minimum Final Segment Length

As with the Short Link Layout, the Long Link Layout respects the minimum value for the length of the final connecting segments of the links. To set the minimal length of the final segment, use the property MinFinalSegmentLength.

For Experts: Additional Features

The following features are available in both Short and Long Link Layouts.

Using a Node-Side Filter

Some applications require that links are not connected to specific sides of certain nodes. The Link Layout algorithms allow you to restrict to which node side a link can connect by using a node-side filter. A node-side filter is any class that implements the interface INodeSideFilter. This interface defines the following method:

```
bool Accept(IGraphModel graphModel, object link, bool origin,
            object node, NodeSide side);
```

This method allows you to let the input link to connect its origin or destination to the input side of the input node.

As an example, assume that the application requires that for end nodes having the name "Obj1" links can connect their origin only at the top and bottom side, for end nodes having the name "Obj2" links can connect their destination only at the left and right side, and links can connect on all sides for the other nodes. You can obtain this result with the following node-side filter:

```
public class MyFilter : INodeSideFilter
{
   public bool Accept(IGraphModel graphModel,
                      Object link,
                      bool origin,
                      Object node,
                      NodeSide side)
   {
     if (node.Name.Equals("Obj1") && origin)
       return (side == NodeSide.Top || side == NodeSide.Bottom);
     if (node.Name.Equals("Obj2") && !origin)
       return (side == NodeSide.Left || side == NodeSide.Right);
     return true;
   }
}
Public Class MyFilter
Implements INodeSideFilter

 Public Function Accept(ByVal graphModel As IGraphModel, ByVal link As Object,_
               ByVal origin As Boolean, ByVal node As Object,_
               ByVal side As NodeSide) As Boolean
   If node.Name = "Obj1" AndAlso origin Then
     Return (side = NodeSide.Top OrElse side = NodeSide.Bottom)
   End If
   If node.Name = "Obj2" AndAlso Not origin Then
     Return (side = NodeSide.Left OrElse side = NodeSide.Right)
   End If
   Return True
 End Function
End Class
```

To set this node-side filter, use the properties ShortLinkLayout.NodeSideFilter and LongLinkLayout.NodeSideFilter. To remove a filter, set the property to **null**.

Using a Node Box Interface

Some applications require that effective area of a node is not exactly its bounding box. For instance, if the node has a shadow, the shadow is included in the bounding box. However, the shadow may not be considered as an obstacle for the links. In this case, the effective bounding box of a node is smaller than the bounding box returned by GraphicObject.GetStyledBounds.

You can modify the effective bounding box of a node by implementing a class that implements the interface INodeBoxProvider. This interface defines the following method:

```
Rectangle2D GetBox(IGraphModel graphModel, Object node);
```

This method allows you to return the effective bounding box. For instance, to obtain a node box provider that returns a smaller bounding box for all nodes having the name "Obj1":

```
public Rectangle2D GetBox(IGraphModel graphModel, Object node)
{
    Rectangle2D rect = graphModel.BoundingBox(node);
    if (node.Name == "Obj1") {
      // need a rect that is 4 units smaller
      rr.Inflate(-4, -4);
    }
    return rect;
}
Public Function GetBox(ByVal graphModel As IGraphModel, ByVal node As Object)
As Rectangle2D
 Dim rect As Rectangle2D = graphModel.BoundingBox(node)
 ' For some nodes, we need a rect that is 4 units smaller
 If node.Name = "Obj1" Then
   rr.Inflate(-4, -4)
 End If
 Return rect
End Function
```

To set this provider, use the NodeBoxProvider property. To remove a provider, set the property to **null**.

Using a Link Connection Box Interface

By default, the connection points of the links are distributed on the border of the bounding box of the nodes. Sometimes, it may be necessary to place the connection points on a rectangle that is smaller or larger than the bounding box. For instance, this can happen when labels are displayed below or above nodes.

You can modify the position of the connection points of the links by implementing a class that implements the interface ILinkConnectionBoxProvider. This is a subinterface of **INodeBoxProvider**. It defines again the method:

```
Rectangle2D GetBox(IGraphModel graphModel, object node);
```

This method allows you to return the effective rectangle on which the connection points of the links are placed.

Additionally, the interface **ILinkConnectionBoxProvider** defines a second method:

```
float GetTangentialOffset(IGraphModel graphModel, object node, NodeSide
nodeSide);
```

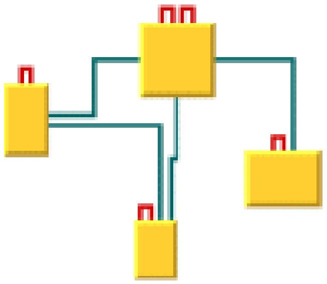
This method is used only in the Short Link Layout. When using the Long Link Layout, just implement the method by returning the value 0.

For Experts: Special Options of the Short Link Layout

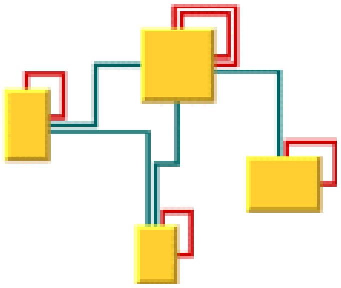
This section describes the customization capabilities which are specific to the Short Link Layout.

Self-link Style

Self-links are links whose origin and destination are the same node. The Short Link Layout provides two optional shapes for *self-links*.



Two-bend Self-link Style



Three-bend Self-link Style

To set the style of the self-links, use the property SelfLinkStyle.

The values for style are defined by the ShortLinkLayoutSelfLinkStyle enumeration:

- **ShortLinkLayoutSelfLinkStyle.TwoBendsOrthogonal**
- **ShortLinkLayoutSelfLinkStyle.ThreeBendsOrthogonal**

Number of Optimization Iterations

The link shape optimization is stopped if the time exceeds the allowed time (see *Allowed Time*) or if the number of iterations exceeds the allowed number of iterations. To set this number, use the property AllowedNumberOfIterations. The default value is 3.

> ***Note:*** *You may want to disable the link shape optimization by setting the number of iterations to zero to increase the speed of the layout process.*

Evenly Spaced Pins Margin Ratio

The margin ratio allows you to customize the way connection points are computed when the connector style (see *Connector Style*) is **ShortLinkLayoutConnectorStyle.EvenlySpaced**, and when the **ShortLinkLayoutConnectorStyle.Automatic** places the connection points using the **ShortLinkLayoutConnectorStyle.EvenlySpaced** style. This option has no effect if the connector style **ShortLinkLayoutConnectorStyle.EvenlySpaced** is used.

In the "evenly spaced" connector style, the connection points of the links are evenly spaced along the node border, preserving a margin to each extremity of the node border. The size of this margin is controlled by the margin ratio and is computed by multiplying the offset between the links by the ratio.

To specify this option, use the **EvenlySpacedConnectorMarginRatio** property.

The specified value must be a positive or zero value. The default value is 0.5. The following table shows examples of values with their meaning.

Ratio value	**Meaning**
0	No margin.
0.5 (default value)	The margin is equal to half the offset between the links.
1	The margin is equal to the offset between the links.

Link Overlap Nodes Forbidden

This option allows you to ask the layout algorithm to avoid strictly to reshape links such that they overlap some nodes. If overlaps are not forbidden, the algorithm tries to avoid overlaps anyway, but may create overlaps, for instance for the link to cross other links.

> ***Note:*** *Forbidding overlaps may slow down the layout and may increase the number of bends for those links that would overlap nodes if overlaps were not strictly forbidden.*

To specify this option, use the property LinkOverlapNodesForbidden.

The default value of this option is **false**.

When overlaps are forbidden, the Short Link Layout algorithm uses the Long Link Layout as an auxiliary algorithm for laying out only the links that would otherwise overlap nodes.

You can retrieve the auxiliary instance of Long Link Layout using the method GetAuxiliaryLongLinkLayout.

This method allows you to get this auxiliary layout instance and to customize its parameters if needed. Notice that you should neither modify the origin and destination point mode, nor disable the preservation of fixed links. Note also that an IGraphModel instance is attached to the LongLinkLayout instance only if needed, therefore the method **GetAuxiliaryLongLinkLayout.GetGraphModel** may return **null**.

Incremental Link Reshape Mode

In incremental mode, it is possible to customize the rules used by the Short Link Layout to determine which links should keep their current shape as much as possible, as computed by the previous layout execution. The incremental link reshape mode allows you to customize these rules separately for two categories of links (for the concepts of "link connection box" and "node box", see also the properties **LinkConnectionBoxProvider** and **NodeBoxProvider**):

- Modified links: the links that have either a different "link connection box" or are connected to nodes which have a different bounding box as during the previous layout execution.
- Unmodified links: the links that have the same "link connection box" and are connected to nodes which have the same bounding box as during the previous layout execution.

The mode can be customized either for both or for only one of these categories of links.

The incremental link reshape mode has no effect if the incremental mode is disabled.

The layout algorithm provides two incremental link reshape modes. You can set the mode globally, in which case all the links have the same mode, or locally on each link, in which case different modes occur in the same drawing.

Global Incremental Link Reshape Mode

To specify the global incremental link reshape mode, use the properties IncrementalModifiedLinkReshapeMode and IncrementalUnmodifiedLinkReshapeMode.

The modes are defined by the enumeraiton ShortLinkLayoutLinkReshapeMode:

- **ShortLinkLayoutLinkReshapeMode.FixedShapeType** (the default)

  The incremental layout preserves the shape type of the link. This means that both the number of bends and the node sides to which the link is connected are preserved.

- **ShortLinkLayoutLinkReshapeMode.FixedNodeSides**

  The incremental layout preserves the node sides to which the links are connected.

- **ShortLinkLayoutLinkReshapeMode.FixedConnectionPoints**

  The incremental layout preserves the connection points of the links.

- **ShortLinkLayoutLinkReshapeMode.Fixed**

  The links are not reshaped at all during incremental layout. Only newly added links are rerouted.

- **ShortLinkLayoutLinkReshapeMode.Free**

  The incremental layout is allowed to freely reshape the links. This is equivalent to a non-incremental behavior for all the links, hence it is recommended to disable the incremental mode instead of using **Free** as global incremental reshape mode.

  Of course, the settings that may have been done by "fixing" links (see *Preserve Fixed Links*) or by customizing the origin or destination point mode (see *Connection Point Mode*) are still respected.

- **ShortLinkLayoutLinkReshapeMode.Mixed**

  Each link can have a different mode.

Individual Incremental Link Reshape Mode

All links have the same incremental link reshape mode unless the global incremental link reshape mode is **ShortLinkLayoutLinkReshapeMode.Mixed**. Only when the global mode is set to **Mixed** can each link have an individual mode. To specify the mode of an individual link use the following methods:

- SetIncrementalModifiedLinkReshapeMode (Object link, ShortLinkLayoutLinkReshapeMode mode)
- GetIncrementalModifiedLinkReshapeMode(Object link)
- SetIncrementalUnmodifiedLinkReshapeMode (Object link, ShortLinkLayoutLinkReshapeMode mode)
- GetIncrementalUnmodifiedLinkReshapeMode(Object link)

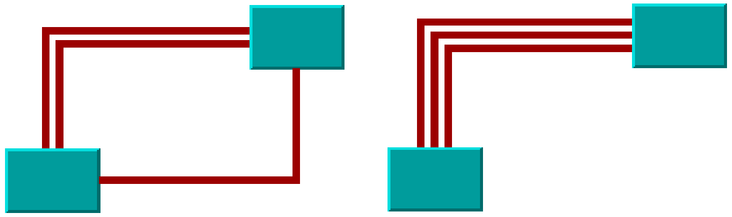
The valid values for mode are:

- **ShortLinkLayoutLinkReshapeMode.FixedShapeType** (the default)
- **ShortLinkLayoutLinkReshapeMode.FixedNodeSides**
- **ShortLinkLayoutLinkReshapeMode.FixedConnectionPoints**
- **ShortLinkLayoutLinkReshapeMode.Free**
- **ShortLinkLayoutLinkReshapeMode.Fixed**

Same Shape for Multiple Links

You can force the layout algorithm to compute the same shape for all the links having common origin and destination nodes. The links will have parallel shapes.

When this option is disabled, the layout is free to compute different shapes for links connecting the same pair of nodes. Generally, different shapes are chosen to avoid some overlaps.



Same-Shape Option Disabled

Same-Shape Option Enabled

To enable this option, use the SameShapeForMultipleLinks. The default value is **false**.

