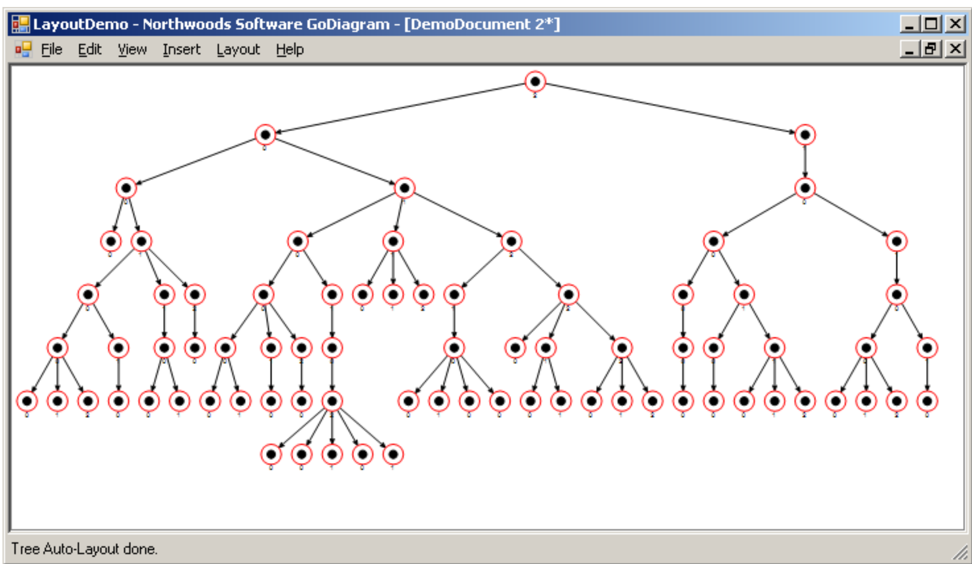
GoLayout for .NET

Automatic Layout Library

for GoDiagram for .NET

User Guide



This guide provides information on using the classes provided in the **GoLayout™ for Microsoft® .NET** for **GoDiagram™ for Microsoft® .NET**.

September 2016

Northwoods Software Corporation
142 Main St.
Nashua, NH 03060

http://www.nwoods.com

Contents

PREFACE

Purpose of this guide:

This guide provides an overview of **GoLayout for .NET**, .NET class libraries containing sets of components for automatically rearranging nodes in a **GoDiagram for .NET** document:

- **GoLayout for .NET Windows Forms**, as the `Northwoods.Go.Layout.dll` assembly

Understanding this guide requires familiarity with the .NET platform and with Windows Forms.

The classes in the libraries are identical—in fact they are all named identically to facilitate learning and porting code. The only difference is the namespace, which is **Northwoods.Go.Layout** for **GoLayout for .NET Windows Forms.**

For more detailed information about the types, classes and interfaces, see the appropriate **GoDiagram for .NET** Class Reference Manual. (online help)

Who should use this guide:

This guide is intended for application programmers using one of the **GoLayout for .NET** libraries.

This manual assumes you are familiar with Microsoft .NET and **GoDiagram for .NET** programming concepts and terminology. If you are not, please refer to your Microsoft .NET and **GoDiagram for .NET** documentation or online help.

Structure of this guide:

This guide is organized as follows:

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

1. INTRODUCTION

The **GoLayout for .NET** class library is a set of classes built to interface with the **GoDiagram for .NET** class library and provide support for automatically laying out graphs (node & arc diagrams) by positioning the nodes and routing the links.

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

GoLayout currently supports three general auto-layout routines: *a force-directed auto-layout* routine, a *layered-digraph auto-layout* routine, and a *tree auto-layout* routine. The force-directed auto-layout routine is intended for use with all types of graph – undirected graphs as well as directed graphs. The layered-digraph and tree auto-layout routines are intended specifically for use with directed graphs, including trees and hierarchies.

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

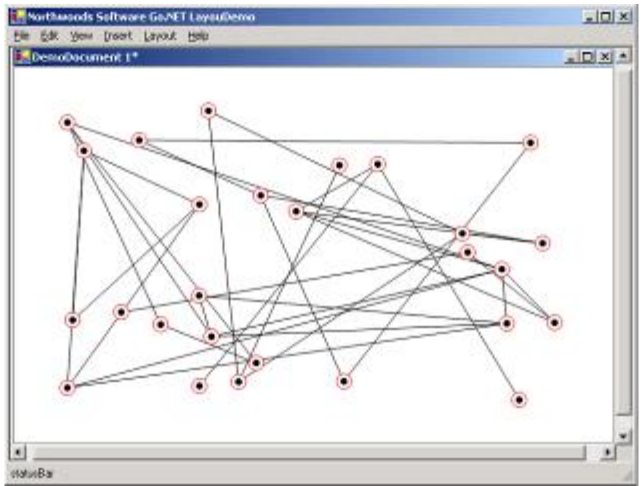


Figure 1. Sample graph before layout

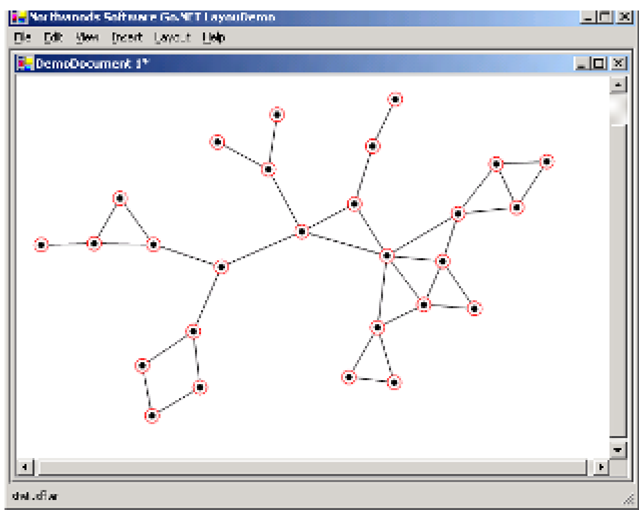


Figure 2. Sample graph after Force-Directed Auto-Layout

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

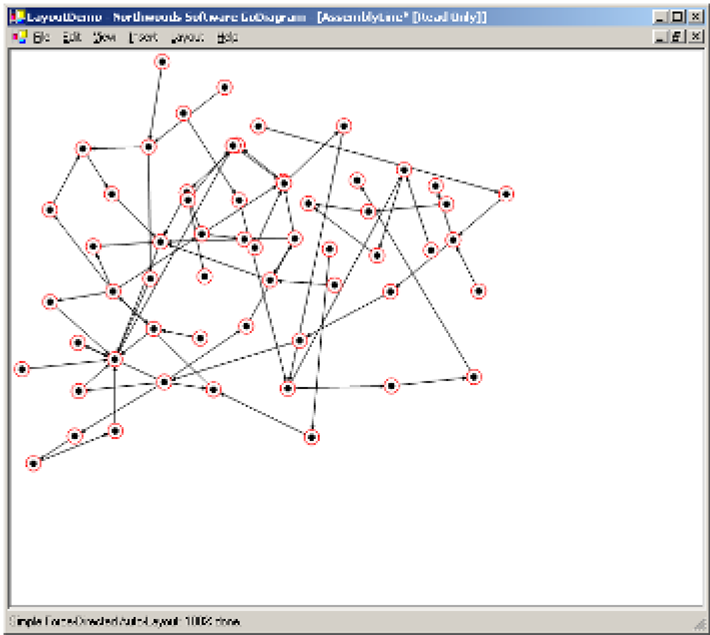


Figure 3. Sample graph before layout

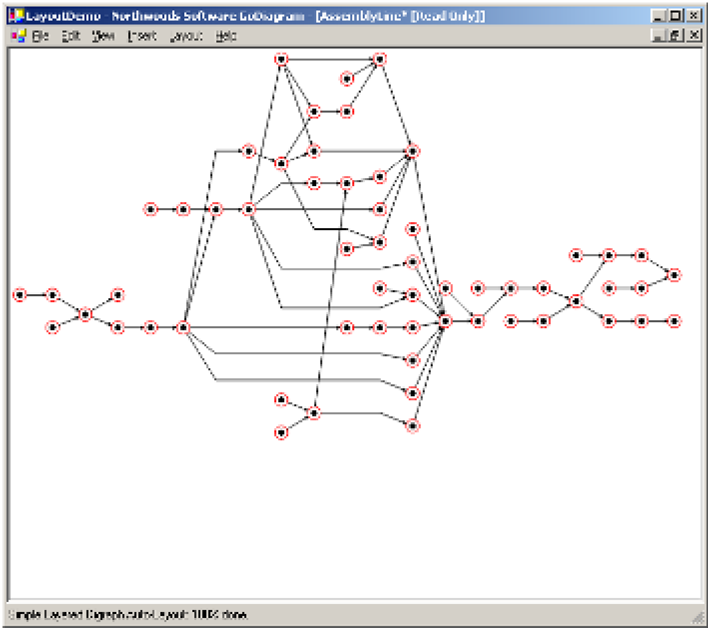


Figure 4. Sample graph after Layered-Digraph Auto-Layout

Figure 5 and Figure 6 illustrate a sample graph before and after applying tree automatic layout.

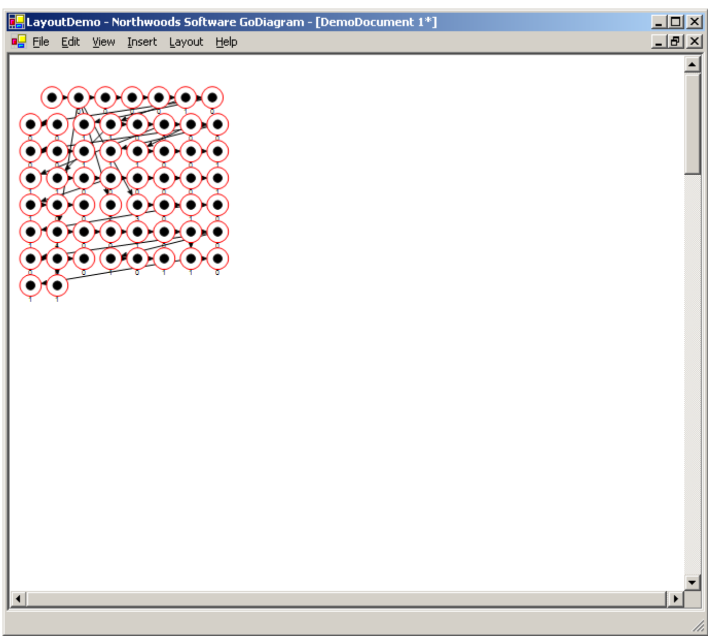


Figure 5. Sample graph before layout

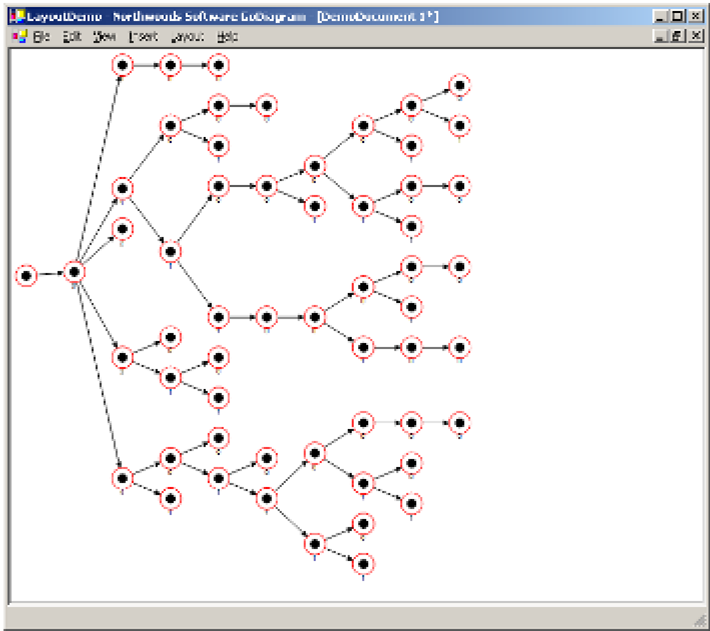


Figure 6. Sample graph after Tree Auto-Layout

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

2. The Layout Demo Sample Application

Introduction to the "Layout Demo" Sample Application

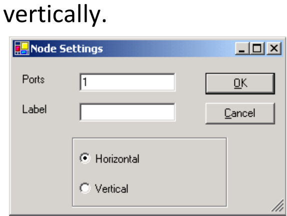
"Layout Demo" is the primary sample application for the **GoLayout** library.

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

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

Layout Demo Menus

This section describes the interesting Layout Demo menu commands.

View Commands	Description
Toggle Arrowheads	Turns arrowheads on links on or off.
Insert Commands	**Description**
Basic Node	Opens a dialog box for creating a new node. The dialog prompts for the number of ports, an optional node label, and whether the ports should be oriented horizontally or vertically. 
Generate Random Tree	Clears the document and adds a bunch of nodes connected together by links to form a tree structure.