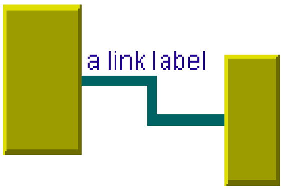
Link Crossing Penalty

The computation of the shape of the links is driven by the objective to minimize a cost function, which is proportional to the number of link-to-link crossings and link-to-node crossings. By default, these two types of crossings have equal weights of 1. You can increase the weight of the link-to-node crossings. To do so, set the value 5 (for instance) for the property LinkToNodeCrossingPenalty. This increases the possibility of obtaining a layout with no link-to-node crossings (or with only a few crossings), with the expense that there may be more link-to-link crossings.

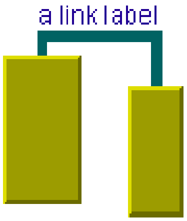
Alternatively, you can increase the weight of the link-to-link crossings. To do so, set the value 3 (for instance) for the property LinkToLinkCrossingPenalty. This increases the possibility of obtaining a layout with no link-to-link crossings (or with only few crossings), with the expense that there may be more link-to-node crossings.

Bypass Distance

If the origin and destination nodes are too close, there may not be enough space for routing the link directly between the end nodes. Therefore, by default, if the end nodes are closer than a threshold distance, the layout chooses link shapes that bypass the interval between close nodes.



End-nodes distance larger than the bypass distance



End-nodes distance smaller than the bypass distance

The bypass distance is the minimum distance between the origin and destination nodes for which a link shape going directly from one node to another is allowed. The algorithm tries to avoid link shapes that connect directly the sides of the end nodes that are closer than the bypass value.

To set the bypass distance, use the property BypassDistance.

The default value is a strictly negative value. If the bypass distance is strictly negative, the value of the minimum final segment length (see *Minimum Final Segment Length*) parameter is used as the bypass distance. This allows the automatic adjustment of the bypass distance according to the current value of the minimum final segment length. This behavior is suitable in most cases. However, you can specify a non-negative value to override the default behavior.

Using a Link Connection Box Interface

By default, the connection points of the links are distributed on the border of the bounding box of the nodes, symmetrically with respect to the middle of each side. Sometimes, it may be necessary to place the connection points on a rectangle smaller or larger than the bounding box, eventually in a nonsymmetric way. For instance, this can happen when labels are displayed below or above nodes.

You can modify the position of the connection points of the links by implementing a class that implements the ILinkConnectionBoxProvider. This interface defines the following method:

```
Rectangle2D GetBox(IGraphModel graphModel, Object node);
```

This method allows you to return the effective rectangle on which the connection points of the links are placed.

A second method defined on the interface allows the connection points to be "shifted" tangentially, in a different way for each side of each node:

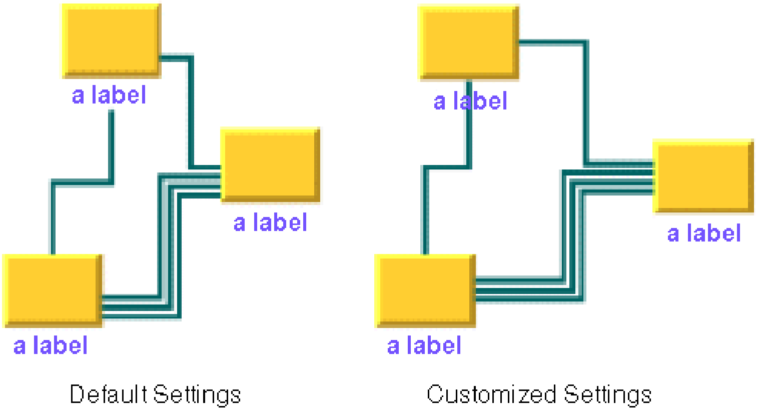
```
float GetTangentialOffset(IGraphModel graphModel, Object node, NodeSide side);
```

For instance, to set a link connection box interface that returns a link connection rectangle that is smaller than the bounding box for all nodes that match a given criteria and shifts up the connection points on the left and right side of all the nodes, implement the interface as follows:

```
public class MyProvider : ILinkConnectionBoxProvider
{
  public Rectangle2D GetBox(IGraphModel graphModel, Object node)
  {
    Rectangle2D rect = graphModel.BoundingBox(node);
    if (node.Name == "Obj1") {
      // need a rect that is 4 units smaller
      rr.Inflate(-4, -4);
    }
    return rect;
  }

  public float GetTangentialOffset(IGraphModel graphModel,
                                 Object node, NodeSide side) {
    switch (nodeSide) {
      case NodeSide.Left:
      case NodeSide.Right:
        return -10; // shift up with 10 for both left and right side
      case NodeSide.Top:
      case NodeSide.Bottom:
      default:
        return 0; // no shift for top and bottom side
    }
}
}
Public Class MyFilter
Implements INodeSideFilter

 Public Function GetBox(ByVal graphModel As IGraphModel, ByVal node As Object)
As Rectangle2D
   Dim rect As Rectangle2D = graphModel.BoundingBox(node)
   ' For some nodes, we need a rect that is 4 units smaller
   If node.Name = "Obj1" Then
     rr.Inflate(-4, -4)
   End If
   Return rect
  End Function

  Public Function GetTangentialOffset(ByVal graphModel As IGraphModel, ByVal
node As Object, ByVal side As NodeSide) As Single
   Dim rect As Rectangle2D = graphModel.BoundingBox(node)
  Select side
   Case NodeSide.Left, NodeSide.Right
     Return -10
   Case NodeSide.Top, NodeSide.Bottom, Else
     Return 0
   End Select
End Function
End Class
```

To set this provider, use the LinkConnectionBoxProvider property. To remove a provider, set the property to null.

The following illustration shows the effects of customizing the connection box. On the left is the result using the default settings: the connection points are distributed on the bounding box of the node (which includes the label) and are symmetric with the middle of each node side (including the label). On the right, is the result after specifying a link connection box provider. On the bottom side of the nodes, the links are now connected to the node (passing over the label), while on the left and right side the nodes are now symmetric to the middle of the node (without the label).

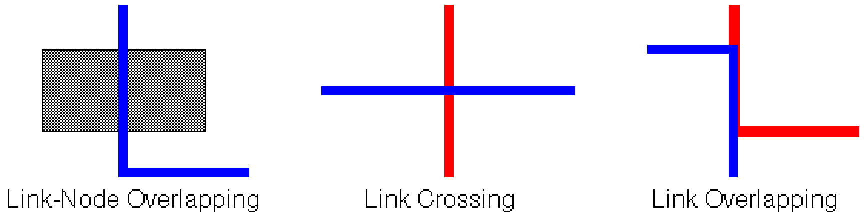


For Experts: Special Options of the Long Link Layout

This section describes the customization capabilities which are specific to the Long Link Layout.

Specifying Additional Obstacles

The Long Link Layout algorithm considers nodes to be obstacles that cannot be overlapped and links to be obstacles that can be crossed at an angle of 90 degree (approximately, if the link style is direct), but that cannot be overlapped.



If an application requires additional obstacles that are not links or nodes, these can be specified using the methods AddRectObstacle(Rectangle2D) and AddLineObstacle(Point2D, Point2D).

Rectangular obstacles behave like nodes: links cannot overlap the rectangles. Line obstacles behave like link segments: other links can cross the line segments, but cannot overlap the segments. These obstacle settings can be removed by means of the methods RemoveAllRectObstacles and RemoveAllLineObstacles.

Penalties for Variable Connection Points

If the termination points of the links are not fixed, the algorithm uses a heuristic to calculate the termination points of each link. It examines all free grid points that are close to the border of the start and end node and assigns a penalty to each grid point. If a node-side filter is installed, the penalty depends on whether the node side is allowed or rejected.

A more precise way to affect how the termination points are chosen is the termination point filter. This enables the user to specify the penalty for each grid point. A termination point filter is a class that implements the interface ILongLinkLayoutTerminationPointFilter that defines the following method:

```
int GetPenalty(IGraphModel graphModel, object link, bool origin,
        object node, Point2D point, int side, int proposedPenalty);
```

To select the origin or destination point of the input link, the input point (a grid point on the input side of the node) is examined. The `proposedPenalty` is calculated by the default heuristic of the algorithm. You can return a changed penalty or you can return `Int32.MaxValue` to reject the grid point. If the grid point is rejected, it is not chosen as termination point of the link.

The termination point filter can be set using the property TerminationPointFilter.

Manipulating the Routing Phases

As mentioned in *Long Link Layout Algorithm*, the algorithm first treats each link individually and then applies a crossing reduction phase to all links. To find a route for an individual link, the algorithm first checks whether a routing (such as a straight line or with only one bend) is possible. If this kind of routing is not possible, it uses a sophisticated, but more time consuming, grid search algorithm with backtracking to find a route with many bends.

To switch off the phase that finds a straight-line or one-bend routing, set the property StraightRouteEnabled to **false**.

The backtrack search for a route with many bends can be affected in the several ways.

You can specify the maximal number of backtrack steps by using the property MaxBacktrack.The default maximal backtrack number is 30000.

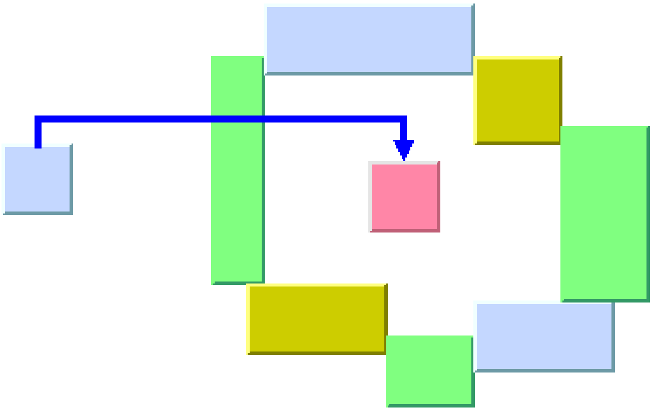
A more convenient way is to specify the maximal time available to search for the route for each link, using the property AllowedTimePerLink.The default allowed time per link is 2000 milliseconds (2 seconds).

Finally, you can specify how many steps should be done during the crossing reduction phase, using the property NumberCrossingReductionIterations.

You can disable the crossing reduction completely by setting this property to **false**.

Fallback Mechanism

The Long Link Layout algorithm may not be able to find a routing for a link, if a connection node is inside an enclave. In the following illustration, the red node is inside an enclave. In this case, the backtrack search algorithm fails to find a routing without overlapping nodes. The backtrack search algorithm may also fail if the situation is so complex that the search exceeds the allowed time per link.



When the backtrack search algorithm fails to find a routing, a simple fallback mechanism is applied that creates a routing with node overlappings. To disable this fallback mechanism, set the property FallbackRouteEnabled to **false**.

If the fallback mechanism is disabled, these links are not routed at all and remain in the same shape as before the layout.

You can retrieve the links that could not be routed in the usual way without the fallback mechanism, using the method GetCalcFallbackLinks.

For instance, you can iterate over these links and apply your own specific fallback mechanism instead of the default fallback mechanism of the Long Link Layout algorithm.

Grid Layout

This section describes the Grid Layout algorithm (class GridLayout).

In This Section

Sample

Provides some sample drawings produced by the algorithm.

What Types of Graphs?

Explains the type of graph on which you can use the algorithm.

Application Domains

Explains the application domains for the algorithm.

Features

Lists the features of the algorithm.

Brief Description of the Algorithm

Provides a short description of the grid layout algorithm.

Code Sample

Provides a sample of code that shows how to use the algorithm.

Generic Features and Parameters

Explains what generic features of the graph layout library are supported and how.

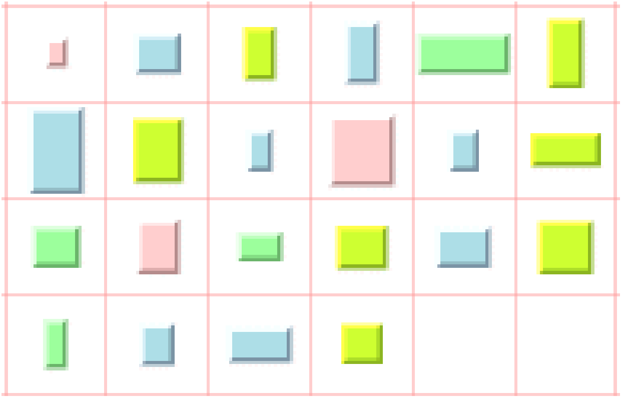
Specific Parameters

Presents the parameters of the grid layout algorithm.

Sample

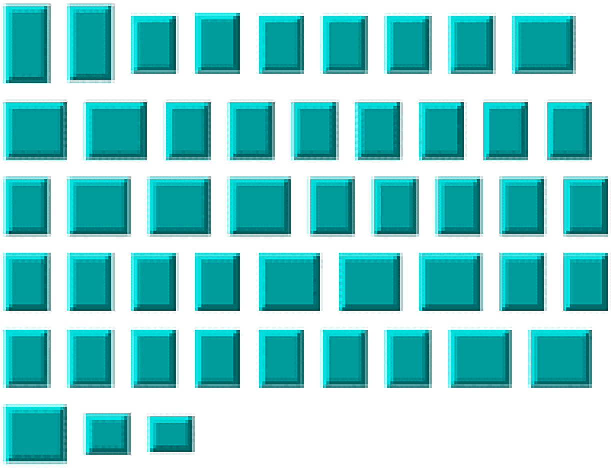
Here are sample drawings produced with the Grid Layout:

The following illustration shows a **TileToGridFixedWidth** mode with center horizontal and vertical alignment. The red lines are drawn here to help identify the grid cells; they are not drawn by the layout algorithm.



The following illustration shows a **TileToRows** mode with center vertical alignment

.



What Types of Graphs?

Any graph. However, the links are never taken into consideration. This algorithm is designed for placing nodes independently of their links, if they have any.

Application Domains

Any domain where a collection of isolated nodes needs to be laid out.

Features

- Arranges a collection of isolated nodes or connected components.
- Takes into account the size of the nodes so that no overlapping occurs.
- Provides several alignment options and dimensional parameters.
- Provides full support for fixed nodes (overlapping of nonfixed nodes with fixed nodes is avoided).
- Provides an incremental mode which helps the retention of a mental map on incremental changes made to a collection of nodes.

Brief Description of the Algorithm

The Grid Layout has two main modes: grid and row/column.

- In grid mode, the layout arranges the nodes of a graph in the cells of a grid (matrix). If a node is too large to fit in one grid cell (with margins), it occupies multiple cells. The size of the grid cells and the margins are parameters of the algorithm.
- In row/column mode, the layout arranges the nodes of a graph either by rows or by columns (according to the specified option). The width of the rows is controlled by the width of the layout region parameter. The height of the columns is controlled by the height of the layout region parameter. The horizontal and vertical margins between the nodes are parameters of the algorithm.

Code Sample

The following example shows how to use the GridLayout class. This code sample shows how to perform a Grid Layout on a graph.

```
using System;
using ILOG.Diagrammer;
using ILOG.Diagrammer.Graphic;
using ILOG.Diagrammer.GraphLayout;


…
Group group = new Group();

// Fill the group with nodes here

GridLayout layout = new GridLayout ();

group.GraphLayout = layout;
group.PerformGraphLayout();
Imports System
Imports ILOG.Diagrammer
Imports ILOG.Diagrammer.Graphic
Imports ILOG.Diagrammer.GraphLayout
```

```
Dim group As Group = New Group

' Fill the group with nodes here

Dim layout As GridLayout = New GridLayout
group.GraphLayout = layout
group.PerformGraphLayout()
```

Generic Features and Parameters

The GridLayout class supports the following generic parameters defined in the GraphLayout class (see *Layout Parameters and Features in the GraphLayout Base Class*):

- *Allowed Time*
- *Layout Region*
- *Preserve Fixed Nodes*
- *Stop Immediately*

The following comments describe the particular way in which these parameters are used by this subclass.

Allowed Time

The layout algorithm stops if the allowed time setting has elapsed. (For a description of this layout parameter in the GraphLayout class, see *Allowed Time*.) The result code in the layout report is **GraphLayoutReportCode.StoppedAndInvalid**.

Layout Region

The layout algorithm uses the layout region setting (either your own or the default setting) to control the size and the position of the graph drawing. All three ways to specify the layout region are available for this subclass. (See *Layout Region*.)

The layout region is considered differently depending on the layout mode. For details, see *Layout Modes*.

Preserve Fixed Nodes

The layout algorithm does not move the nodes that are specified as fixed. (See *Preserve Fixed Nodes*.) Moreover, nonfixed nodes are placed in such a manner that overlaps with fixed nodes are avoided.

Stop Immediately

The layout algorithm stops after cleanup if the method StopImmediately is called. (For a description of this method in the GraphLayout class, see *Stop Immediately*.) If the layout stops early because the allowed time has elapsed, the result code in the layout report is **GraphLayoutReportCode.StoppedAndInvalid**.

Specific Parameters

The following parameters are specific to the GridLayout class.

Order Parameter

The order parameter specifies how to arrange the nodes. To specify the ordering option for the nodes, you specify a **IComparer** in the NodeComparator property.

The valid values for comparator are:

- **GridLayout.AutomaticOrdering**

  The algorithm is free to choose the order in such a way that it tries to reduce the total area occupied by the layout.

- **GridLayout.NoOrdering**

  No ordering is performed.

- **GridLayout.DescendingHeight**

  The nodes are ordered in the descending order of their height.

- **GridLayout.AscendingHeight**

  The nodes are ordered in the ascending order of their height.

- **GridLayout.DescendingWidth**

  The nodes are ordered in the descending order of their width.

- **GridLayout.AscendingWidth**

  The nodes are ordered in the ascending order of their width.

- **GridLayout.DescendingArea**

  The nodes are ordered in the descending order of their area.

- **GridLayout.AscendingArea**

  The nodes are ordered in the ascending order of their area.

- **GridLayout.AscendingIndex**

  The nodes are ordered in the ascending order of their index (see GridLayout.SetIndex(object, int)).

- **GridLayout.DescendingIndex**

  The nodes are ordered in the descending order of their index (see GridLayout.SetIndex(object, int)).

- **null** (Nothing in Visual Basic)

The nodes are ordered in an arbitrary way.

- Any other implementation of **System.Collections.IComparer** interface.

  The nodes are ordered according to this custom comparer.

The default is **GridLayout.AutomaticOrdering**.

Note that in incremental mode (see IncrementalMode property) and with fixed nodes (see SetFixed(object, bool)), the order of the nodes is not completely preserved.

Note also that, if the layout mode is **TileToGridFixedWidth** or **TileToGridFixedHeight**, the order options are applied only for nodes whose size (including margins) is smaller than the grid cell size (see HorizontalGridOffset and VerticalGridOffset properties).

Layout Modes

The Grid Layout algorithm has four layout modes. To select a layout mode use the LayoutMode property:

The valid values for mode are:

- **GridLayoutMode.TileToGridFixedWidth** (the default).

  The nodes are placed in the cells of a grid (matrix) that has a fixed maximum number of columns. This number is equal to the width of the layout region parameter divided by the horizontal grid offset.

- **GridLayoutMode.TileToGridFixedHeight**

  The nodes are placed in the cells of a grid (matrix) that has a fixed maximum number of rows. This number is equal to the height of the layout region parameter divided by the vertical grid offset.

- **GridLayoutMode.TileToRows**

  The nodes are placed in rows. The maximum width of the rows is equal to the width of the layout region parameter. The height of the row is the maximum height of the nodes contained in the row (plus margins).

- **GridLayoutMode.TileToColumns**

  The nodes are placed in columns. The maximum height of the columns is equal to the height of the layout region parameter. The width of the column is the maximum width of the nodes contained in the column (plus margins).

Alignment Parameters

The alignment options control how a node is placed over its grid cell or over its row or column (depending on the layout mode). The alignment can be set globally for all nodes, in which case all nodes are aligned in the same way, or locally on each node, with the result that different alignments occur in the same drawing.

Global Alignment Parameters

To set the horizontal alignment for all nodes use the HorizontalAlignment property:

The valid values for the horizontal alignment are defined by the GridLayoutHorizontalAlignment enumeration:

- **GridLayoutHorizontalAlignment.Center** (the default)

  The node is horizontally centered over its grid cell or row or column.

- **GridLayoutHorizontalAlignment.Left**

  The node is horizontally aligned on the left of its grid cell(s) or column. Not used if the layout mode is **GridLayoutMode.TileToRows**.

- **GridLayoutHorizontalAlignment.Right**

  The node is horizontally aligned on the right of its grid cell(s) or column. Not used if the layout mode is **GridLayoutMode.TileToRows**.

- **GridLayoutHorizontalAlignment.Mixed**

  Each node can have a different alignment. The alignment of each individual node can be set with the result that different alignments can occur in the same graph.

To set the vertical alignment for all nodes usre the VerticalAlignment property: The valid values for the vertical alignment are defined by the enumeration GridLayoutVerticalAlignment:

- **GridLayoutVerticalAlignment.Center** (the default)

  The node is vertically centered over its grid cell or row or column.

- **GridLayoutVerticalAlignment.Top**

  The node is vertically aligned on the top of its cell(s) or row. Not used if the layout mode is **GridLayoutMode.TileToColumns**.

- **GridLayoutVerticalAlignment.Bottom**

  The node is vertically aligned on the bottom of its grid cell(s) or row. Not used if the layout mode is **GridLayoutMode.TileToColumns**.

- **GridLayoutVerticalAlignment.Mixed**

  Each node can have a different alignment. The alignment of each individual node can be set with the result that different alignments can occur in the same graph.

Alignment of Individual Nodes

All nodes have the same alignment unless the vertical or horizontal alignments are set to **Mixed**. Only when the global alignment is set to mixed can each node have an individual alignment style.

To set and retrieve the alignments of an individual node, use the following methods:

```
void SetHorizontalAlignment(object node, GridLayoutHorizontalAlignment
alignment);
void SetVerticalAlignment(object node, GridLayoutVerticalAlignment alignment);
GridLayoutHorizontalAlignment GetHorizontalAlignment(object node);
GridLayoutVerticalAlignment GetVerticalAlignment(object node);
```

Maximum Number of Nodes Per Row or Column

By default, in **GridLayoutMode.TileToRows** or **GridLayoutMode.TileToColumns** mode, the layout places as many nodes on each row or column as possible given the size of the nodes and the dimensional parameters (layout region and margins). If needed, the layout can additionally respect a specified maximum number of nodes per row or column.

To set the maximum number of nodes per row or column use the **MaxNumberOfNodesPerRowOrColumn** property.

The default value is **In32.MaxValue**, that is, the number of nodes placed in each row or column is bounded only by the size of the nodes and the dimensional parameters. The specified value must be at least 1. The property has no effect if the layout mode is **GridLayoutMode.TileToGridFixedWidth** or **GridLayoutMode.TileToGridFixedHeight**.

Incremental Mode

The Grid Layout algorithm normally places all the nodes from scratch. If the graph incrementally changes because you add, remove, or resize nodes, the subsequent layout may differ considerably from the previous layout. To avoid this effect and to help the user to retain a mental map of the graph, the algorithm has an incremental mode. In incremental mode, the layout tries to place the nodes at the same location or in the same order as in the previous layout whenever it is possible.

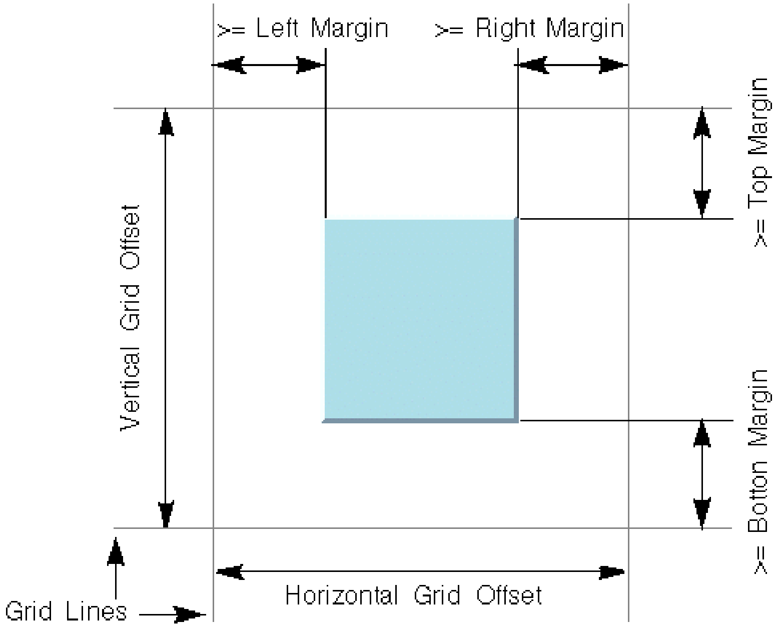
To enable the incremental mode set the IncrementalMode property to **true**.