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

Layout Demo Quick Start

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

Force-Directed Auto-Layout

First, we examine the force-directed auto-layout routine. To begin, create a number of one-port nodes using the **Basic Node** menu item or by double clicking on the background. Move them around and link them together into a cycle to create a graph similar to that in Figure 7. Notice that the nodes have an initial color of red. LayoutDemo uses the color of a node to demonstrate some of the customizable aspects of the Layout routines. Double-click anywhere inside of a node to change its color.

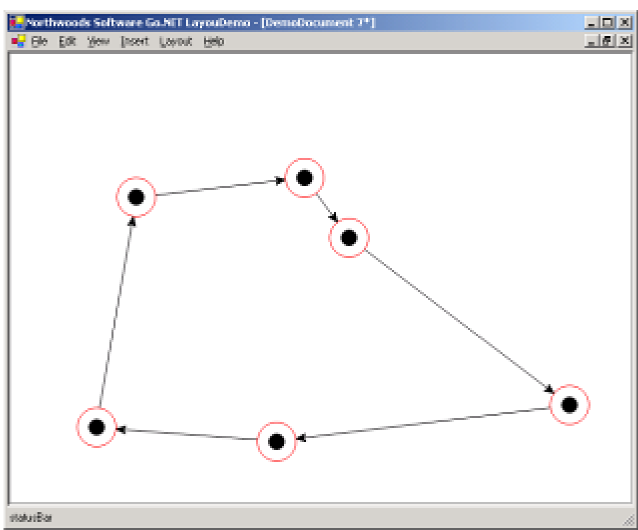


Figure 7. Example 1

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

After creating a graph, choose the **Force-Directed Auto-Layout** menu item, or type Ctrl-F. This will bring up the modeless form illustrated in Figure 8.

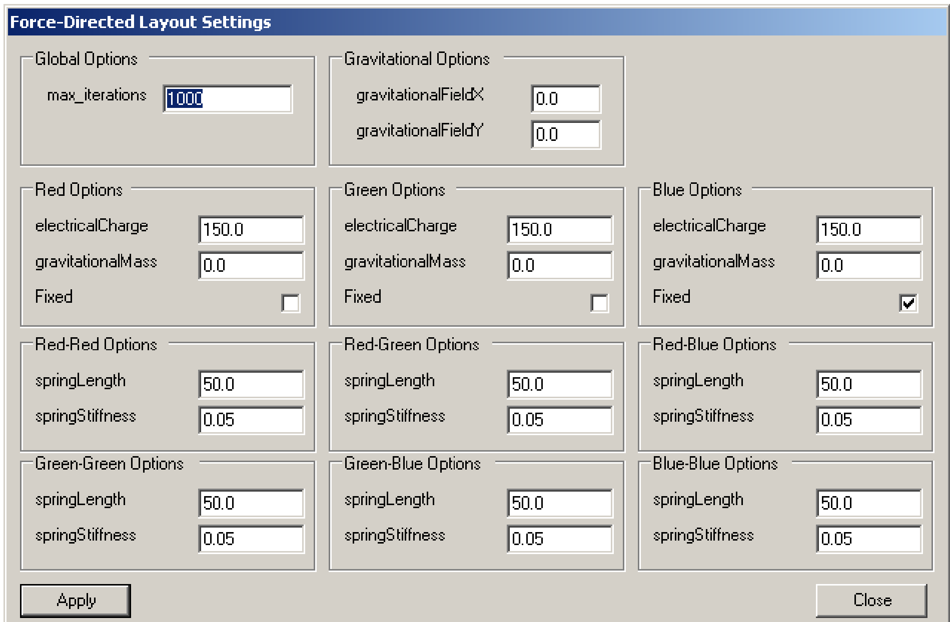


Figure 8. Dialog box for Force-directed Auto-Layout

Examine the different options available for the force-directed auto-layout, but leave the default values and click **Apply** (or type Enter). The graph will animate as it moves towards its final position, similar to that shown in Figure 9.

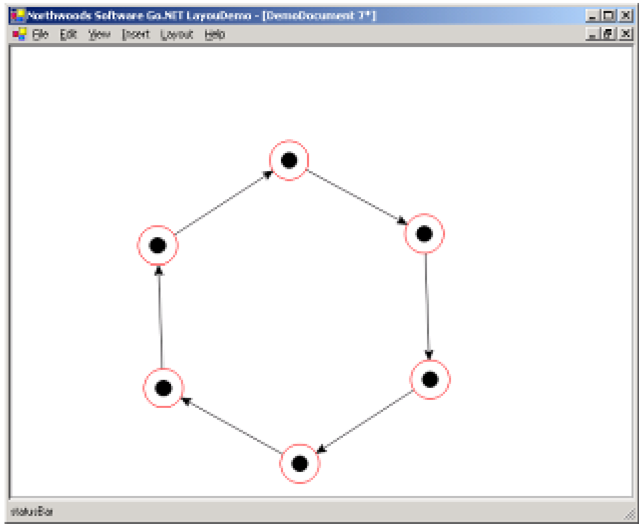


Figure 9. Result of applying Force-Directed Auto-Layout to Example 1

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

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

See what happens when you change some of the default values. Choose the **Force-Directed Auto-Layout** menu item, but change the value of **electricalCharge** under **Red Options** to 300 and click **Apply**. Notice that with a higher electrical charge, the nodes repel each other more, and the result is a graph with greater distances between adjacent nodes.

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

As a final example, move one node some distance away from the rest of the nodes. Double-click on the node to change its color to green. Choose the **Force-Directed Auto-Layout** menu item, select **fixed** under **the Green Options**, and change the value of **springLength** under **Red-Green Options** to 200 and click **Apply**. Now, the green node will remain fixed and the other nodes move towards it. Further, the longer **springLength** between the red and green nodes will result in a greater distance between the red and green nodes than between the red nodes, as illustrated in Figure 10.

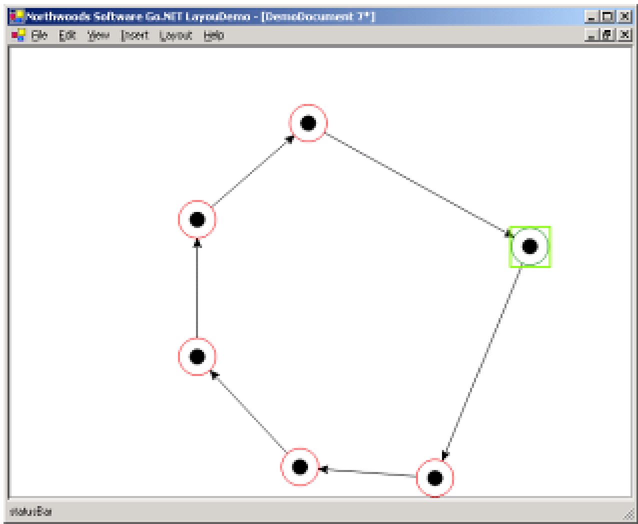


Figure 10. Result of changing parameters

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

Setting a **gravitationalFieldX** and **gravitationalFieldY** induces a field over the entire document. The gravitational field only affects nodes with a **gravitationalMass**. Try values of -1 for **gravitationalFieldX** and 1 for **gravitationalMass**.

Layered-Digraph Auto-Layout

Next, we examine the layered-digraph auto-layout routine. Create a new document, create a number of one-port nodes using the **Basic Node** menu item or by double clicking on the background, move them around, and link them together into a tree similar to that shown in Figure 11.

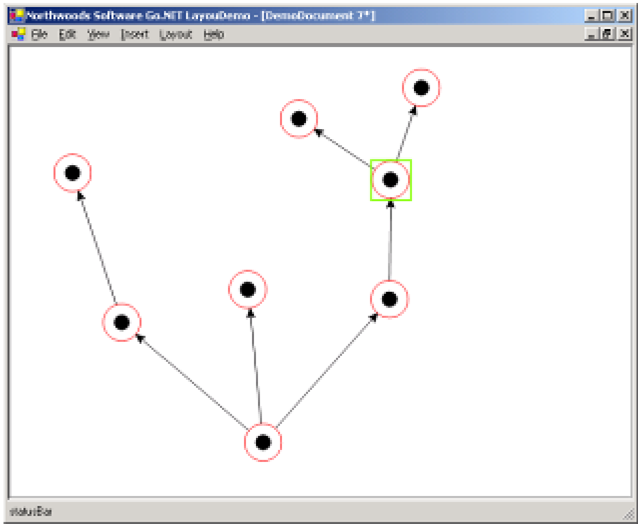


Figure 11. Sample Directed Graph

Now, choose the **Layered-Digraph Auto-Layout** menu item, or type Ctrl-L. This will bring up the modeless form shown in Figure 12.

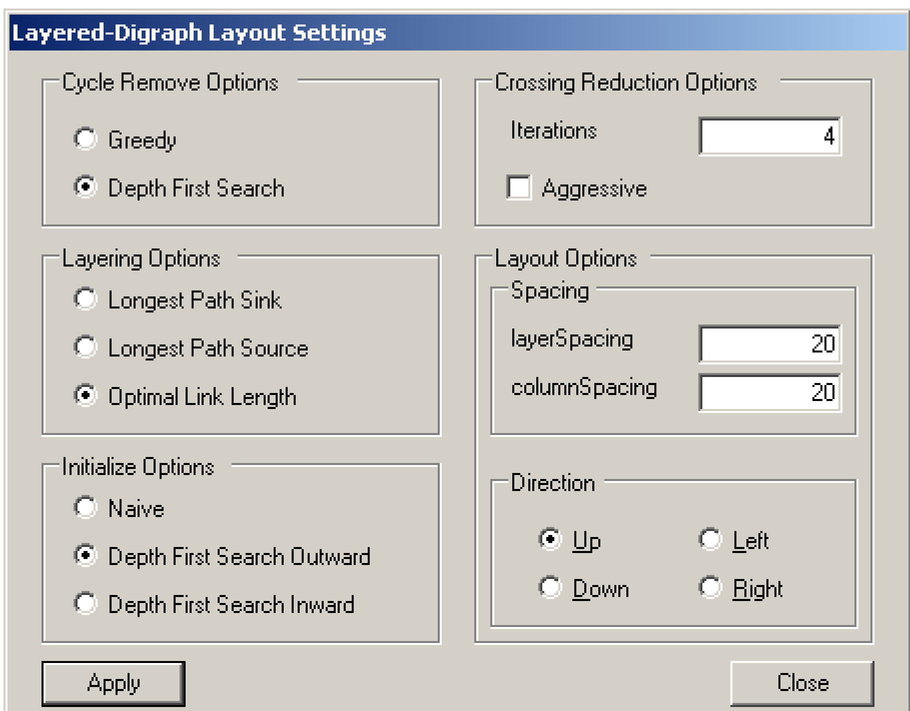


Figure 12. Layered Digraph Auto-Layout Options dialog box

Examine the different options available for the layered-digraph auto-layout, but leave the default values and click **Apply** (or type Enter). The graph will be redrawn in its final position in a manner similar to that shown in Figure 13. The exact arrangement may depend on the order in which you drew the links.

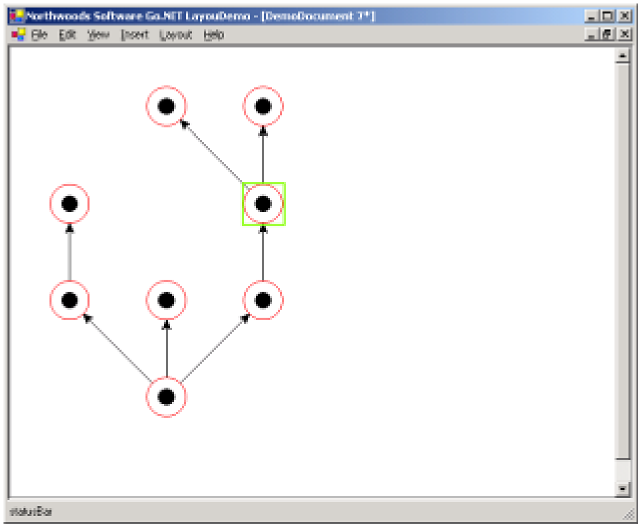


Figure 13. Resulting layout after Layered-Digraph Auto-Layout

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

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

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

Figure 14 illustrates a more complicated graph which has been drawn using the layered-digraph auto-layout routine.