To preserve the stability, the incremental mode may keep some regions free. Therefore, the total area of the layout may be larger than in nonincremental mode, and, in general, the layout may not look as nice as in nonincremental mode.

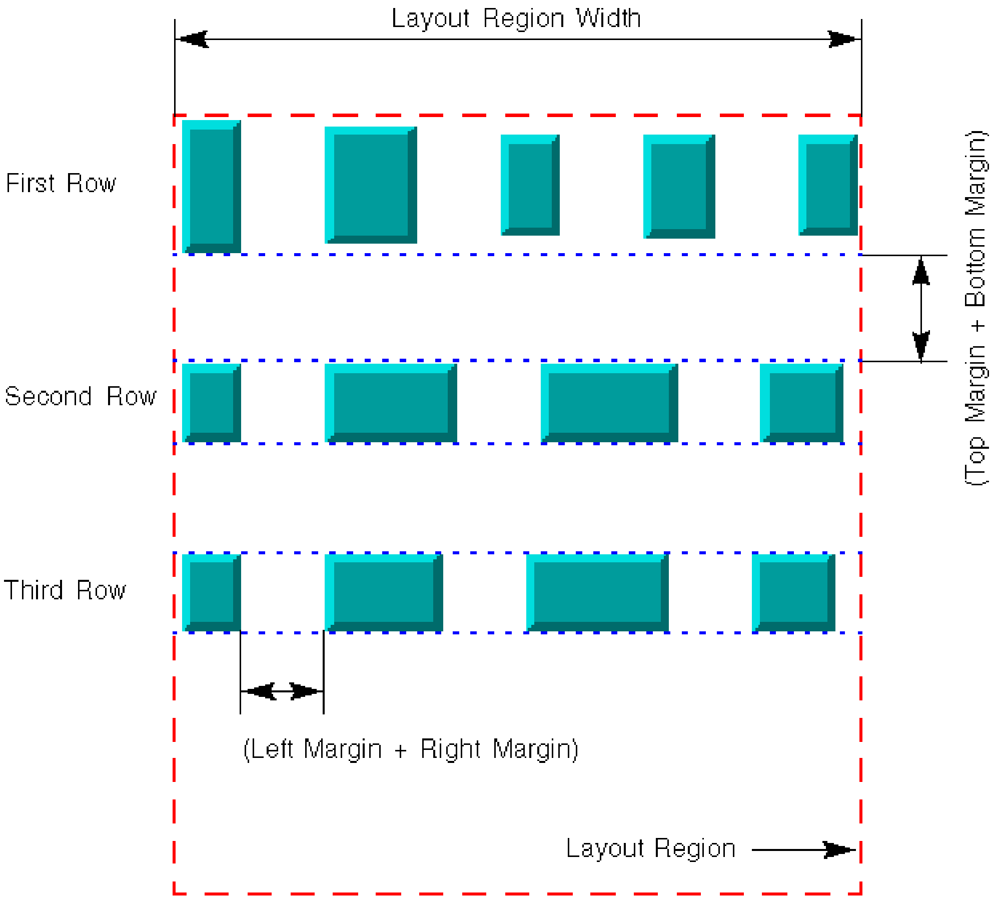
Dimensional Parameters

The following illustrations explains the dimensional parameters used in the Grid Layout algorithm. These parameters are explained in the sections that follow.

The following illustration shows dimensional parameters for the grid mode of the Grid Layout algorithm.



The following illustration shows dimensional parameters for the row/column mode of the Grid Layout algorithm.



Grid Offset

The grid offset parameters control the spacing between grid lines. It is taken into account only by the grid mode (layout modes **TileToGridFixedWidth** and **TileToGridFixedHeight**). To set the horizontal and vertical grid offset use the HorizontalGridOffset and VerticalGridOffset properties.

The grid offset is the critical parameter for the grid mode. If the grid offset is larger than the size of the nodes (plus margins), an empty space is left around the node. If the grid offset is smaller than the size of the nodes (plus margins), the node will need to be placed on more than one grid cell. The best choice for the grid offsets depends on the application. It can be computed according to either the maximum size of the nodes (plus margins) or the medium size, and so on. Of course, if all the nodes have a similar size, the choice is straight-forward.

Margins

The margins control the space around each node that the layout algorithm keeps empty. To set the margins use the **TopMargin**, **BottomMargin**, **LeftMargin** and **RightMargin** properties.

The meaning of the margin parameters is not the same for the grid modes as for the row/ column modes:

- In grid modes, they represent the minimum distance between the node border and the grid line
- In row/column modes, they are used to control the vertical distance between the rows or the horizontal distance between the columns and the horizontal or vertical minimal distance between the nodes in the same row or column.

The default value for all the margin parameters is 5.

Random Layout

This section describes the Random Layout algorithm (class RandomLayout).

In This Section

Sample

Provides some sample drawings produced by the algorithm.

What Types of Graphs?

Explains the type of graph on which you can use the algorithm.

Features

Lists the features of the algorithm.

Limitations

Explains the limitations of the algorithm.

Brief Description of the Algorithm

Provides a short description of the random layout algorithm.

Code Sample

Provides a sample of code that shows how to use the random layout algorithm.

Generic Features and Parameters

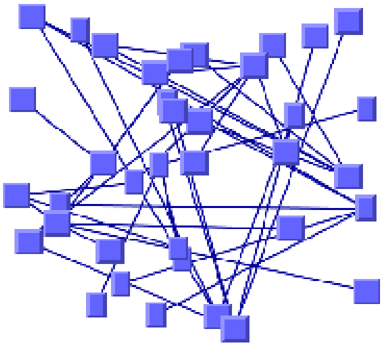
Explains what generic features of the graph layout library are supported and how.

Specific Parameters

Presents the parameters of the random layout algorithm.

Sample

Here is a sample drawing produced with the Random Layout:



What Types of Graphs?

Any type of graph:

- Connected and disconnected graphs
- Planar and nonplanar graphs

Features

Random placement of the nodes of a graph inside a given region.

Limitations

- The algorithm computes random coordinates for the upper-left corner of the graphic objects representing the nodes. In some cases, this may not be appropriate.
- To ensure that the nodes do not overlap the margins of the layout region, the algorithm computes the coordinates randomly inside a region whose width and height are smaller than the width and height of the layout region. The difference is the maximum width and the maximum height of the nodes, respectively. In some cases, this may not be appropriate.

Brief Description of the Algorithm

The Random Layout algorithm is not really a layout algorithm. It simply places the nodes at randomly computed positions inside a user-defined region. Nevertheless, the Random

Layout algorithm may be useful when a random, initial placement is needed by another layout algorithm or in cases where an aesthetic, readable drawing is not important.

Code Sample

The following example uses the RandomLayout class and shows how to perform a Random Layout on a graph.

```
using System;
using ILOG.Diagrammer;
using ILOG.Diagrammer.Graphic;
using ILOG.Diagrammer.GraphLayout;


…
Group group = new Group();

// Fill the group with nodes and links here

RandomLayout layout = new RandomLayout();

group.GraphLayout = layout;
group.PerformGraphLayout();
Imports System
Imports ILOG.Diagrammer
Imports ILOG.Diagrammer.Graphic
Imports ILOG.Diagrammer.GraphLayout

Dim group As Group = New Group

' Fill the group with nodes and links here

Dim layout As RandomLayout = New RandomLayout

group.GraphLayout = layout
group.PerformGraphLayout()
```

Generic Features and Parameters

The RandomLayout class supports the following generic parameters defined in the GraphLayout class (see *Layout Parameters and Features in the GraphLayout Base Class*):

- *Layout Region*
- *Percentage of Completion Calculation*
- *Preserve Fixed Links*
- *Preserve Fixed Nodes*
- *Random Generator Seed Value*
- *Stop Immediately*

The following comments describe the particular way in which these parameters are used by this subclass.

Layout Region

The layout algorithm uses the layout region setting (either your own or the default setting) to control the size and the position of the graph drawing. All three ways to specify the layout region are available for this subclass. (See *Layout Region*.)

Percentage of Completion Calculation

The layout algorithm calculates the estimated percentage of completion. This value can be obtained from the layout report during the run of the layout. (For a detailed description of this features, see *Percentage of Completion Calculation* and *Graph Layout Event Handlers*.)

Preserve Fixed Links

The layout algorithm does not reshape the links that are specified as fixed. (See *Preserve Fixed Links*.)

Preserve Fixed Nodes

The layout algorithm does not move the nodes that are specified as fixed. (See *Preserve Fixed Nodes*.)

Random Generator Seed Value

The Random Layout uses a random number generator to compute the coordinates. You can specify a particular value to be used as a *seed value*. (See *Random Generator Seed Value*.) For the default behavior, the random generator is initialized using the current system clock. Therefore, different layouts are obtained if you perform the layout repeatedly on the same graph.

Stop Immediately

The layout algorithm stops after cleanup if the method StopImmediately is called. (For a description of this method in the **GraphLayout** class, see *Stop Immediately*.) If the layout stops early because the allowed time has elapsed, the result code in the layout report is **GraphLayoutReportCode.StoppedAndInvalid**.

Specific Parameters

The following parameters are specific to the RandomLayout class.

Link Style

When the layout algorithm moves the nodes, straight-line links will automatically follow the new positions of their end nodes. If the graph contains other types of links (for example, a Link object with ShapeType Orthogonal), the shape of the link may not be appropriate because the intermediate points of the link will not be moved. In this case, you can ask the layout algorithm to automatically remove all the intermediate points of the links (if any). To do this, use the LinkStyle property:

The valid values for style are:

- **RandomLayoutLinkStyle.NoReshape**

  None of the links is reshaped in any manner.

- **RandomLayoutLinkStyle.StraightLine**

  All the intermediate points of the links (if any) are removed. This is the default value.

Performing Layout on Nested Graphs

This section explains how to define and perform graph layout algorithms on nested graphs.

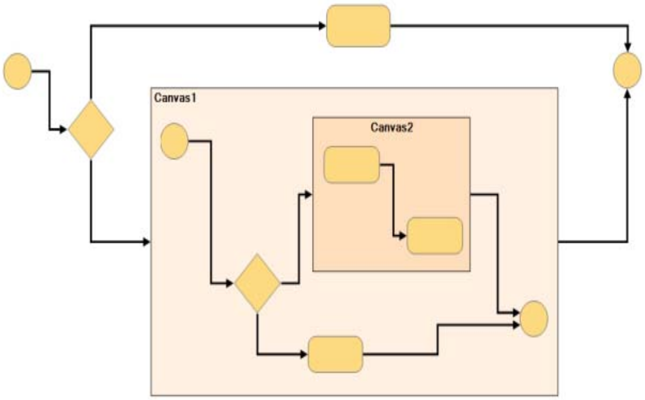
What are Nested Graphs?

A nested graph (also called a subgraph) is a graph contained in another graph. The subgraph is considered as a node of its parent graph (that is, it can have links connected to it), and it also contains nodes and links.

In IBM® ILOG® Diagram for .NET, toplevel graphs are usually represented by Group objects, and nested graphs are represented by Canvas objects contained in a toplevel **Group** or in another **Canvas**.

> ***Note: Group*** *objects contained in a another* ***Group*** *or in a* ***Canvas*** *are not considered as subgraphs by the graph layout algorithms. The children of a subgroup are laid out as if they were direct children of the parent of the group.*

The following picture shows a **Group** containing a **Canvas** (named Canvas1) that contains another **Canvas** (named Canvas2).



What is Recursive Layout?

graph layout algorithms can be performed recursively on a graph that contains subgraphs. This recursive layout is performed as follows:

- The layout is performed on each nested graph in depth first order, that is, on a subgraph first, then on its parent graph.
- After the layout has been performed on a subgraph, the size of the subgraph (that is, the Canvas object) is adjusted to fit the size of its children.
- When the layout is performed on the parent graph, the subgraph is moved as a whole.

Performing the Same Layout on All Nested Graphs

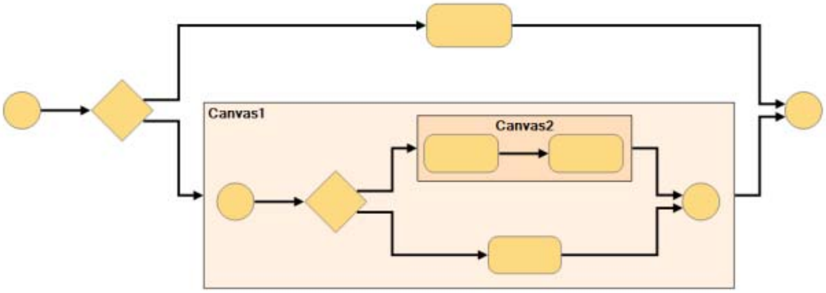
If you want to perform the same graph layout algorithm on a toplevel graph and all its subgraphs, do the following:

1. Set the GraphLayout and/or LinkLayout properties of the toplevel Group to the desired graph layout.
2. Make sure the LayoutRecursively property is **true**.
3. Call the PerformGraphLayout method of the toplevel group.