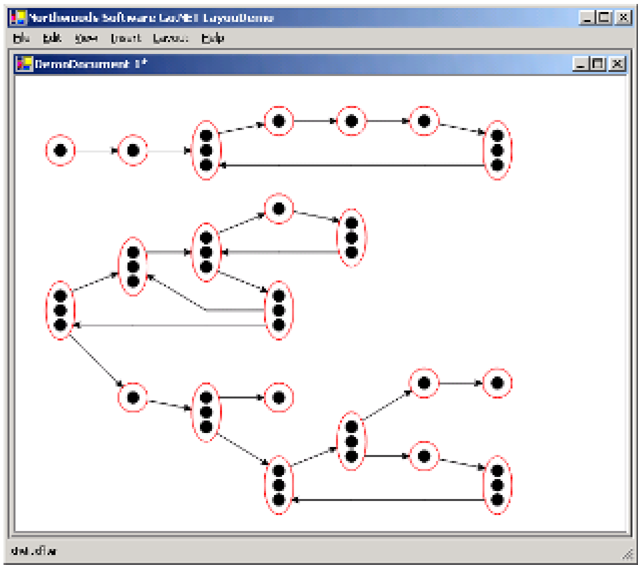


Figure 14. Result of applying Layered-Digraph Auto-Layout to more complex graph

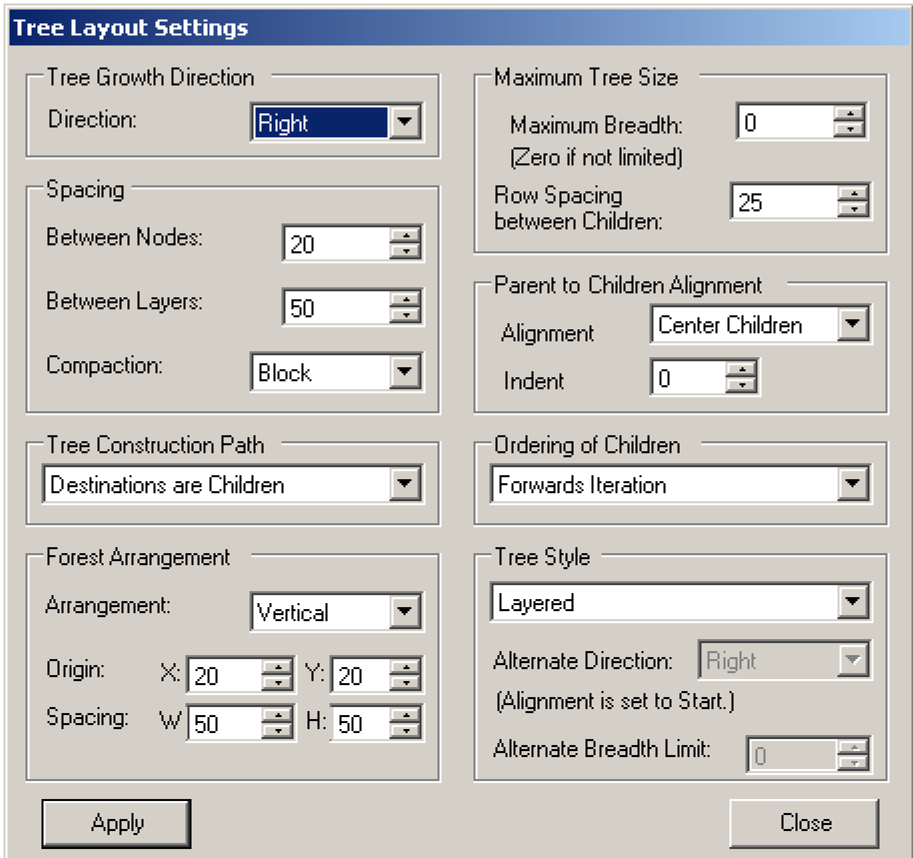
This graph shows the consideration that the layered-digraph auto-layout routines give to nodes with multiple ports. The relative positions of ports within a node are used both in reducing the number of link crossing and in straightening the links.

Tree Auto-Layout

Finally we take a quick look at the tree auto-layout.

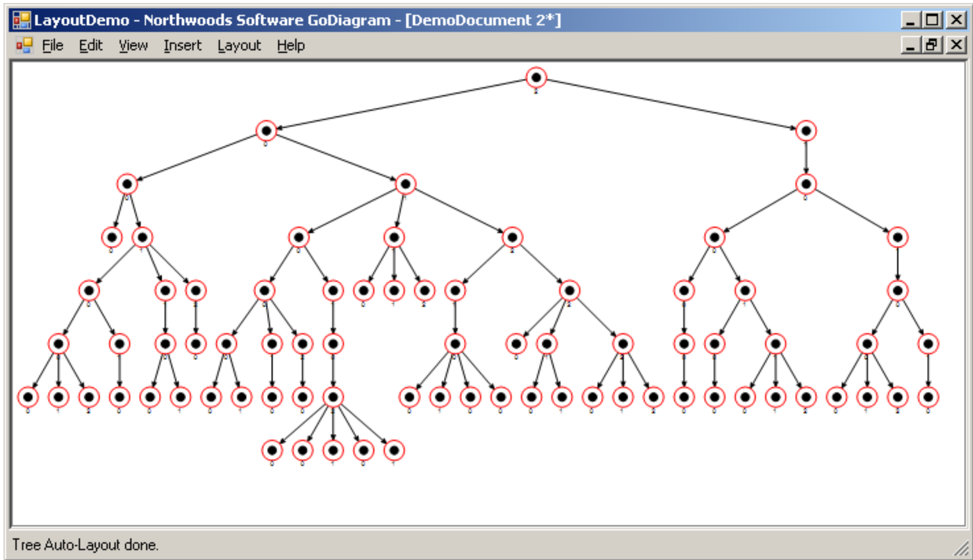
You can construct your own tree of nodes manually, or you can invoke the Generate Random Tree command on the Insert menu.

Then invoking the Tree Layout command on the Layout menu, or typing Ctrl-T, will bring up the following modeless form:

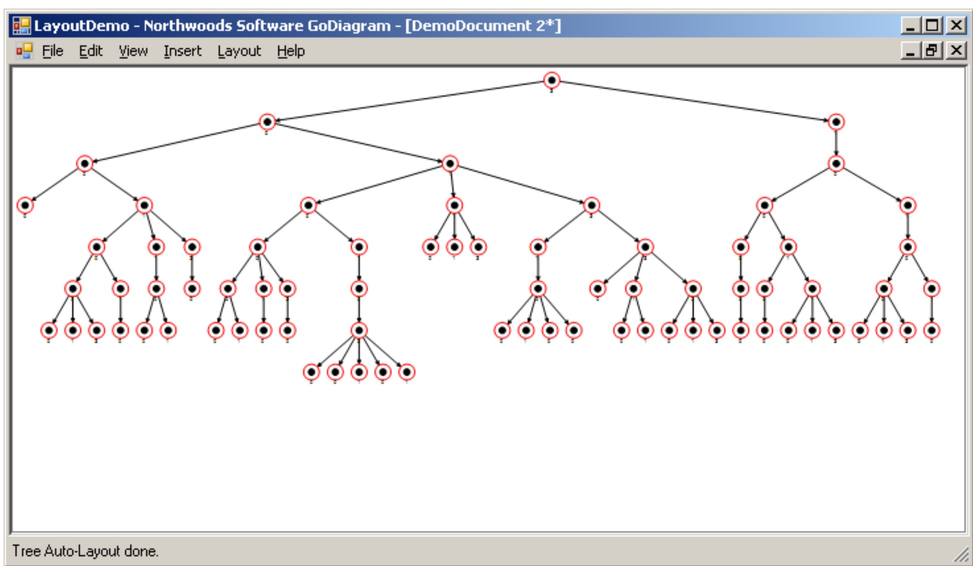


You can change any of the settings and click on **Apply** to perform the layout, or just type Ctrl-T again in the canvas window.

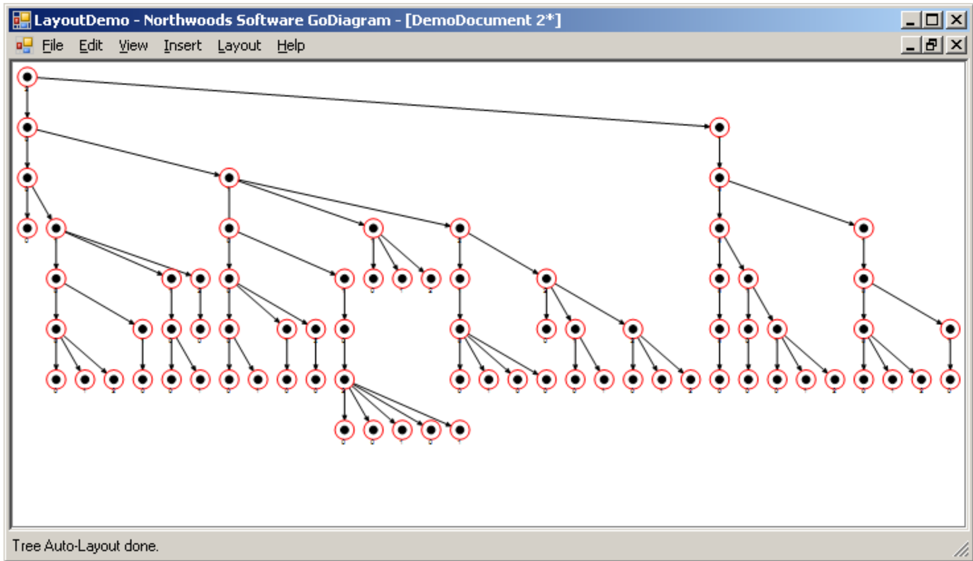
The **Spacing** settings control how close the nodes are to each other, between sibling nodes, and between parents and their children. The **Compaction** option controls whether subtrees are fit closer together to allow overlap in depth without overlapping any individual nodes. For example, with Block compaction:



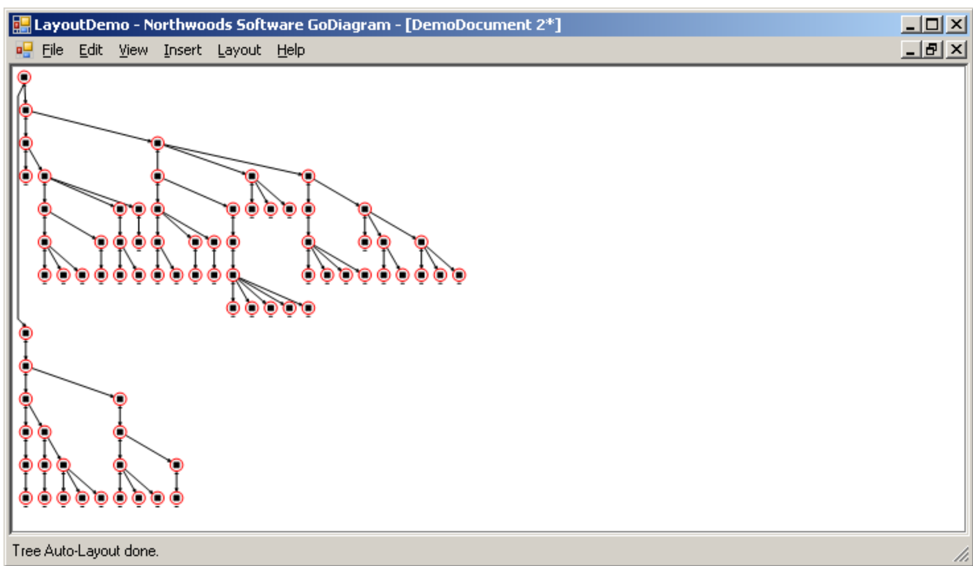
But with no compaction:



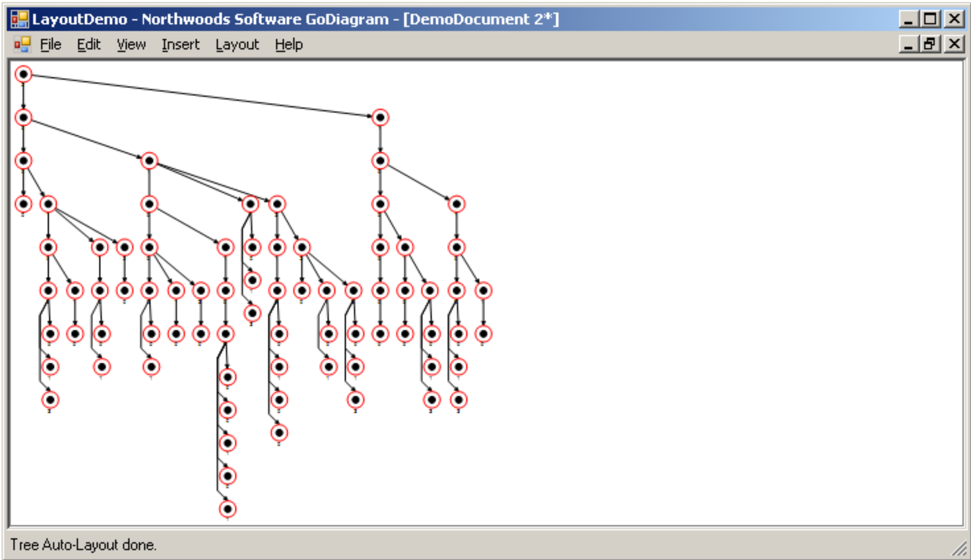
You can also control how the parent node is positioned relative to its children. If you specify an **Alignment** of Start, and **Compaction** is back to the default value of Block, you get the following effect:



You can have broad trees take up less breadth by limiting the breadth. Subtrees and nodes can be laid out in multiple rows. For example, the same tree laid out with a **Maximum Breadth** of 1500:

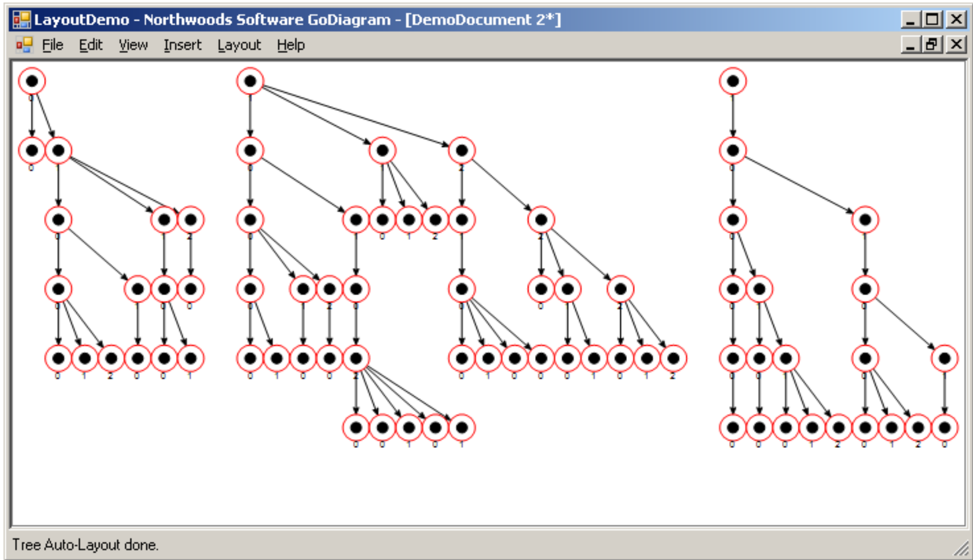


You can also reduce the breadth by using the **Tree Style** that uses a very narrow breadth limit for those parent nodes that only have children but no grandchildren, called LastParents. When the tree style is LastParents, you can specify different tree layout settings for those "last parent generation" nodes. If you specify a **Breadth Limit** of just 1, it will force all of those children to be laid out in a single column:



Note how the children of a node, if there isn't enough breadth allowed to place them all in a single row, will lay them out in multiple rows. In the above case, where the **Breadth Limit** is 1, all of the children are forced to be in new rows, causing them all to form a single column of children.

Finally, this Tree Layout form allows you to control how separate trees are laid out relative to each other in the document. The **Forest Arrangement** settings control how separate trees are positioned next to each other, where the first one is positioned (the **Origin**) and how much space there should be between separate trees. For example, after modifying the graph by deleting the three nodes at the root, and then specifying a Horizontal **Arrangement**:


LayoutDemo - Northwoods Software GoDiagram - [DemoDocument 2*]
File Edit View Insert Layout Help
Tree Auto-Layout done.

3. Quickly Adding Layout to Your Go Application

Integrating **GoLayout** into an existing **Go** application is very easy. This section will take you through the steps of adding **GoLayout** to a generic **Go** application.

References

Make sure your project refers to the **Go** and **GoLayout** assemblies. These are located in the `lib` subdirectory of the GoDiagram installation. The assemblies are named:

```
Northwoods.Go.dll

Northwoods.Go.Layout.dll
```

Make sure your source code uses the **Go** and **GoLayout** namespaces:

VB.NET for Windows Forms:

```
Imports Northwoods.Go
Imports Northwoods.Go.Layout
```

C# for Windows Forms:

```
using Northwoods.Go;
using Northwoods.Go.Layout;
```

The **Imports/using** statement lets the .NET compiler know about the base **Go** classes and the **GoLayout** library.

Performing an Automatic Layout

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

You can dynamically create an instance of a layout object, assign its properties, and then call **PerformLayout**. For example:

VB.NET:

```
  Public Sub LayerAction()
    Dim layout As GoLayoutLayeredDigraph = New GoLayoutLayeredDigraph()
```

```
    layout.Document = goView1.Document
    ' maybe set other properties too . . .
    layout.PerformLayout()
  End Sub
```

C#:

```
  void LayerAction() {
    GoLayoutLayeredDigraph layout = new GoLayoutLayeredDigraph());
    layout.Document = this.goView1.Document;
    // maybe set other properties too . . .
    layout.PerformLayout();
  }
```

Or a command to perform force-directed automatic layouts:

VB.NET:

```
  Public Sub ForceAction()
    Dim layout As GoLayoutForceDirected = New GoLayoutForceDirected()
    layout.Document = goView1.Document
    ' maybe set other properties too . . .
    layout.PerformLayout()
  End Sub
```

C#:

```
  void ForceAction() {
    GoLayoutForceDirected layout = new GoLayoutForceDirected();
    layout.Document = this.goView1.Document;
    // maybe set other properties too . . .
    layout.PerformLayout();
  }
```

Finally, for tree layouts:

VB.NET:

```
  Public Sub TreeAction()
    Dim layout As GoLayoutTree = New GoLayoutTree()
    layout.Document = goView1.Document
    ' maybe set other properties too . . .
    layout.PerformLayout()
  End Sub
```

C#:

```
  void TreeAction() {
    GoLayoutTree layout = new GoLayoutTree();
    layout.Document = this.goView1.Document;
    // maybe set other properties too . . .
    layout.PerformLayout();
  }
```

That's it!

The constructors for **GoLayoutLayeredDigraph** and **GoLayoutForceDirected** and **GoLayoutTree** used above initialize the auto-layout options to default values. Then, once the **Document** property has been set, **PerformLayout()** can automatically create a **GoLayoutNetwork** from it. Clearly, these values will not be suitable for all applications. See the Advanced Options section of this guide for details regarding customizing the auto-layout routines. Further customization is available by subclassing the **GoLayout** classes. See the Advanced Options section of this guide and the Reference Manual Help file for details about the appropriate properties and methods.

One common variation is to just perform an automatic layout on the nodes in one layer of the document. For example:

VB.NET:

```
GoLayer aLayer = ...
Dim layout As GoLayoutLayeredDigraph = new GoLayoutLayeredDigraph()
layout.Document = myView.Document
layout.Network = layout.CreateNetwork()
layout.Network.AddNodesAndLinksFromCollection(aLayer, false)
layout.PerformLayout()
```

C#:

```
GoLayer aLayer = ...;
GoLayoutLayeredDigraph layout = new GoLayoutLayeredDigraph();
layout.Document = myView.Document;
layout.Network = layout.CreateNetwork();
layout.Network.AddNodesAndLinksFromCollection(aLayer, false);
// maybe set other properties too . . .
layout.PerformLayout();
```

The second argument to **AddNodesAndLinksFromCollection** controls whether to layout only objects implementing **IGoNode** (if true) or all objects (if false).

If you just want to do an automatic layout on the objects that the user has selected, in the code above replace "`aLayer`" with "`aView.Selection`". However note that this assumes that the user will select all the relevant links in addition to the nodes. If not all of the links are selected, only a partial **GoLayoutNetwork** is created, thereby giving the layout routine a different view of the graph than you might have expected.

4. GoLayout Concepts

Design Philosophy

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

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

The default implementations of these functions should be adequate for most applications, but subclassing the **GoLayout** classes can lead to better layouts.

IGoLayoutNetwork, IGoLayoutNetworkNode, and IGoLayoutNetworkLink

The **IGoLayoutNetwork** interface provides a view of an abstract network (graph) of nodes and directed links. These nodes and links, implementing the **IGoLayoutNetworkNode** and **IGoLayoutNetworkLink** interfaces, generally correspond to top-level **GoObject**s in a **GoDocument**. An instance of **IGoLayoutNetwork** is a collection containing instances of **IGoLayoutNetworkNode**s and **IGoLayoutNetworkLink**s.

For each of the kinds of **GoLayout**, there is a set of layout-specific classes that implement **IGoLayoutNetwork**, **IGoLayoutNetworkNode,** and **IGoLayoutNetworkLink**. For example, **GoLayoutTree** makes use of a **GoLayoutTreeNetwork** that contains **GoLayoutTreeNode**s and **GoLayoutTreeLink**s.

The purpose of **IGoLayoutNetwork** is to provide a framework for manipulating the state of nodes and links without affecting the document **GoObject**s.

By default, a network is created from a **GoDocument** by adding all top-level **GoObject**s that are not ports or links as nodes to the network. Alternatively, any other **IGoCollection** object can be used instead such that only those **GoObject**s that are in the collection are added as nodes to the network.

All top-level **IGoLink**s as are added, by default, as links to the network. If an **IGoCollection** is provided, then only those **IGoLink**s that are in that collection are added as links to the network. Note that links which are in the collection, but whose corresponding to-nodes and from-nodes are not in the collection, will not be added to the network.

The majority of applications will simply let the auto-layout class construct the network from a document. They need to construct an auto-layout class and then add nodes and links from the current **GoDocument** or another IGoCollection. However, more sophisticated results can be achieved by combining modifications to the **IGoLayoutNetwork** with auto-layout subclasses written to recognize the modifications. For example, you can programmatically modify the network to remove some network nodes and/or network links. This would cause a subset of the **GoObject**s in the document to be moved.

As another example, network nodes and network links that have no relationship to any **GoObject**s on the screen can be introduced into the network to influence the final layout. The **IGoLayoutNetworkNode** and **IGoLayoutNetworkLink** interfaces provide a property, named **UserObject**, for Objects used to hold user information which can be used to mark or otherwise distinguish particular nodes and links in the network.

Those interested in writing subclasses of the auto-layout classes should familiarize themselves with the specific network node and network link classes, particularly the **GoObject** property. This property returns the top-level **GoObject** (in the **GoDocument**) which is represented by the **IGoLayoutNetworkNode** or **IGoLayoutNetworkLink**.

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

The **GoObject** property will be useful for tailoring the function result to application specific details. However, be aware that some auto-layout classes introduce “artificial” nodes or links, which do not correspond to any top-level **GoObject**. For these nodes and links, **GoObject** returns null.

GoLayout

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

```
public abstract void PerformLayout();
public virtual void RaiseProgress(float progress, String msg) {}
public event GoLayoutProgressEventHandler Progress;
```

The **PerformLayout** method is called to perform the actual layout. Since **PerformLayout** is an abstract method in the **GoLayout** class, the **GoLayout** class is an abstract class; hence, no **GoLayout** object can be created.

The **RaiseProgress** method is called by subclasses of **GoLayout** at various times with a parameter between 0.0f and 1.0f, to indicate the progress through the layout routine. By default, **RaiseProgress** just calls **OnProgress** to call all **Progress** event handlers.

In addition, **GoLayout** defines a constructor and a set of properties to get and set layout's network and document. The constructor creates an instance with a null network and a null document. Until the document is set to a non-null value by setting the **Document** property, **PerformLayout()** will return without doing anything.

GoLayoutForceDirected

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

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

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

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

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

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

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

Note that in some cases the forces will never be reduced to zero, regardless of the number of iterations. For example, consider two unconnected nodes each with an electrical charge. Such nodes will continue moving away from each other forever. For this reason it is frequently desirable to fix one or more nodes prior to performing a force directed layout.

GoLayoutLayeredDigraph

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

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

In the Layering step, all nodes in the input network are assigned to layers. If there is a link L = (U, V), then Layer(U) > Layer(V). Three layering routines are provided: Longest Path Sink Layering, Longest Path Source Layering, and

Optimal Link Length Layering. Figure 15 and Figure 16 illustrate the results of each of these.