The following example shows how to perform a left-to-right hierarchical layout on a graph and its subgraphs.

```
HierarchicalLayout hl = new HierarchicalLayout();
hl.LinkStyle = HierarchicalLayoutLinkStyle.Orthogonal;
hl.Position = new Point2D(20, 100);
group.GraphLayout = hl;
group.LayoutRecursively = true;
group.PerformGraphLayout();
Dim hl As HierarchicalLayout = New HierarchicalLayout
hl.LinkStyle = HierarchicalLayoutLinkStyle.Orthogonal
hl.Position = New Point2D(20, 100)
group.GraphLayout = hl
group.LayoutRecursively = True
group.PerformGraphLayout()
```

The following picture shows the resulting layout.



Specifying a Different Layout on a Subgraph

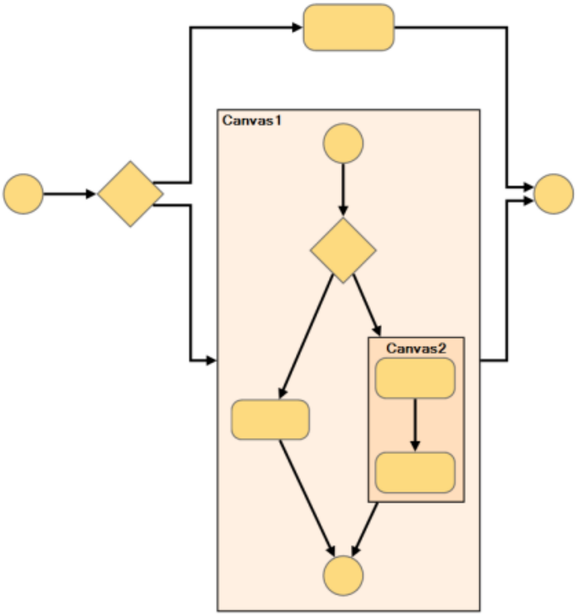
If you want to perform a different layout on one of the subgraphs, you must set the GraphLayout property of the subgraph to the layout algorithm for the subgraph.

The following example shows how to perform a top-to-bottom on the Canvas1 subgraph only.

```
HierarchicalLayout hl = new HierarchicalLayout();
hl.LinkStyle = HierarchicalLayoutLinkStyle.Orthogonal;
hl.Position = new Point2D(110, 5);
group.GraphLayout = hl;
Canvas canvas1 = ((Diagram1)group).Canvas1;
hl = new HierarchicalLayout();
hl.FlowDirection = LayoutFlowDirection.Bottom;
```

```
canvas1.GraphLayout = hl;
group.LayoutRecursively = true;
group.PerformGraphLayout();
Dim hl As HierarchicalLayout = New HierarchicalLayout
hl.LinkStyle = HierarchicalLayoutLinkStyle.Orthogonal
hl.Position = New Point2D(110, 5)
group.GraphLayout = hl
Dim canvas1 As Canvas = CType(group, Diagram1).Canvas1
hl = New HierarchicalLayout
hl.FlowDirection = LayoutFlowDirection.Bottom
canvas1.GraphLayout = hl
group.LayoutRecursively = True
group.PerformGraphLayout()
```

The following picture shows the resulting layout. Note that the Canvas2 subgraph inherits the top-to-bottom layout from its parent Canvas1.



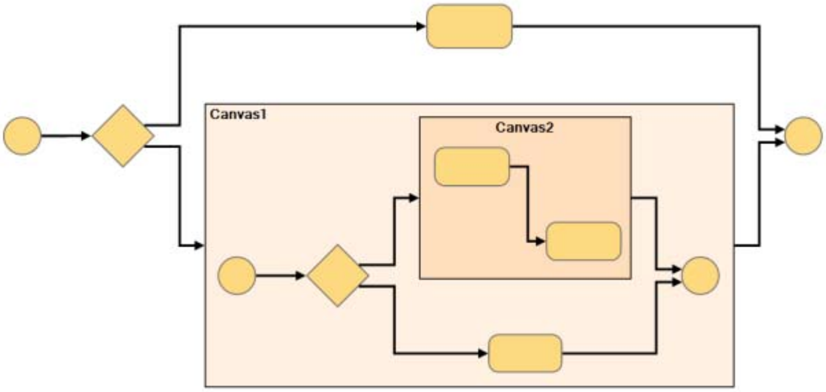
Preventing the Layout on a Subgraph

It you want the layout not to be performed on a subgraph, you must set its GraphLayoutActive property to **false**.

The following example shows how to prevent the layout from being performed on the Canvas2 subgraph.

```
HierarchicalLayout hl = new HierarchicalLayout();
hl.LinkStyle = HierarchicalLayoutLinkStyle.Orthogonal;
hl.Position = new Point2D(20, 60);
group.GraphLayout = hl;
Canvas canvas2 = ((Diagram1)group).Canvas2;
canvas2.GraphLayoutActive = false;
group.LayoutRecursively = true;
group.PerformGraphLayout();
Dim hl As HierarchicalLayout = New HierarchicalLayout
hl.LinkStyle = HierarchicalLayoutLinkStyle.Orthogonal
hl.Position = New Point2D(20, 60)
group.GraphLayout = hl
Dim canvas2 As Canvas = CType(group, Diagram1).Canvas2
canvas2.GraphLayoutActive = False
group.LayoutRecursively = True
group.PerformGraphLayout()
```

The following picture shows the resulting layout. Note that the children of Canvas2 have kept their initial relative positions.



Advanced API for Recursive Layout

The properties and methods of the GraphicContainer explained above let you perform and customize recursive graph layouts in most cases.

For more advanced needs, you can also use the following classes of the ILOG.Diagrammer.GraphLayout namespace that provide more options to customize and perform recursive layouts:

- RecursiveLayout
- MultipleLayout

- RecursiveMultipleLayout

Using Advanced Features

This section describes advanced features of IBM® ILOG® Diagram for .NET graph layout algorithms.

In This Section

Explains how to specify in which geometrical space the layout must be performed.

Releasing Resources Used During the Layout

Describes how to release resources that are used during the layout.

Defining Your Own Type of Layout

Explains how to define your own type of layout.

Related Documentation

Lists the books dedicated to graph layout that have been published.

Questions and Answers about Using the Layout Algorithms

Provides some helpful suggestions for using the layout algorithms.

Using a Graph Layout Report

Graph layout reports are objects used to store information about the particular behavior of a layout algorithm. After the layout is completed, this information is available to be read from the layout report.

Layout Report Classes

Each layout class instantiates a particular class of **ILOG.Diagrammer.GraphLayoutReport** each time the layout is performed. Layout Report Classes shows the layout classes and their corresponding layout reports.

Layout Class	**Layout Report Class**
HierarchicalLayout	GraphLayoutReport
TreeLayout	**GraphLayoutReport**
ForceDirectedLayout	ForceDirectedLayoutReport
GridLayout	**GraphLayoutReport**
RandomLayout	**GraphLayoutReport**
ShortLinkLayout	**GraphLayoutReport**
LongLinkLayout	**GraphLayoutReport**
MultipleLayout	MultipleLayoutReport
RecursiveLayout	RecursiveLayoutReport
RecursiveMultipleLayout	**RecursiveLayoutReport**

Creating a Layout Report

All layout classes inherit the PerformLayout method from the GraphLayout class. This method calls CreateLayoutReport to obtain a new instance of the layout report. This instance is returned when **PerformLayout** returns. The default implementation in the base layout class creates an instance of GraphLayoutReport. Some subclasses override this method to return an appropriate subclass. Other classes do not need specific information to be stored in the layout report and do not override **CreateLayoutReport**. In this case, the base class **GraphLayoutReport** is used.

When using the layout classes provided with IBM ILOG Diagram for .NET, you do not need to instantiate the layout report yourself. This is done automatically.

Accessing a Layout Report

If you do not use a diagram component, you usually call layout via the method **PerformLayout** which returns the layout report. The following example shows how to read the information from the layout report in this case:

```
 ...
try {
        GraphLayoutReport layoutReport = layout.PerformLayout();
        if (layoutReport.getCode() ==
                                   GraphLayoutReportCode.LayoutDone)
                System.Console.Writeln("Layout done.");
        else
                System.Console.Writeln("Layout not done, code = " +
                                        layoutReport.Code);
}
catch (GraphLayoutException ex) {
        System.Console.Writeln(ex.ToString());
}
...
Try
 Dim layoutReport As GraphLayoutReport = layout.PerformLayout()
 If layoutReport.getCode = GraphLayoutReportCode.LayoutDone Then
   System.Console.Writeln("Layout done.")
 Else
   System.Console.Writeln("Layout not done, code = " + layoutReport.Code)
 End If
Catch ex As GraphLayoutException
 System.Console.Writeln(ex.ToString)
End Try
```

Information Stored in a Layout Report

The base class **GraphLayoutReport** stores the following information:

- Code
- Layout Time

- Percentage of Completion
- Additional Information

Code

The read-only property Code contains information about special, predefined cases that may have occurred during the layout. The possible values of the GraphLayoutReportCode are the following:

- **GraphLayoutReportCode.LayoutDone** appears if the layout was performed successfully.
- **GraphLayoutReportCode.StoppedAndValid** appears if the layout was performed but was stopped before completion, either because the layout time elapsed or because the method StopImmediately was called. The positions of nodes and links are valid at the stopping point because the layout algorithm uses an iterative mechanism.
- **GraphLayoutReportCode.StoppedAndInvalid** appears if a (noniterative) layout was performed but was stopped before completion, either because the layout time elapsed or because the method **StopImmediately** was called. The positions of nodes and links are not valid at the stopping point. Often, they have not yet been changed at all.
- **GraphLayoutReportCode.NotNeeded** appears if the layout was not performed because no changes occurred in the graph and parameters since the last time the layout was performed successfully.
- **GraphLayoutReportCode.EmptyGraph** appears if the graph is empty.

Layout Time

The read-only property LayoutTime contains the total duration (in milliseconds) of the layout algorithm at the end of the layout.

Percentage of Completion

The read-only property PercentageComplete contains an estimation of the percentage of the layout that has been completed. This can be used if the layout algorithm supports the generic percentage completion calculation feature. See *Percentage of Completion Calculation*. It is typically used inside layout event handlers that are described in the following section.

Additional Information

Additional information for particular layout algorithms is stored by the subclasses of **GraphLayoutReport**. For details, see the API Reference Manual for these classes:

- ForceDirectedLayoutReport
- MultipleLayoutReport
- RecursiveLayoutReport

Using Event Handlers

All layout classes support two kinds of events: layout events and parameter events. The API therefore provides:

- *Graph Layout Event Handlers*
- *Property Changed Event Handlers*

Graph Layout Event Handlers

The layout event mechanism provides a way to inform the end user of what is happening during the layout. At times, a layout algorithm may take a long time to execute, especially when dealing with large graphs. In addition, an algorithm may not converge in some cases. No matter what the situation, the end user should be informed of the events that occur during the layout. This can be done by implementing a simple progress bar or by displaying appropriate information, such as the percentage of completion after each iteration or step.

The layout event handling is done by attaching a GraphLayoutStepPerformedEventHandler to the LayoutStepPerformed event and implementing the expected behavior in the event handler. In this way, you can, for example, read information about the current state of the layout report (see *Using a Graph Layout Report*) after each iteration or step of the layout algorithm:

```
void CreateLayout() {
  HierarchicalLayout layout = new HierarchicalLayout();
  ...
  layout.LayoutStepPerformed +=
     new  GraphLayoutStepPerformedEventHandler(OnLayoutStepPerformed);
  ...
}

void OnLayoutStepPerformed(object sender,
                           GraphLayoutStepPerformedEventArgs e)
{
  GraphLayoutReport layoutReport = e.LayoutReport;
  System.Console.Writeln("Percentage of completion: " +
                       layoutReport.PercentageComplete);
}
Sub CreateLayout()
 Dim layout As HierarchicalLayout = New HierarchicalLayout
 ...
 AddHandler layout.LayoutStepPerformed, AddressOf OnLayoutStepPerformed
 ...
End Sub

Sub OnLayoutStepPerformed(ByVal sender As Object, _
                          ByVal e As GraphLayoutStepPerformedEventArgs)
 Dim layoutReport As GraphLayoutReport = e.LayoutReport
 System.Console.Writeln("Percentage of completion: " + _
                        layoutReport.PercentageComplete)
End Sub
```

Property Changed Event Handlers

The layout parameter event mechanism provides a way to inform the end user that any layout property has changed. This is useful, for instance, when the layout property values are displayed in a dialog box that needs to be updated to indicate property changes.