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

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

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

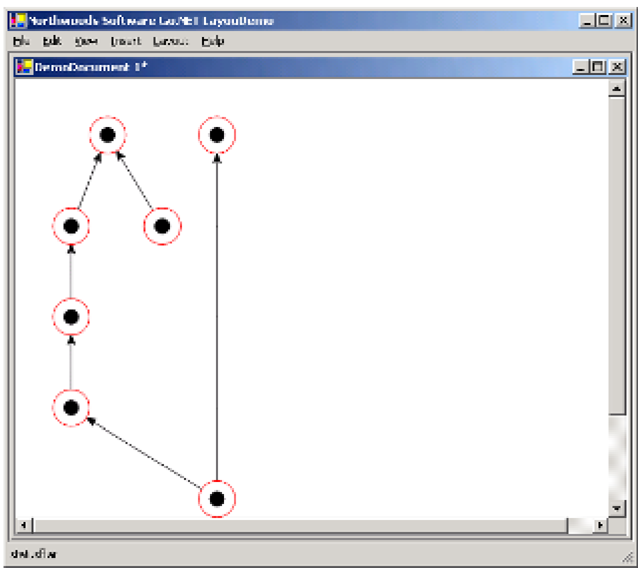


Figure 15. Longest Path Sink Layering

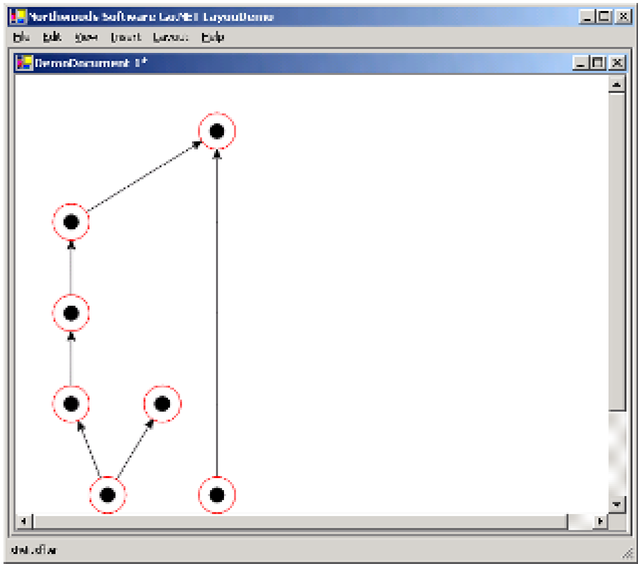


Figure 16. Longest Path Source Layering

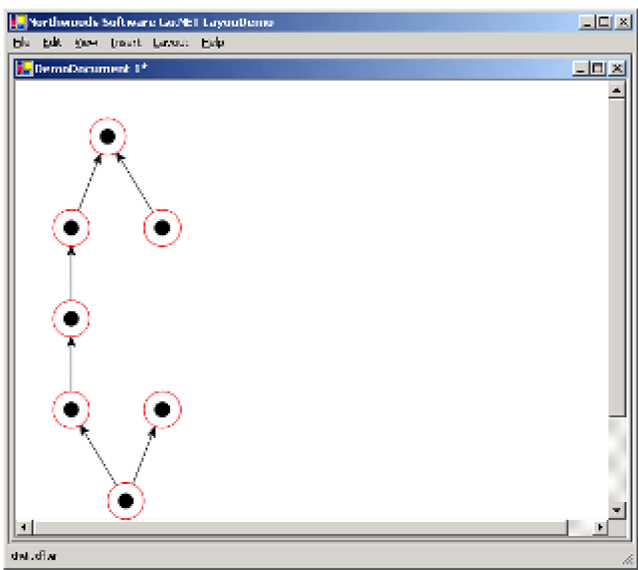


Figure 17. Optimal Link Length Layering

Following the Layering step, there are two minor steps that prepare the network for later steps. The Make Proper step converts the input network into a proper digraph; i.e., artificial nodes and links are introduced into the network such that every link is between nodes in adjacent layers. This has the effect of breaking up long links into a sequence of artificial nodes, and making sure no link will cross through any node.

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

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

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

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

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

GoLayoutTree

The layered-digraph autolayout algorithm is intended to handle any directed graph. However, it is very common to want to lay out subsets of directed graphs that form trees. Furthermore, with a tree layout, there are additional features that are sensible to define, such as the ordering of child nodes and the alignment of the parent node relative to its children. For generality we assume there can be many trees in the network – i.e. it forms a “forest”.

GoLayoutTree.PerformLayout performs several steps:

1. Walk the **GoLayoutTreeNetwork** to build the tree structure(s).
2. Assign various **GoLayoutTreeNode** properties to guide the layout process.
3. Sort the children of each parent node.
4. Associate any annotation objects with each **GoLayoutTreeNode**, such as **GoBalloon** comments.
5. Layout each tree.
6. Position each tree in the document.

“Depth” measurements are along the same direction as the angle at which the tree is growing. “Breadth” measurements are along the perpendicular direction. Thus when the tree is growing horizontally (e.g. **GoLayoutTree.Angle** is zero) the breadth of a node or of a subtree is its height. When the tree is growing vertically (e.g. **GoLayoutTree.Angle** is 90) breadth corresponds to width.

Constructing trees

As with every **GoLayout**, you can restrict the layout to operate on a subset of a **GoDocument** by providing an **IGoLayoutNetwork** that specifies all the nodes and links to consider. But rather than requiring this network to form a strict tree, we allow it to be an arbitrary graph. The **GoLayoutTree.Roots** property lets you specify the **GoObject**s that are to be the roots of the trees. The **GoLayoutTree.Path** property controls the direction in which the layout follows links to go from parent nodes to child nodes. If you don’t specify any **Roots** explicitly, **GoLayoutTree** will try to find reasonable roots from which to start. Afterwards, the **GoLayoutTreeNode.Parent** and **Children** properties will define the actual trees.

Assign node properties

Each **GoLayoutTreeNode** has a number of properties that direct how the tree layout will position the node relative to its parent, its siblings, and its children.

Initially each **GoLayoutTreeNode** will get its properties from the defaults provided by **GoLayoutTree**. But you can override **GoLayoutTree.AssignTreeNodeValues** to specify particular values for particular **GoLayoutTreeNode**s.

Sort children

The **Children** of each **GoLayoutTreeNode** need to be ordered before the layout actually happens. The ordering can be natural, or it can sort the children using an **IComparer**.

Associate balloon comments

The standard behavior is to search for any **GoBalloon** comments that refer to the **GoObject**s that are represented by **GoLayoutTreeNode**s. These comments increase the size of the **GoLayoutTreeNode**. When the nodes and links are actually laid out in the document, the comments associated with those nodes are also laid out near their nodes.

Layout trees

GoLayoutTree.LayoutTree does the tree-layout of **GoLayoutTreeNode**s, respecting various properties such as **GoLayoutTreeNode.Angle**, **Alignment**, **Compaction**, **NodeIndent**, **NodeSpacing**, **LayerSpacing**, and **BreadthLimit**. This does relative positioning of all of the **GoLayoutTreeNode**s.

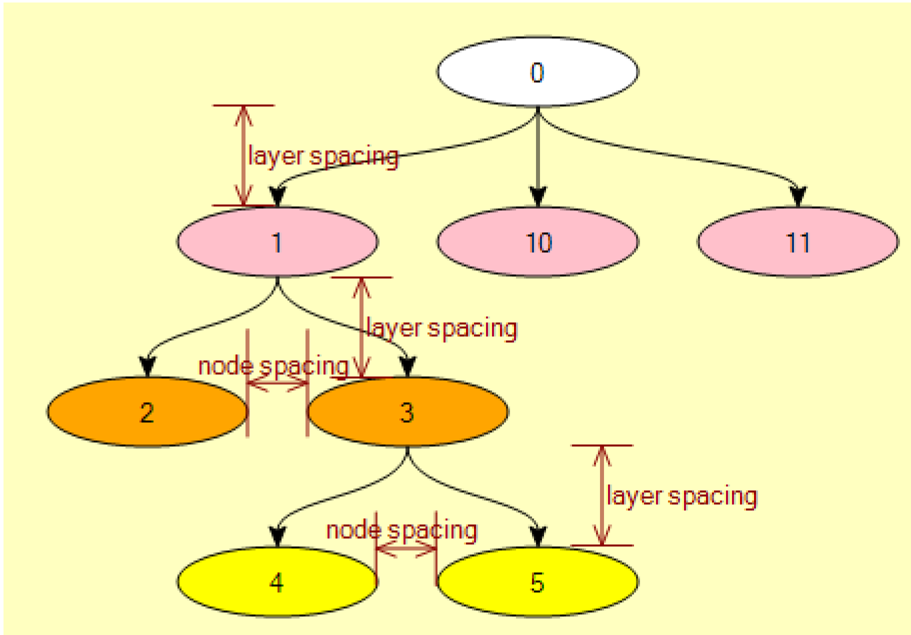
Arrange trees

Finally all of the **GoLayoutTreeNode**s know where they are positioned relative to other **GoLayoutTreeNode**s, but their corresponding **GoObject**s need to be positioned for real. Furthermore, separate trees in the forest may need to be positioned so they do not overlap each other. The **GoLayoutTree.Arrangement** property controls how separate trees are positioned, using the **ArrangementOrigin** and **ArrangementSpacing** properties for guidance.

Comments are also positioned relative to their associated nodes; the **GoLayoutTree.CommentSpacing** and **CommentMargin** properties help guide that work.

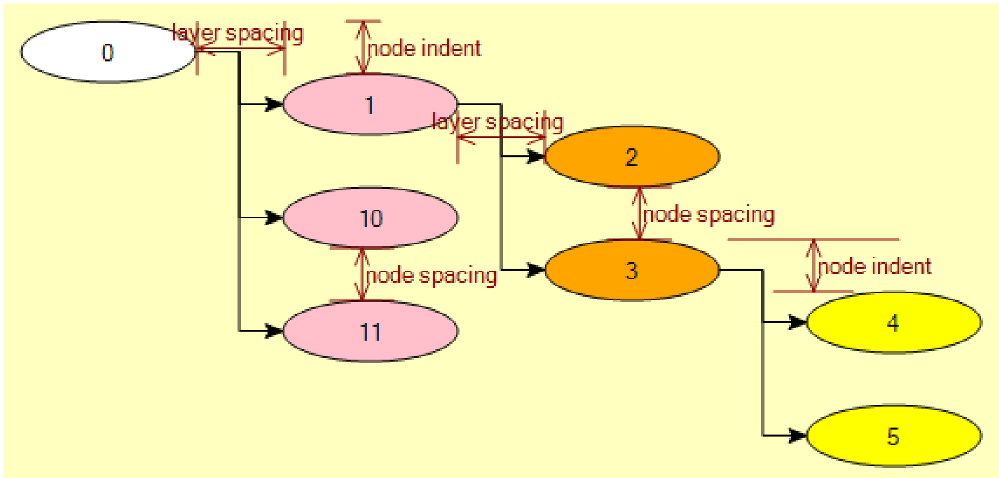
GoLayoutTree and GoLayoutTreeNode Properties

Here is a simple tree, with an **Angle** of 90, so that as the tree grows deeper it gains height. “Depth” corresponds to height; “breadth” corresponds to width. Each node is color-coded by its level (or layer) in the tree. The links have a **Style** that is **GoStrokeStyle.Bezier**. We have marked the effect of the **LayerSpacing** and **NodeSpacing** properties.



(By the way, these examples were constructed using **GoBasicNode**s, with their **LabelSpot** set to **GoObject.Middle**, their **AutoResizes** set to false, and their **Shape.Size** set to 100 x 35.)

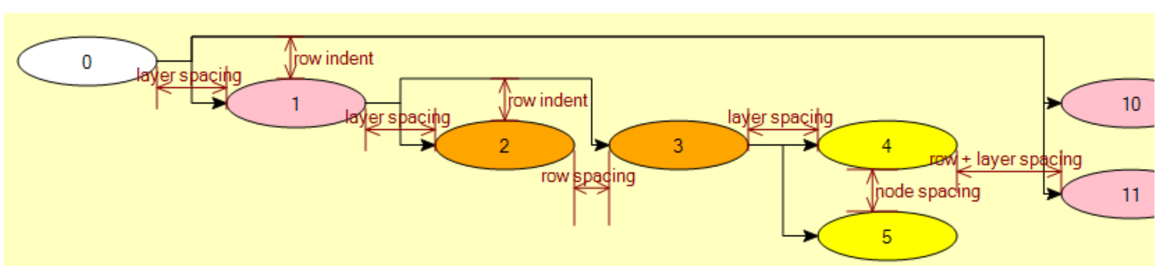
Now for the same tree, but with an **Angle** of zero. Now "depth" corresponds to width and "breadth" corresponds to height. The **LayerSpacing** and **NodeSpacing** properties have same meaning, but with a different orientation. This tree also sets the value of **NodeIndent**, which reserves some initial space at the start or end of each row of children. Furthermore the **Alignment** property has been set to **GoLayoutTreeAlignment.Start**. (The **NodeIndent** property is really only meaningful when the **Alignment** is **Start** or **End**, not when the alignment is a centering one.)