The property event handling is done by attaching a GraphLayoutPropertyChangedEventHandler to the PropertyChanged event and implementing the expected behavior in the event handler.

When you implement the **GraphLayoutPropertyChangedEventHandler**, you have access to the GraphLayoutPropertyChangedEventArgs. This contains a flag accessible by the property ParametersUpToDate:

- It returns **true** if the event occurs at the end of a run of the layout when the layout is considered up-to-date with respect to the layout properties.
- It returns **false** if the event occurs when any layout property has changed.

The **GraphLayoutPropertyChangedEventArgs** also gives access to the name of the property that has changed: ParameterName and to the layout instance to which the changed property belongs: GraphLayout. The following example shows how to implement and register a layout property event handler.

```
void CreateLayout() {
HierarchicalLayout layout = new HierarchicalLayout();
...
layout.PropertyChanged += new
GraphLayoutPropertyChangedEventHandler(OnPropertyChanged);
...
}

void OnPropertyChanged(object sender,
                       GraphLayoutPropertyChangedEventArgs e)
{
System.Console.Writeln("Property: " + e.ParameterName +
                       " changed on : " + e.GraphLayout);
System.Console.Writeln("Parameters up to date: " + e.ParametersUpToDate);
}
Sub CreateLayout()
 Dim layout As HierarchicalLayout = New HierarchicalLayout
 ...
 AddHandler layout.PropertyChanged, AddressOf OnPropertyChanged
 ...
End Sub

Sub OnPropertyChanged(ByVal sender As Object, _
                      ByVal e As GraphLayoutPropertyChangedEventArgs)
  System.Console.Writeln("Property: " + e.ParameterName + _
                       " changed on: " + e.GraphLayout)
  System.Console.Writeln("Properties up to date: " + e.ParametersUpToDate)
End Sub
```

Using the Graph Model

The interface IGraphModel defines a suitable, generic API for graphs that have to be laid out with IBM® ILOG® Diagram for .NET graph layout algorithms.

All the layout algorithms provided in IBM ILOG Diagram for .NET are designed to lay out a graph model. This allows applications to benefit from the graph layout algorithms whether or not they use the IBM ILOG Diagram for .NET graph. However, to make things very simple for the common case of applications that use a GraphicContainer, it is not mandatory to work directly with the graph model except for some advanced features such as filtering (see *Using the Filtering Features to Lay Out a Part of a Graph*).

The Graph Model Concept

With a graph model, you can use already-built graphs, nodes, and links that have been developed with or without IBM ILOG Diagram for .NET and apply the layout algorithms of IBM ILOG Diagram for .NET. The graph model defines the basic, generic operations for performing the layout. Instead of using a concrete graph class such as **GraphicContainer** directly, the layout algorithms interact with the graph via the graph model. This is the key for achieving a truly generic graph layout framework.

A concrete implementation of the graph model must be written to adapt the graph model to specific graph, node, and link objects. This plays the role of an "adapter" or bridge between the application objects and the graph model. This architecture makes it much easier to add graph features to existing applications.

> ***Note:*** *If an application uses the IBM ILOG Diagram for .NETgraphic objects (**GraphicContainer**, GraphicObject, and so on), the graph can be attached directly to the layout instance without explicitly using a graph model. (See the method Attach(GraphicContainer).) In this case, the appropriate adapter (GraphicContainerAdapter) will be created internally. This adapter can be retrieved using the method GetGraphModel, which will return an instance of **GraphicContainerAdapter**.*

The IGraphModel Interface

The methods defined by **IGraphModel** can be divided into several categories that provide information on the structure of the graph, the geometry of the graph, modification of the graph geometry, and notification of changes in the graph.

This section is divided as follows:

- *Information on the Structure of the Graph*
- *Information on the Geometry of the Graph*

- *Modification of the Geometry of the Graph*
- *Notification of Changes*
- *Storing and Retrieving Data Objects ("Properties")*

Information on the Structure of the Graph

The following properties and methods defined by **IGraphModel** allow the layout algorithms to retrieve information on the structure of the graph:

- ICollection Nodes
- ICollection Links
- bool IsNode(Object obj)
- bool IsLink(Object obj)
- ICollection GetLinksFrom(Object node)
- ICollection GetLinksTo(Object node)
- Object GetFrom(Object link)
- Object GetTo(Object link)

The following properties and methods are provided for use with nested graphs (see also *Performing Layout on Nested Graphs*):

- IGraphModel Parent
- **IGraphModel** Root
- IGraphModel GetGraphModel(Object subgraph)
- ICollection Subgraphs
- bool IsSubgraph(Object obj)
- ICollection InterGraphLinks
- bool IsInterGraphLink(Object obj)

Information on the Geometry of the Graph

The following methods defined by **IGraphModel** allow the layout algorithms to retrieve information on the geometry of the graph:

- Rectangle2D BoundingBox(Object nodeOrLink)
- Point2D[] GetLinkPoints(Object link)
- float GetLinkWidth(Object link)

The **BoundingBox** method is called by a layout algorithm whenever it needs to get the position and the dimensions of a node or a link. The other methods are used mainly by link layout algorithms.

Modification of the Geometry of the Graph

The following methods defined by **IGraphModel** allow a layout algorithm to modify the geometry of the graph:

- void MoveNode(Object node, float x, float y)
- void ReshapeLink(Object link, ReshapeLinkStyle style, Point2D fromPoint, ReshapeLinkMode fromPointMode, Point2D[] points, int startIndex, int length, Point2D toPoint, ReshapeLinkMode toPointMode)

Layout algorithms that compute new coordinates for the nodes use the **MoveNode** method. Layout algorithms that compute new shapes for the links call the **ReshapeLink** method.

Notification of Changes

The following event defined by **IGraphModel** allows a layout algorithm to be notified of changes in the graph:

```
event GraphModelContentsChangedEventHandler ContentsChanged
```

A "change" in the graph can be a structure change (that is, a node or a link was added or removed) or a geometry change (that is, a node or a link was moved or reshaped). The graph model event mechanism provides a means to keep the layout algorithms informed of these changes. When the layout algorithm is restarted on the same graph, it is able to detect whether the graph has changed since the last time the layout was successfully performed. If necessary, the layout can be performed again. If there is no change in the graph, the layout algorithm can avoid unnecessary work by not performing the layout. To know whether the previous layout is still valid or it must be redone, the layout algorithms call the following method of the model:

```
bool IsLayoutNeeded(GraphLayout layout)
```

> ***Note:*** *The creation of the graph model event handler is done transparently by the GraphLayout class. Therefore, there is usually no need to manipulate this handler directly.*

Storing and Retrieving Data Objects ("Properties")

The following methods defined by **IGraphModel** allow a layout algorithm to store data objects for each node, link, or graph:

- void SetProperty(Object nodeOrLink, String key, Object value)
- Object GetProperty(Object nodeOrLink, String key)

- void SetProperty(String key, Object value)
- Object GetProperty(String key)

The layout algorithm may need to associate a set of properties with the nodes and links of the graph or with the graph itself. Properties are a set of key-value pairs, where the key is a **String** object and the value can be any kind of information value.

> ***Note:*** *Creating a property and associating it with a node, a link, or a graph is handled transparently by the layout algorithm whenever it is necessary. Therefore, there is usually no need to manipulate the properties directly. However, if needed, you can do this in your own subclass of* ***GraphLayout****.*

The AbstractGraphModel class

The AbstractGraphModel class provides a concrete implementation of some of the properties and methods defined by **IGraphModel**. This class can be used as base class for a concrete implementation of **IGraphModel**. It encapsulates some services that are common to all graph models.

Using the Class GraphicContainerAdapter

The GraphicContainerAdapter class is a concrete subclass of **AbstractGraphModel** that allows a GraphicContainer to be laid out using the layout algorithms provided in IBM ILOG Diagram for .NET. It provides an implementation for all the abstract methods. It also provides an overridden implementation of some nonabstract methods of **AbstractGraphModel** to improve efficiency by taking advantage of the characteristics of the **GraphicContainer**.

If an application uses the **GraphicContainer** class, the graph can be attached directly to the layout instance without explicitly using the adapter. See the method Attach(GraphicContainer). In this case, a **GraphicContainerAdapter** is created internally by the layout class. The adapter can be retrieved using the method GetGraphModel, which will return an instance of **GraphicContainerAdapter**.

Notice that such an internally created adapter is not allowed to be attached to any other layout instance, nor to be used in any way once the method Detach has been called on the layout instance.

In case you need to be able to do any of the above operations, directly create the instance of **GraphicContainerAdapter** and attach it using **Attach(GraphModel)**.

To know whether a given **GraphModel** instance has been created using **Attach(GraphicContainer)**, you can use the method getOriginatingLayout. This method returns a non-null object if the model has not been created using **Attach(GraphicContainer)**.

Additionally, the **GraphicContainerAdapter** class provides a way to filter the **GraphicContainer**. By using the filtering mechanism, you specify a particular set of nodes and links that have to be taken into account by the layout algorithm. (See *Choosing the Layout Coordinate Space*.)

The **GraphicContainerAdapter** class allows you to specify the order of nodes, as considered by its property Nodes. For this purpose, you can provide your own implementation of **System.Collections.IComparer** to define the order of the nodes. Then, specify this comparer by using the property NodeComparer.

The **GraphicContainerAdapter** class also allows you to specify the Transformer that has to be used for computing the geometry of the graph. (See *Choosing the Layout Coordinate Space*)

> ***Note:*** *For details on how to write your own adapter, see Laying Out Third-Party Graphs.*

Laying Out Third-Party Graphs

To understand this section better, see*Using the Graph Model*.

It is sometimes necessary to add graph layout features to an existing application. If the application already uses the IBM® ILOG® Diagram for .NET graph (GraphicContainer) to manipulate and display graphs, using the graph layout algorithms provided in IBM ILOG Diagram for .NET is a straightforward process. No adapter has to be written.

However, the case may arise where an application uses its own classes for nodes, links, and graphs, and where, for some reason, you do not want to replace these classes with IBM ILOG Diagram for .NET classes. To enable the graph layout algorithms to work with these graph objects, a custom adapter must be written.

The adapter must implement the interface IGraphModel. As a convenience, it can extend the provided AbstractGraphModel and only implement its abstract methods. The nonabstract methods of **AbstractGraphModel** have a default implementation that is really functional. However, they may not be optimal because they do not take advantage of the characteristics of the underlying graph implementation. For better performance, the following nonabstract methods can be overridden in the adapter class, to take advantage from a direct storage of the property object in the nodes and links (assuming this feature is available):

void SetProperty(Object nodeOrLink, String key, Object value)

Object GetProperty(Object nodeOrLink, String key)

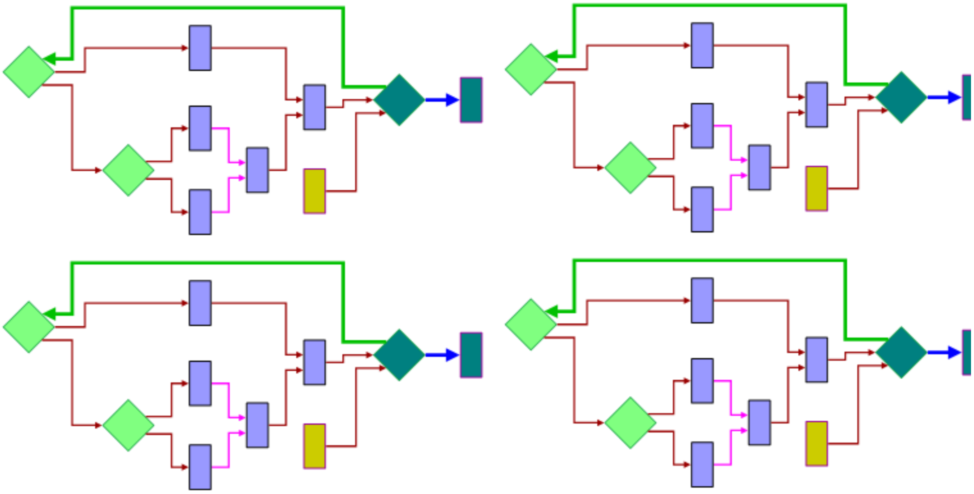
void SetProperty(String key, Object value)

Object GetProperty(String key)

The efficiency of the layout algorithm depends directly on the efficiency of the implementation of the adapter class and the underlying graph data structure.

Laying Out Connected Components of a Disconnected Graph

IBM® ILOG® Diagram for .NET provides special support for the layout of a *disconnected graph*. If a graph is composed of several connected components or contains isolated nodes (nodes without any links), it can be desirable to apply the layout algorithm separately on each connected component and then to position the connected components using a specialized layout algorithm (usually, GridLayout). The following illustration shows an example of a graph containing four connected components. Simply by enabling the layout of the connected components on the regular layout instance (here, HierarchicalLayout), the connected components are automatically identified and laid out individually. Finally, the four connected components are positioned using a highly customizable placement algorithm (**GridLayout**).



To indicate whether a subclass of GraphLayout supports this feature, use the method SupportsLayoutOfConnectedComponents.

The default implementation returns **false**. A subclass can override this method in order to return **true** to indicate that this feature is supported.

IBM ILOG Diagram for .NET allows you to enable the layout of the connected components using the property LayoutOfConnectedComponentsEnabled.

The default value is the value returned by the method IsLayoutOfConnectedComponentsEnabledByDefault.

The default implementation of this method in **GraphLayout** returns **false**. For some of the layout classes, it is appropriate that this feature is enabled by default. Therefore, ForceDirectedLayout overrides this method to return **true**.

If enabled on a layout class that supports this feature, the method PerformLayout of the class **GraphLayout** cuts the attached graph model into connected components and lays out each connected component separately.

How does the layout of connected components feature work when this mechanism is enabled in the layout classes that support this feature? Instead of directly calling the method Layout to perform the layout on the entire graph, the method **PerformLayout** first cuts the graph into connected components. Then, each connected component is laid out separately by a call of the method layout. To do this, the attached graph is temporarily changed into internally generated graphs corresponding to each of the connected components of the original graph. Finally, the layout instance defined by the property LayoutOfConnectedComponents is used to position the connected components. The default is an instance of **GridLayout**. Its layout region parameter is set by default to the rectangle (0, 0, 800, 800). The property LayoutMode is set to **GridLayoutMode.TileToRows**.

> ***Note:*** *The Tree and Hierarchical layouts contain built-in support for disconnected graphs. For the Tree and Hierarchical layouts, the result can be different from the result of the generic mechanism (the layout of connected components feature) provided by the base class* ***GraphLayout****. Depending your particular needs, you can use either the generic mechanism or the built-in support.*

Using the Filtering Features to Lay Out a Part of a Graph

To understand this section better, see*Using the Graph Model*.

Applications sometimes need to perform the layout algorithm on a subset of the nodes and links of a graph. If the graph is not a GraphicContainer, the custom adapter should support the filtering of a graph. (See *Laying Out Connected Components of a Disconnected Graph*.) The properties that are related to the structure of the graph (Nodes, Links, InterGraphLinks, and so on as shown in *Information on the Structure of the Graph*) must behave just as if the graph has changed in some way. They must take into account only the nodes and links that belong to the part of the graph that must be laid out.

For applications that use **GraphicContainer**, the filtering feature is built into the GraphicContainerAdapter. To do this, the **GraphicContainerAdapter** needs a way to know, for each node or link, whether it must be taken into account during the layout. This is the role of the "filter" class, IGraphLayoutFilter. The **IGraphLayoutFilter** has only one method: Accept(GraphicContainerAdapter model, GraphicObject obj).

To specify a filter, use the property Filter. If a filter is specified, the **GraphicContainerAdapter** calls the **Accept** method for each node or link whenever necessary. If the method returns **true**, the **GraphicContainerAdapter** considers the node or the link as part of the graph that needs to be laid out. Otherwise, it ignores the node or the link.

> ***Note:*** *A link is filtered out automatically as soon as one or both of its origin and destination nodes are filtered out.*

As a convenience, you can also use the method SetIgnored. This specification is obeyed by the **GraphicContainerAdapter** if no filter is specified.

Choosing the Layout Coordinate Space

To understand this section better, see *Using the Graph Model*.

Graph layout algorithms have to deal with the geometry of the graph, that is, the position and shape of the nodes and links. For a graph represented using a GraphicContainer, the geometry is defined in the untransformed coordinate space of the container, while the drawing in a DiagramView is done in the transformed coordinate space of the view. The graph layout API allows you to specify in which geometrical space the layout must be performed. This section is divided as follows:

- Specifying the Coordinates Mode
- Specifying the Reference Transformer
- Specifying the Reference View

Specifying the Coordinates Mode

By default, the GraphicContainerAdapter considers the geometry of the nodes and links in a coordinate space which is appropriate for most of the cases. In some situations, it can be useful to specify a different coordinate space.

To specify the coordinate space, use the property CoordinatesMode. The values are the following:

- **CoordinatesMode.GraphicContainerCoordinates**

  The geometry of the graph is computed using the coordinate space of the graphic container encapsulated by the adapter, without applying any transformation.

This mode should be used if you visualize the graph at zoom level 1, or you do not visualize it at all, or the graph contains only graphic objects for which the drawing size is proportional with the zoom level. In all these cases there is no need to take the transform into account during the layout.

Note that in this mode the dimensional parameters of the layout algorithms are considered as being specified in graphic container coordinates. The reference transform and the reference view are not used.

- **CoordinatesMode.ViewCoordinates**

  The geometry of the graph is computed in the coordinate space of the diagram view. More exactly, all the coordinates are transformed using the current reference transform.

  This mode should be used if you want the dimensional parameters of the layout algorithms to be considered as being specified in diagram view coordinates.

- **CoordinatesMode.InverseViewCoordinates**

  The geometry of the graph is computed using the coordinate space of the diagram view and then applying the inverse transformation using the reference transform. This mode is equivalent to the "graphic container coordinates" if the size of the drawing of the graphic objects is proportional with the zoom level. (A small difference may exist because of the limited precision of the computations.)

  On the contrary, if the size of the drawing of the graphic objects is not proportional with the zoom level (for example, links with a maximum line width), this mode gives different results than the graphic container coordinates mode. These results are optimal if the graph is visualized using the same transform as the one taken into account during the layout.

  Note that in this mode the dimensional parameters of the layout algorithms are considered as being specified in graphic container coordinates.

The default mode is **CoordinatesMode.InverseViewCoordinates**.

The coordinates mode can also be specified directly on the layout instances. For details, see *Coordinates Mode*.

Specifying a Reference Transform

A reference transform can be specified explicitly using the property ReferenceTransform.

In most cases, it is not necessary to specify a reference transform because it is chosen automatically according the following rules:

- If a reference transform is specified, the specified transform is used.
- If a reference view has been specified, the transform of the reference view is used.

- If the **GraphicContainer** encapsulated in the **GraphicContainerAdapter** has at least one **DiagramView**, the transformer of the first view is returned.

The cases where you may need to specify a reference transformer or a reference view are the following:

- The size of the drawing of the graphic objects is not proportional with the zoom level (that is, the layout cannot be correctly computed independently of the transform used for drawing the graph) and more than one diagram view is attached to the graphic container.
- The size of the drawing of the graphic objects is not proportional with the zoom and you want to perform the layout without attaching a diagram view to the graphic container. (Therefore, the default rule for choosing the current transform of the first diagram view as the reference transform cannot be applied.)

If the size of the drawing of the graphic objects is not proportional with the zoom level and the graphic container is displayed simultaneously in several views, you can use the reference view to indicate the view for which you want the drawing of the graph to be optimal.

If you specified a reference transform but want to reset this setting and go back to the default behavior, set the value **null** for the **ReferenceTransform** property.

Specifying a Reference View

Optionally, a **DiagramView** can be specified as a reference view for the **GraphicContainerAdapter**. If a reference view is specified, its current transform (at the moment when the layout is started) is automatically used as the reference transform. Usually, applications use the same diagram view that is used for the display of the **GraphicContainer** as the reference view (but this is not mandatory).

To specify the reference view, use the following property ReferenceView.

Releasing Resources Used During the Layout

Various objects need to be created during the layout process. Most commonly, these are:

- Layout instances (subclasses of GraphLayout).
- Graphic container adapters (GraphicContainerAdapter).
- Other adapters (implementing IGraphModel).
- For recursive layout, you may also instantiate layout providers (subclasses of ILayoutProvider). See also *What is Recursive Layout?*.
- Finally, some of the layout parameters are internally stored as property objects attached to the graph object or to its nodes and links.

To ensure that all these allocated objects are correctly released, you must respect some rules:

1. When a layout instance instantiated by your code is no longer useful, call the method Detach on it to ensure that no graph or graph model is still attached to it. Note that you can freely reuse a layout instance once the previously attached model has been detached.
2. Layout parameters that are specific to a node or a link are cleaned when calling **Detach**. This cleaning is done only for nodes and links that are still in the graph when the **Detach** method is called. If per-node or per-link parameters have been specified and the node or the link needs to be removed before the **Detach** method can be called, you can call the methods CleanNode or CleanLink to perform the cleaning for the node or the link. However, you only need to do so if the removed node or link is reused by your code after removal. Otherwise, if your code does not keep any reference to it, the node or link will be garbage collected anyway, together with the property objects eventually stored by the layout.
3. When a graphic container adapter (or other graph models) instantiated by your code is no longer useful, call the method Dispose to ensure that the resources it has used are released. Note that an adapter (or graph model) must not be used once it has been disposed.

When a layout provider (an instance of ILayoutProvider) instantiated by your code is no longer useful, call the method DetachLayouts(model, true), passing as arguments the graph models that have been used for performing a recursive layout with this provider.

Defining Your Own Type of Layout

If the layout algorithms provided with IBM® ILOG® Diagram for .NET do not meet your needs, you can develop your own layout algorithms by subclassing GraphLayout.

When a subclass of **GraphLayout** is created, it automatically fits into the generic IBM ILOG Diagram for .NET layout framework and benefits from its infrastructure:

- generic parameters: see *Layout Parameters and Features in the GraphLayout Base Class*.
- notification of progress: see *Using Event Handlers*.
- capability to lay out any graph object using the generic graph model: see *Using the Graph Model*.
- capability to apply the layout separately for the connected components of a disconnected graph: see *Laying Out Connected Components of a Disconnected Graph*.
- capability to lay out nested graphs (see *Performing Layout on Nested Graphs*).

This section is divided as follows:

- *Code Sample*
- *Steps for Implementing the Layout Method*

Code Sample

To illustrate the basic ideas for defining a new layout, the following simple example shows a possible implementation of the simplest layout algorithm, the Random Layout. The new layout class is called `MyRandomLayout`.

The following example shows the skeleton of the class:

```
public class MyRandomLayout : ILOG.Diagrammer.GraphLayout.GraphLayout
{
    public MyRandomLayout()
      : base()
    {
    }

    public MyRandomLayout(MyRandomLayout source)
      : base(source)
    {
    }

    public override ILOG.Diagrammer.GraphLayout.GraphLayout Copy()
    {
      return new MyRandomLayout(this);
    }

    protected override void Layout()
    {
      ...
    }
}
Public Class MyRandomLayout
Inherits ILOG.Diagrammer.GraphLayout.GraphLayout

 Public Sub New()
   MyBase.New
 End Sub

 Public Sub New(ByVal source As MyRandomLayout)
   MyBase.New(source)
 End Sub

 Public Overloads Overrides Function Copy() As
ILOG.Diagrammer.GraphLayout.GraphLayout
   Return New MyRandomLayout(Me)
 End Function

 Protected Overloads Overrides Sub Layout()
   ...
 End Sub
End Class
```

The constructor with no arguments is empty. The `copy` constructor and the `Copy` method are implemented; they are used when laying out a nested graph (see *Performing Layout on Nested Graphs*).