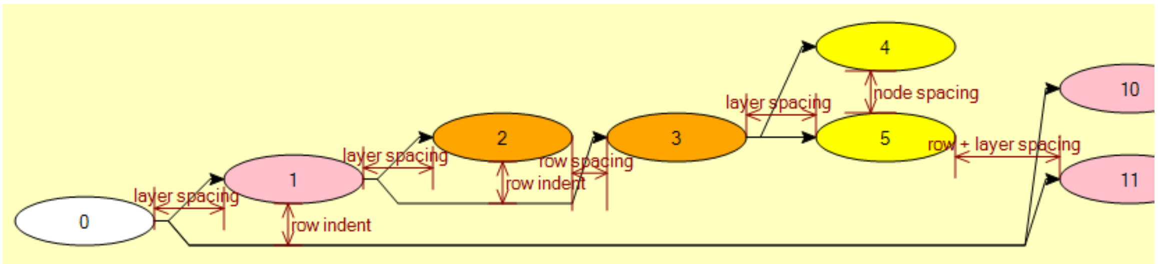
Next is the same tree again, but with the **BreadthLimit** set. This property tries to limit the breadth of a subtree to the given value. (The default value of zero means "no limit" -- all of the children are laid out in a single row, no matter how broad that subtree gets.)

When there is a limit on the breadth, and there isn't enough room to position all of the subtrees (descendent nodes), the auto-layout will position children in additional rows. In this next example, notice that nodes #10 and #11 have been placed in a second row of pink nodes. Furthermore, note that node #1 has two rows of children (for nodes #2 and #3). But there is enough room for the children of node #3 to just place them in a single row.

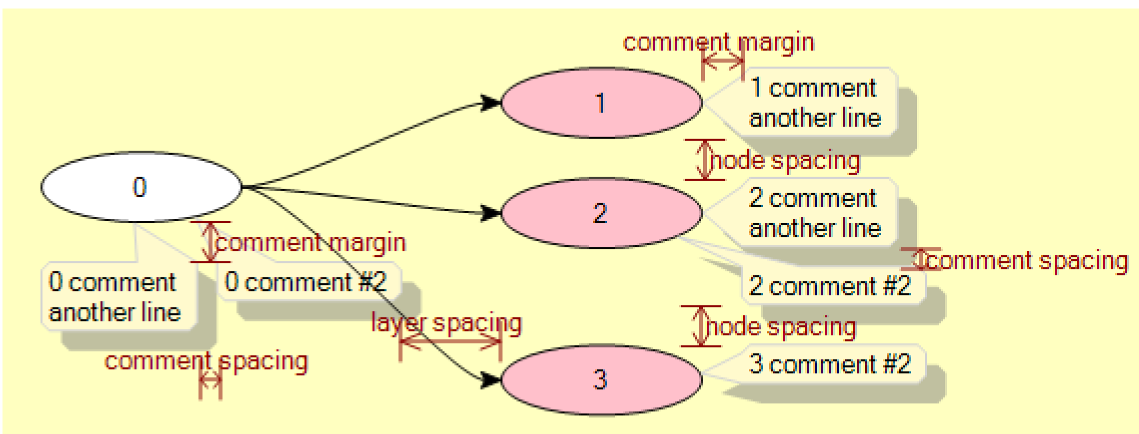
When specifying a **BreadthLimit**, the **Alignment** should be **Start** or **End**. The **RowIndent** property reserves room for the links that are routed around earlier rows to get to following rows. (The default value for **RowIndent** of 10 is fine for most applications.) The **RowSpacing** property specifies the distance between rows. After the last row, additional **LayerSpacing** room is reserved, to increase the visual distinction between a group of rows for one parent and a different layer.



When the **Alignment** is **GoLayoutTreeAlignment.End**, one gets the same positioning, but in the opposite direction. Note also that these links have simple straight segments, whereas the previous screen shot has **Orthogonal** links.

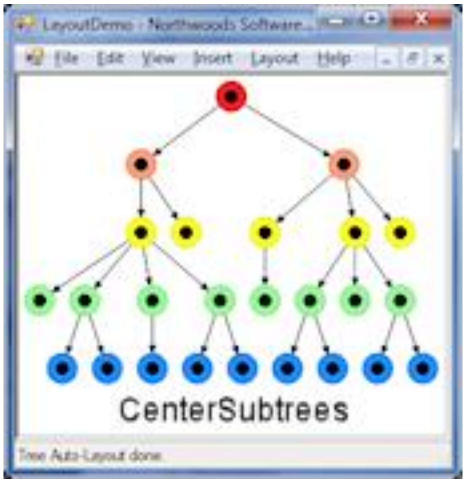


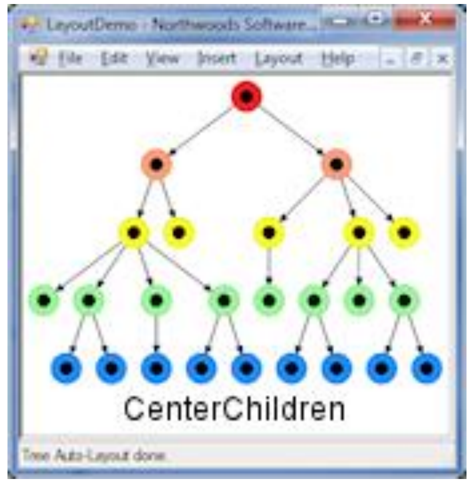
Finally, when you have **GoBalloon** comments associated with nodes, you can control the spacing by setting the **CommentMargin** and **CommentSpacing** properties.

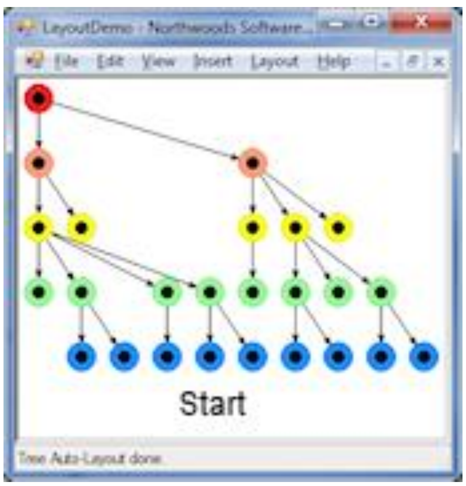


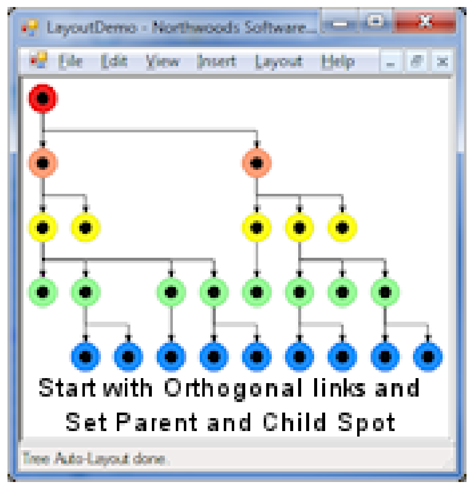
Comparison of Alignments

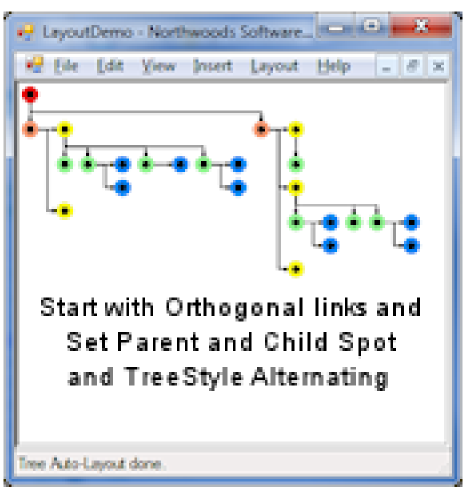
Here are examples of the same graph using different values of **GoLayoutTreeAlignment.**

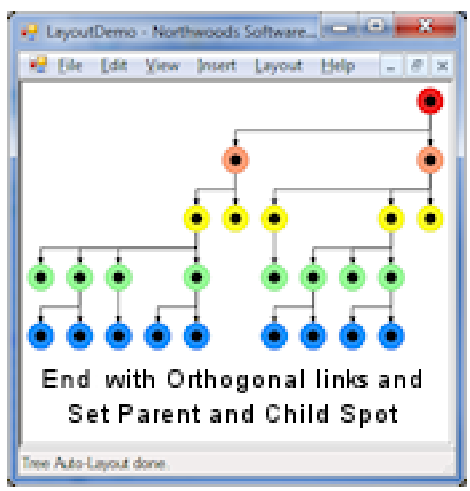

LayoutDemo - Northwoods Software...
File Edit View Insert Layout Help
CenterSubtrees
Tree Auto-Layout done.

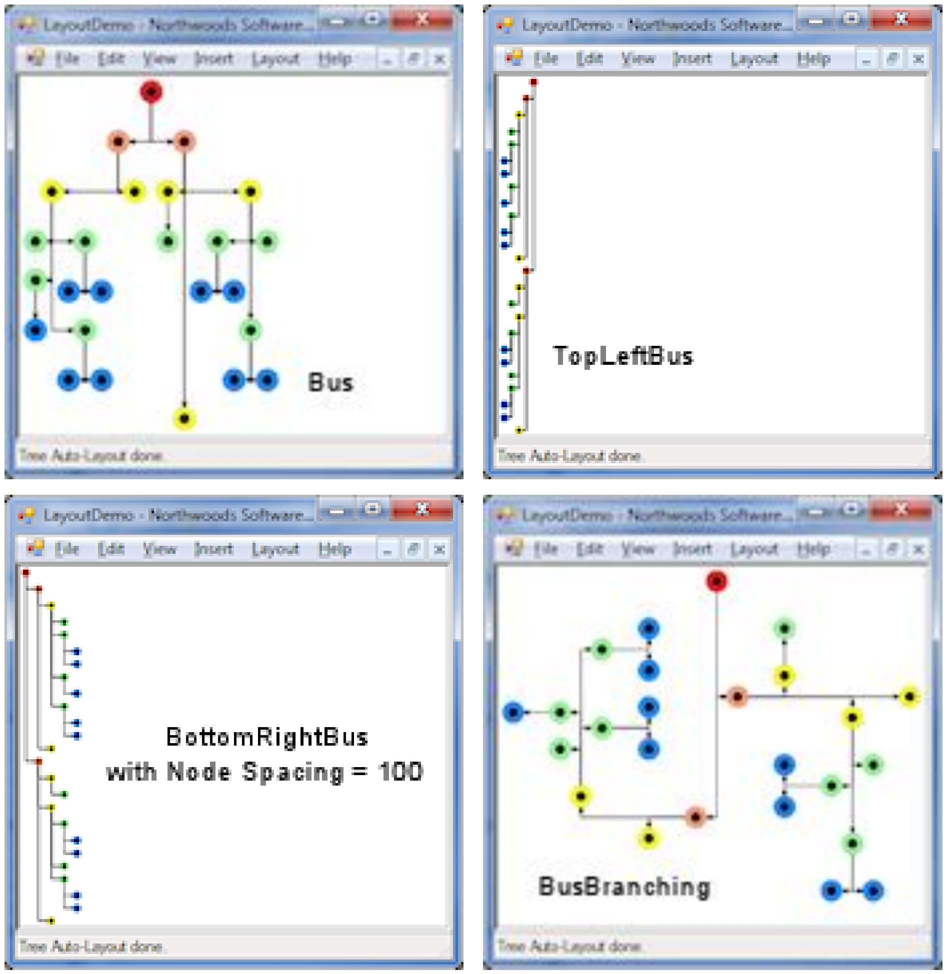

LayoutDemo - Northwoods Software...
File Edit View Insert Layout Help
CenterChildren
Tree Auto-Layout done.


LayoutDemo - Northwoods Software...
File Edit View Insert Layout Help
Start
Tree Auto-Layout done.


LayoutDemo - Northwoods Software...
File Edit View Insert Layout Help
Start with Orthogonal links and
Set Parent and Child Spot
Tree Auto-Layout done.


LayoutDemo - Northwoods Software...
File Edit View Insert Layout Help
Start with Orthogonal links and
Set Parent and Child Spot
and TreeStyle Alternating
Tree Auto-Layout done.


LayoutDemo - Northwoods Software...
File Edit View Insert Layout Help
End with Orthogonal links and
Set Parent and Child Spot
Tree Auto-Layout done.



Note that **GoLayoutTreeAlignment .BusBranching** only works well with **GoLayoutTreeStyle.Layered**.

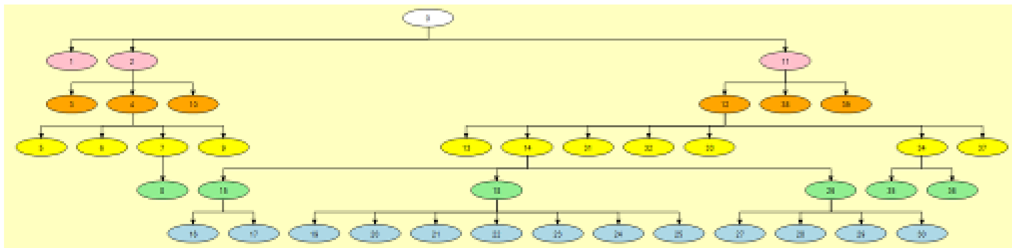
GoLayoutTree Styles

The normal **GoLayoutTree.Style** is **GoLayoutTreeStyle.Layered.** This style has each node lay out its children in the manner specified by the properties set on **GoLayoutTree**, as described in the previous section. However, you can make simple customizations of the tree layout by specifying other **Style**s.