Then, the abstract method Layout of the base class is implemented as follows:

```
protected override void Layout()
{
  // obtain the graph model
  IGraphModel graphModel = GetGraphModel();
  // obtain the layout report
  GraphLayoutReport layoutReport = GetLayoutReport();
  int stepsToGo = graphModel.Nodes.Count;
  int stepsGone = 1;

  GraphLayoutReportCode resultCode = GraphLayoutReportCode.LayoutDone;

  // obtain the layout region
  Rectangle2D layoutRegion = GetCalcLayoutRegion();
  float xMin = layoutRegion.X;
  float yMin = layoutRegion.Y;
  float xMax = layoutRegion.X + layoutRegion.Width;
  float yMax = layoutRegion.Y + layoutRegion.Height;

  // initialize the random generator
  System.Random random = ((UseSeedValueForRandomGenerator) ?
    new System.Random((int)(SeedValueForRandomGenerator )) :
    new System.Random());

  // browse the nodes
  ICollection nodes = graphModel.Nodes;
  bool isPreserveFixedNodes = PreserveFixedNodes;
  float x;
  float y;
  Rectangle2D bbox;
  object node;

  // browse the nodes
  foreach (Object node in nodes) {
    // skip fixed nodes
    if (!(isPreserveFixedNodes && GetFixed(node)))
    {
      bbox = graphModel.BoundingBox(node);
     // compute coordinates
      x =
        xMin + Math.Max(0, xMax - bbox.Width - xMin) *
               (float)(random.NextDouble());
      y =
        yMin + Math.Max(0, yMax - bbox.Height - yMin) *
               (float)(random.NextDouble());

      // move the node to the computed position
      graphModel.MoveNode(node, x, y);

      IncreasePercentageComplete(stepsGone++ * 100 / stepsToGo);
      // notify handlers on layout events
      OnLayoutStepPerformedIfNeeded();
      if (IsLayoutTimeElapsed() || IsStoppedImmediately())
      {
        resultCode = GraphLayoutReportCode.StoppedAndInvalid;
        break;
      }

      IncreasePercentageComplete(100);
      OnLayoutStepPerformed(false, false);

      // set the layout report code
      layoutReport.Code = resultCode;
    }
  }
}
Protected Overloads Overrides Sub Layout()
  Dim graphModel As IGraphModel = GetGraphModel()
  Dim layoutReport As GraphLayoutReport = GetLayoutReport()
  Dim stepsToGo As Integer = graphModel.Nodes.Count
  Dim stepsGone As Integer = 1
  Dim resultCode As GraphLayoutReportCode = GraphLayoutReportCode.LayoutDone
  Dim layoutRegion As Rectangle2D = GetCalcLayoutRegion()
  Dim xMin As Single = layoutRegion.X
  Dim yMin As Single = layoutRegion.Y
  Dim xMax As Single = layoutRegion.X + layoutRegion.Width
  Dim yMax As Single = layoutRegion.Y + layoutRegion.Height
  Dim random As Random
  If (UseSeedValueForRandomGenerator) Then
    random = New Random(SeedValueForRandomGenerator)
  Else
    random = New Random()
  End If

  Dim isPreserveFixedNodes As Boolean = PreserveFixedNodes
  Dim x As Single
  Dim y As Single
  Dim bbox As Rectangle2D

  For Each node As Object In graphModel.Nodes
    If Not (isPreserveFixedNodes AndAlso GetFixed(node)) Then
      bbox = graphModel.BoundingBox(node)
      x = xMin + Math.Max(0, xMax - bbox.Width - xMin) * random.NextDouble
      y = yMin + Math.Max(0, yMax - bbox.Height - yMin) * random.NextDouble
      graphModel.MoveNode(node, x, y)
      IncreasePercentageComplete(stepsGone * 100 / stepsToGo)
      stepsGone += 1
      OnLayoutStepPerformedIfNeeded()
      If IsLayoutTimeElapsed() OrElse IsStoppedImmediately() Then
        resultCode = GraphLayoutReportCode.StoppedAndInvalid
        Exit For
      End If
    End If
  Next
  IncreasePercentageComplete(100)
  OnLayoutStepPerformed(False, False)
  layoutReport.Code = resultCode
End Sub
```

Steps for Implementing the Layout Method

In this example, the **Layout** method is implemented using the following main steps:

1. Obtain the graph model (GetGraphModel on the layout instance).

   ```
   IGraphModel graphModel = GetGraphModel();
   ```

2. Obtain the instance of the layout report that is automatically created when the PerformLayout method from the superclass is called (GetLayoutReport on the layout instance). See *Using a Graph Layout Report*.

   ```
   GraphLayoutReport layoutReport = GetLayoutReport();
   ```

3. Obtain the layout region parameter to compute the area where the nodes will be placed.

   ```
   Rectangle2D rect = GetCalcLayoutRegion();
   ```

4. Initialize the random generator.

   ```
   System.Random random = (UseSeedValueForRandomGenerator) ?
     new System.Random(SeedValueForRandomGenerator) :
     new System.Random();
   ```

   (For information on the seed value parameter, see *Random Generator Seed Value*.)

5. Get the collection of nodes using the property Nodes.

   ```
   ICollection nodes = graphModel.Nodes;
   ```

   (For details on fixed nodes, see *Preserve Fixed Nodes*).

6. Move each node to the newly computed coordinates inside the layout region (MoveNode).

   ```
   graphModel.MoveNode(node, x, y);
   ```

7. Notify the handlers on layout events that a new node was positioned (OnLayoutStepPerformedIfNeeded on the layout instance). This allows the user to implement, for example, a progress bar if a layout event handler was registered on the layout instance.

   ```
   OnLayoutStepPerformedIfNeeded();
   ```

   For details on event handlers, see *Using Event Handlers*.

8. Finally, set the appropriate code in the layout report.

   ```
   GraphLayoutReportCode resultCode = GraphLayoutReportCode.LayoutDone;
   ```

```
if (IsLayoutTimeElapsed() || IsStoppedImmediately())
{
  resultCode = GraphLayoutReportCode.StoppedAndInvalid;
  break;
}

layoutReport.Code = resultCode;
```

Of course, depending on the characteristics of the layout algorithm, some of these steps may be different or unnecessary, or other steps may be needed.

Depending on the particular implementation of your layout algorithm, other methods of the **GraphLayout** class may need to be overridden. For instance, if your subclass supports some of the generic parameters of the base class, you must override the Supports[ParameterName] method (see *Layout Parameters and Features in the GraphLayout Base Class*). For further information about the API of the class **GraphLayout**, please refer to the API Reference Manual.

Related Documentation

Several books dedicated to graph layout have 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.mypearsonstore.com/bookstore/product.asp?isbn=0133016153

Kaufmann, Wagner (Eds.): Drawing Graphs, Lecture Notes in Computer Science Vol. 2025, Springer 2001. See:

http://link.springer.de/link/service/series/0558/tocs/t2025.htm

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 at:

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://jgaa.info/

Algorithmica

http://link.springer-ny.com/link/service/journals/00453/

Computational Geometry: Theory and Applications

http://www.elsevier.com/locate/comgeo

Journal of Visual Languages and Computing

http://www.elsevier.com/locate/jvlc

The following journals occasionally publish papers on graph layout:

Information Processing Letters

http://www.elsevier.com/locate/ipl

Computer-aided Design

http://www.elsevier.com/locate/cad

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. The 2008 Symposium on Graph Drawing was held in Heraklion, Crete, Greece:

http://gd2008.org/

The 2009 Symposium will be held in Chicago, USA.

Questions and Answers about Using the Layout Algorithms

This section provides some helpful suggestions for using the layout algorithms. You may find some answers to questions that come up when using the graph layout API.

Question	**Answer**
I perform the layout and nothing happens (no node is moved). Why?	One possible reason may be: the layout algorithms provided in IBM® ILOG® Diagram for .NET are all designed to do nothing, by default, if no change occurred in the graph since the last time the layout was performed successfully on the same graph. A change means that a node was moved, or a node or link was added, removed, or reshaped. Note that you can force the layout to be performed again, even if no change occurred, by calling the PerformLayout(boolean force) method with a **true** value for the force argument. Another possible reason may be: an error or a special case occurred during the layout. First, you should check whether the **PerformLayout** method has thrown an exception. If no exception was thrown, check the Code on the instance of the layout report returned by the **PerformLayout** method. Check this value with respect to the documentation of the appropriate layout report class. (For details, see *Using a Graph Layout Report*.)
With the ForceDirectedLayout, after having performed the layout once, I don't see any movement even if I use the force layout option. Why?	The reason is probably that the first time you performed the layout, the algorithm reached the convergence. When the layout is performed again, it detects that the convergence has been already reached and stops. If you really want to continue working, for instance in order to "declutter" a particular part of the graph, you may need to move one or several nodes in order to change the initial configuration. (The algorithm is dependent on the initial configuration.)

After performing the layout, the graph is laid out far from its initial position. Why?	Some layout algorithms use a layout region parameter to control the size and position of the layout. (For details, see *Layout Region*.) Depending on the value of this parameter, the nodes may be moved far from their initial positions. To know whether a layout algorithm is designed to use a layout region parameter, check the documentation to see if the layout class overrides the SupportsLayoutRegion method of the base class in order to return **true**. Other algorithms have a different mechanism that allows you to specify the desired location of the layout. It may happen that the default value of the location parameter is such that the graph is laid out far from its initial position.
When I use certain layout algorithms on certain graphs, there are overlapping nodes. Why and what can I do?	One possible reason may be related to the different ways layout algorithms deal with the size of the nodes: - The Tree, Hierarchical, and Grid algorithms always avoid overlapping nodes. (The link layout algorithms do not move the nodes. They only reshape the links such that the crossings and overlaps are reduced. The size of the nodes is taken into account.) - The Force-Directed Layout (with the option "Respect Node Sizes" enabled) succeeds in avoiding overlapping nodes in many cases, but not always. In any case, if the layout algorithm supports the layout region mechanism (see *Layout Region*), you should try to increase the size of the layout region. For example, if your graph contains hundreds of nodes, it is not reasonable to use a small layout region, such as 600x600. There will be not enough space for all the nodes. You should try a larger layout region, for example 5000x5000. The optimal size of the layout region depends, of course, not only on the number of nodes, but also on their size. If the nodes are relatively large with respect to the size of the layout region, it may be necessary to adjust some of the parameters (for instance, the preferred link length for the Force-Directed Layout).

In some networks, there are two (or more) subnetworks that are not connected. How will this affect the layout algorithms?	This depends on the layout class you use: - HierarchicalLayout, TreeLayout: They have built-in support for disconnected graphs. Alternatively, you can use the automatic support from the base class. (See *Layout of Connected Components*.) - ShortLinkLayout, LongLinkLayout, GridLayout, RandomLayout: These algorithms support both connected and disconnected graphs. Their behavior is the same for both categories of graphs. - ForceDirectedLayout: This algorithm supports disconnected graphs, but usually it is better to rely on the automatic "layout of connected components" parameter. (See *Layout of Connected Components*.)
There are some attributes of the network that we know about (for instance, we know what the core switch is and what the center should be). Are such attributes taken into account by the layout algorithm?	It depends on the layout algorithm. - In the Tree Layout, you can specify the root node. - In the Hierarchical Layout algorithm, you can specify node position indices and level indices, as well as relative positioning constraints.

If I use IBM ILOG Diagram for .NET on different computers. I sometimes get different layouts for the same graph and with the same parameters. Why?	There are two possible reasons: 1. Different computers may be slower or faster. If the layout algorithm you use stops the computation when the specified allowed time has elapsed, a slower computer will cause the computation to stop earlier. That may be the cause of different results. This may happen even with the same computer if the charge of the computer is increased. You may need to increase the allowed time specification when running on a slower computer. 2. If you use a layout algorithm that uses the random generator and if you use the default option for the seed value (that is, the system clock is used), you get different results for each successive run of the layout on the same graph. This allows you to obtain alternative results and to choose the one you prefer. If you want to prevent different results for successive runs, you can specify a constant seed value.
I use the Link Layout algorithms to lay out the links (representing routes) of a network of graphical objects (towns) geo-positioned on a cartographical map. When several links connect to the same side of a node, they overlap, while I expect them to respect the "link offset" (or the "grid size") parameter of the Link Layout. Why?	Some dimensional parameters of the layout algorithms need to be chosen with respect to the size of the nodes. This is the case of the "link offset" and the "bypass distance" parameters for the Short Link Layout and the grid size for the Long Link Layout. Indeed, their default values are not appropriate when the nodes are very large. Often, nodes placed on a map, for instance a world map, have a very large size. Compared to this size, the default values of the parameters are so small that they appear to be zero. The solution is to increase the values of the dimensional parameters, taking into account the size of the nodes. If different nodes have different sizes, either the medium or the largest size of the nodes can be used to compute the parameters as a fraction of this size.

Index