GoLayoutTreeStyle.LastParents is a commonly used style to have the fringes of the tree be laid out differently from the whole tree. A "Last Parent" is a node that is a parent (i.e. there is at least one child) but that does not have any grand-children.

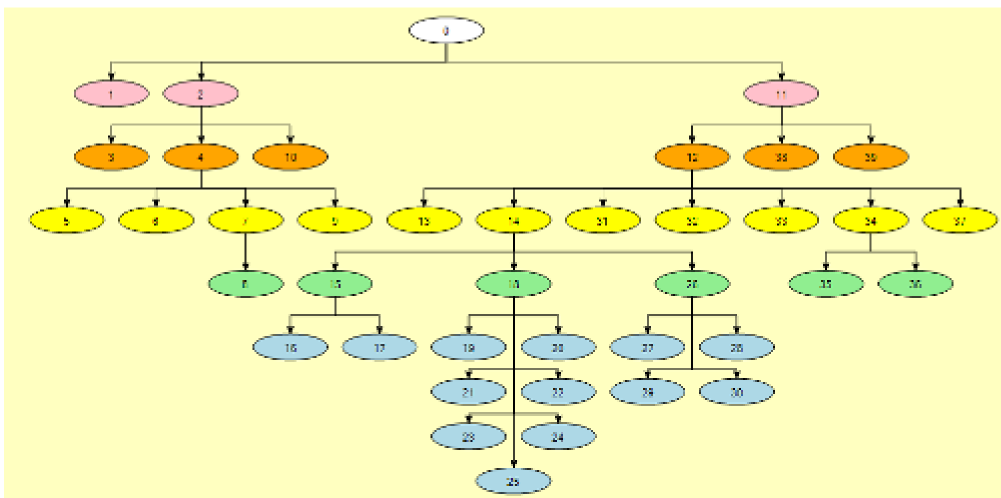
Here is a tree laid out with the properties:

```
layout.Angle = 90
```



The same tree, but with

```
layout.Style = GoLayoutTreeStyle.LastParents
layout.AlternateDefaults.Angle = 90
layout.AlternateDefaults.BreadthLimit = 100
```



Note how the leaf nodes, when there were no siblings with children, are arranged to take much less breadth. The alternate **BreadthLimit** was a bit larger than twice the breadth of the nodes plus the **NodeSpacing**, causing the nodes to be arrayed in two columns.

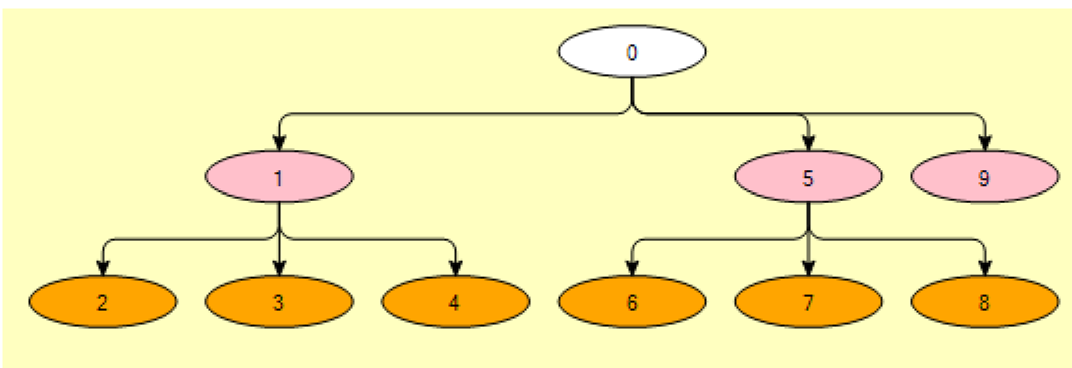
GoLayoutTree actually has two sets of default node properties, one held by **RootDefaults** and one held by **AlternateDefaults**. All of the **GoLayoutTree** properties relating to nodes just delegate to the corresponding property on **RootDefaults**.

When the **Style** is **LastParents**, the **AlternateDefaults** properties are used to initialize all of the "last parent" nodes' properties. All the other nodes inherit their properties from their parent node, just as with the normal style. Except root nodes, of course, inherit their properties from the **GoLayoutTree.RootDefaults**.

When the **Style** is **Alternating**, every node inherits from their grand-parent node. However, root nodes get their properties from **GoLayoutTree.RootDefaults**, as always, and the immediate children of root nodes get their properties from **GoLayoutTree.AlternateDefaults**.

When you set a property relating to nodes on **GoLayoutTree**, you may need to remember to also set that property on the **GoLayoutTree.AlternateDefaults**, as the example above did for the **Angle** property.

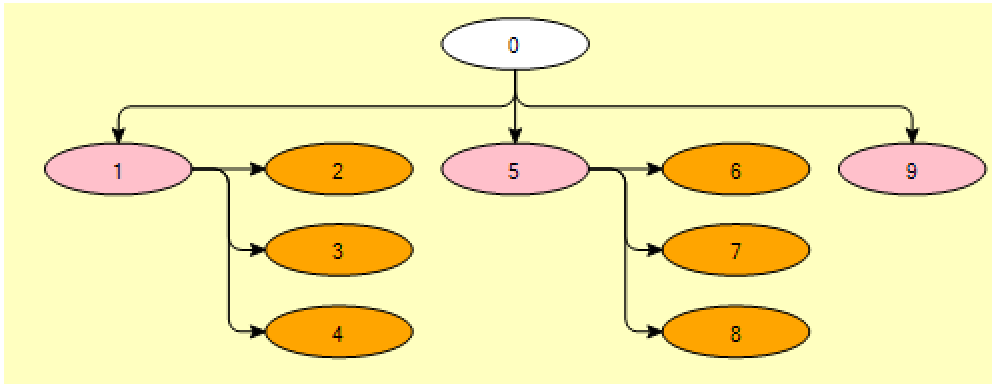
Here's another tree, also with an **Angle** of 90:



Now we'll change the **Angle** and **Alignment** of the last parents:

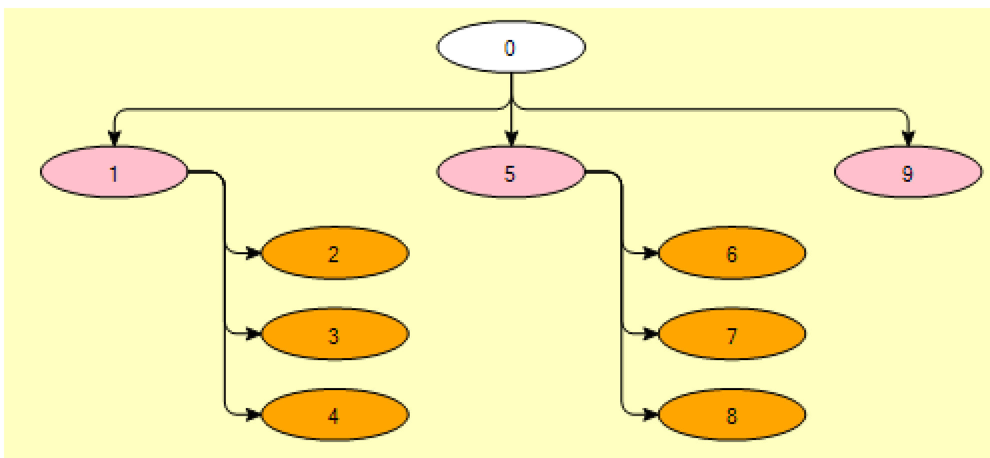
```
layout.Style = GoLayoutTreeStyle.LastParents
layout.AlternateDefaults.Angle = 0
layout.AlternateDefaults.Alignment = GoLayoutTreeAlignment.Start;
```

Note below how node #0, with an **Angle** of 90, is growing downwards, but each of the LastParent nodes (#1 and #5) are growing towards the right, because the alternate angle is zero.



To shift those nodes #2,3,4 and #6,7,8 down, we can specify a **NodeIndent** for those last parent nodes (#1 and #5). Remember that the **NodeIndent** controls how much initial space there is in a row. Since nodes #1 and #5 are growing towards the right, the row actually extends vertically.

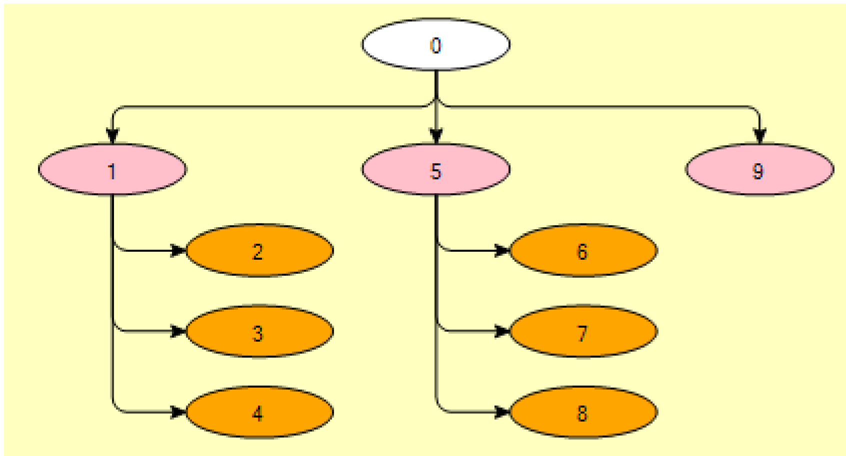
```
layout.AlternateDefaults.NodeIndent = 55
```



For nodes #1 and #5, the **Angle** is now 0, meaning towards the right, so if we want to reduce the horizontal space between #1 and #2,3,4, we need to reduce the **LayerSpacing** for those “last parent” nodes. To avoid having the links snake around, we’ll also set the **GoPort.FromSpot** for those parent nodes to be at the middle of the bottom of the node.

Caution: using the **PortSpot** and **ChildPortSpot** properties is possible only for ports on nodes with a single **Port**, such as **GoBasicNode**, **GoIconicNode**, and **GoBoxNode**. If you are using a node such as **GoTextNode** or **GoMultiTextNode,** where there are many small ports positioned at particular places on each node, you will need to programmatically relink to connect to the appropriate port.

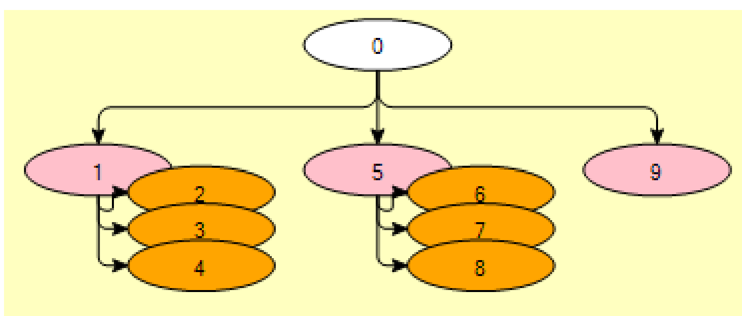
```
layout.AlternateDefaults.LayerSpacing = 0
layout.AlternateDefaults.PortSpot = GoObject.MiddleBottom
```



If we had wanted to change where links connected to the child nodes, we could change the value of **ChildPortSpot**.

You can even get them to overlap by using negative values for spacing.

```
layout.AlternateDefaults.LayerSpacing = -30
layout.AlternateDefaults.NodeSpacing = -10
layout.AlternateDefaults.NodeIndent = 15
```



Of course you will need to adjust the spacing and indentation sizes to match the sizes of your nodes.

Note that the above examples set the **PortSpot** to a particular object spot, in this case **MiddleBottom**. This causes the **GoPort.FromSpot** of the parent node to be set to that particular value. The default value of **GoObject.NoSpot** would cause

GoLayoutTree to assign a **GoPort.FromSpot** value that is appropriate for the **Angle**.

You can also specify the **ChildPortSpot** property to cause **GoLayoutTree** to assign the **GoPort.ToSpot** of the child nodes. The default value of **NoSpot** causes it to assign an appropriate spot given the **Angle** to all of the children's ports.

If your nodes have ports that already have port spots that you want to keep, set **SetsPortSpot** and/or **SetsChildPortSpot** to false.

Automatic Layout inside SubGraphs

An automatic layout of a document only arranges the top-level nodes and links. If you are using **GoSubGraph** nodes, you will notice that the interiors of subgraphs do not get laid out at all.

If you want to perform an automatic layout of the children of a **GoSubGraph**, you will need to create and initialize a **GoLayoutNetwork** explicitly, rather than depending on the default behavior that uses the whole **GoDocument**. For example, the following code performs an automatic layout of the **GoSubGraph** that is the primary selection:

VB.NET:

```
Dim sg As GoSubGraph = myView.Selection.Primary as GoSubGraph
If Not sg Is Nothing Then
  Dim layout As GoLayoutLayeredDigraph = new GoLayoutLayeredDigraph()
  layout.Document = myView.Document
  layout.Network = layout.CreateNetwork()
  layout.Network.AddNodesAndLinksFromCollection(sg, true)
  ' maybe set other properties too . . .
  layout.PerformLayout()
End If
```

C#:

```
GoSubGraph sg = myView.Selection.Primary as GoSubGraph;
if (sg != null) {
  GoLayoutLayeredDigraph layout = new GoLayoutLayeredDigraph();
  layout.Document = myView.Document;
  layout.Network = layout.CreateNetwork();
  layout.Network.AddNodesAndLinksFromCollection(sg, true);
  layout.PerformLayout();
}
```

You can accomplish a thorough automatic layout of a document containing subgraphs by iterating over the whole document and recursively descending into each **GoSubGraph**. You will need to perform an automatic layout of each of the children before doing the automatic layout of the graph, because the layout of the children of a **GoSubGraph** will change the size of that node.

C#:

```
public void LayoutGraph(IGoCollection coll, GoDocument doc) {
  foreach (GoObject obj in coll) {
    GoSubGraph sg = obj as GoSubGraph;
    if (sg != null) {
      bool expanded = sg.IsExpanded;
      if (!expanded) sg.Expand();
      LayoutGraph(sg, doc);
      if (!expanded) sg.Collapse();
    }
  }
  GoLayoutLayeredDigraph layout = new GoLayoutLayeredDigraph();
  layout.Document = doc;
  layout.Network = layout.CreateNetwork();
  layout.Network.AddNodesAndLinksFromCollection(coll, true);
  layout.PerformLayout();
}
```

Note how this code constructs new **GoLayoutLayeredDigraphNetwork** instances for each subgraph, each initialized with the nodes and links in the argument collection, until the final network encompasses all the top-level nodes and links in the document.

This automatic layout of subgraphs depends on the links that describe the subgraphs being children of the **GoSubGraph** nodes—they must not belong to the document as top-level objects. If you find that all of the children of a **GoSubGraph** are being laid out in a single straight line, then the problem is due to how you constructed the **GoSubGraph**. Unlike the normal custom of adding newly created instances of **GoLink** or **GoLabeledLink** to the document's **LinksLayer**, you should add links to the **GoSubGraph** that is the first parent that both nodes have in common. The static/shared **GoSubGraph.Reparent…** methods of may be useful in this regard.

5. Advanced Options

This section provides details regarding customizing the **GoLayout** routines. Referring to the **GoLayout** API Reference files will be helpful when reading this section.

GoLayoutForceDirected

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

When a **GoLayoutForceDirected** is constructed, it uses a default of 1000. To change the default, set the static/shared property `DefaultMaxIterations`.

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

```
protected virtual float SpringStiffness(GoLayoutNetworkLink pLink);
protected virtual float SpringLength(GoLayoutNetworkLink pLink);
protected virtual float ElectricalCharge(GoLayoutNetworkNode pNode);
protected virtual float ElectricalFieldX(PointF xy);
protected virtual float ElectricalFieldY(PointF xy);
protected virtual float GravitationalMass(GoLayoutNetworkNode pNode);
protected virtual float GravitationalFieldX(PointF xy);
protected virtual float GravitationalFieldY(PointF xy);
protected virtual bool IsFixed(GoLayoutNetworkNode pGoNode);
```

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

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

VB.NET:

```
Protected Overrides Function ElectricalFieldX(ByVal xy As PointF)
     As Single
   Dim border As Single = 50
   Dim min As Single = 0
   Dim max As Single = Document.Size.Width
   If xy.X <= 0 Then
     Return 300
   End If
   If xy.X < min + border Then
     Return (300 / ((min - xy.X) * (min - xy.X)))
   End If
   Return 0
 End Function

 Protected Overrides Function ElectricalFieldY(ByVal xy As PointF)
     As Single
   Dim border As Single = 50
   Dim min As Single = 0
   Dim max As Single = Document.Size.Height
   If xy.Y <= 300 Then
     Return 300
   End If
   If xy.Y < min + border Then
     Return (300 / ((min - xy.Y) * (min - xy.Y)))
   End If
   Return 0
 End Function
```

C#:

```
protected override float ElectricalFieldX(PointF xy) {
  float border = 50;
  float min = 0;
  float max = Document.Size.Width;
  if (xy.X <= 0)
    return 300;
  if (xy.X < min + border) {
    return (300 / ((min - xy.X) * (min - xy.X)));
  }
  return 0;
}

protected override float ElectricalFieldY(PointF xy) {
  float border = 50;
  float min = 0;
  float max = Document.Size.Height;
```

```
  if (xy.Y <= 300)
    return 300;
  if (xy.Y < min + border) {
    return (300 / ((min - xy.Y) * (min - xy.Y)));
  }
  return 0;
}
```

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

By adjusting the values of the **SpringLength** and **SpringStiffness**, one can achieve a number of sophisticated results. For example, by increasing the **SpringLength** between red and green nodes, it is possible to group the nodes by color as illustrated in Figure 18. Keep in mind that the colors of nodes are part of the LayoutDemo application, and not a part of the **GoLayout** code itself.

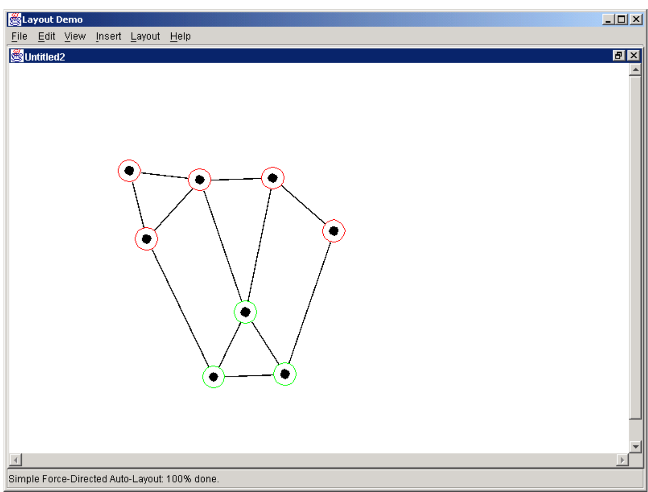


Figure 18. Sample graph after adjusting spring length and thickness

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

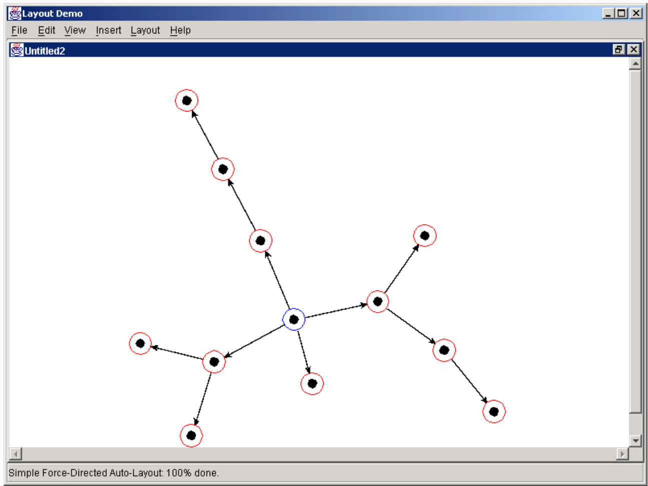


Figure 19. Sample graph before applying gravity field

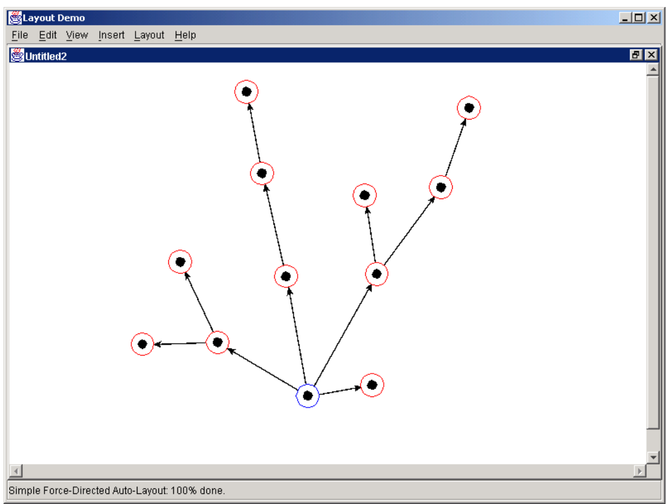


Figure 20. Sample graph after applying gravity field

In both networks, the blue root node is fixed. In the network of Figure 19, no gravitational field has been set. In the network of Figure 20, a slight gravitational field pointing upward has been added, which results in a more natural layout for a tree.

GoLayoutForceDirected has two other methods that can be overridden:

```
protected virtual bool UpdatePositions();
protected virtual void LayoutNodesAndLinks(bool isfinal);
```

The **UpdatePositions** method is used each iteration to calculate the forces on each node and to move the node to its new position; it returns true if additional iterations are needed to find a local equilibrium. Overriding the **UpdatePositions** method can be used to add entirely new forces to the layout.

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

GoLayoutLayeredDigraph

Most of the customization available in the **GoLayoutLayeredDigraph** class is accessed through its properties:

```
  public int LayerSpacing { get; set; }
  public int ColumnSpacing { get; set; }
  public GoLayoutDirection DirectionOption { get; set; }
  public GoLayoutLayeredDigraphCycleRemove CycleRemoveOption { get;
set; }
  public GoLayoutLayeredDigraphLayering LayeringOption { get; set; }
  public GoLayoutLayeredDigraphInitIndices InitializeOption { get; set;
}
  public int Iterations { get; set; }
  public GoLayoutLayeredDigraphAggressive AggressiveOption { get; set;
}
  public GoLayoutLayeredDigraphPack PackOption { get; set; }
```

See the **GoLayout** Class Reference Guide for a detailed description of these properties. The **LayerSpacing** and **ColumnSpacing** properties determine the minimum space (in logical units) between nodes in adjacent layers and columns. Generally, since nodes have width and height, additional space is reserved around nodes. However, **ColumnSpacing** will also determine the minimum space between long links that are drawn parallel and adjacent to one another, as illustrated in Figure 21.

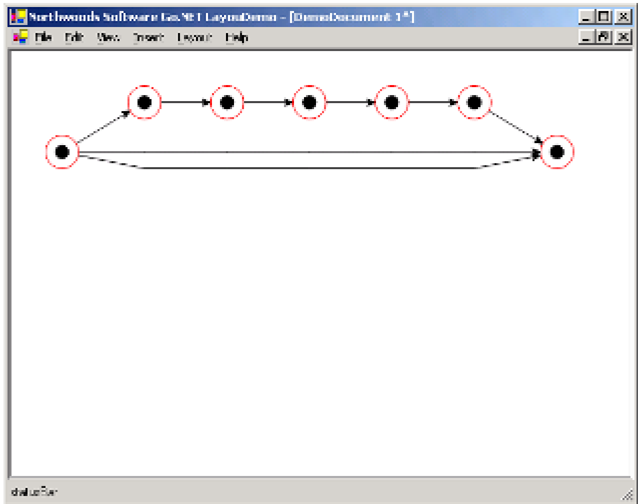


Figure 21. A graph showing the use of ColumnSpacing

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

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

```
protected virtual void RemoveCycles();
protected virtual void AssignLayers();
protected virtual void MakeProper();
protected virtual void InitializeIndices();
protected virtual void InitializeColumns();
protected virtual void ReduceCrossings();
protected virtual void StraightenAndPack();
protected virtual void LayoutNodesAndLinks();
```

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

One method that is more often overridden is **AssignLayers**. A common need is to make sure that certain nodes are assigned to certain layers. For example, if one wants to put all of the nodes that have no links coming into them at one side of the graph, one could override **AssignLayers** to first call the base method to get the standard work done, and then one could reassign the layer of all of those terminal nodes to be equal to the maximum layer.

The second category of methods override spacing methods:

```
protected virtual int NodeMinLayerSpace(GoLayoutNetworkNode pNode);
protected virtual int NodeMinColumnSpace(GoLayoutNetworkNode pNode);
```

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

The final category of methods override layering methods:

```
protected virtual int LinkMinLength(GoLayoutNetworkLink pLink);
protected virtual float LinkLengthWeight(GoLayoutNetworkLink pLink);
```

The **LinkMinLength** method indicates the minimum length of the link, measured in layers. For example, if link L = (U,V), then Layer(U) – Layer(V) ≥ LinkMinLength(L). The default implementation gives multi-links (multiple links between the same pairs of nodes) a minimum length of 2, and all other links a minimum length of 1. This ensures that multi-links are drawn distinctly, illustrated in Figure 22.

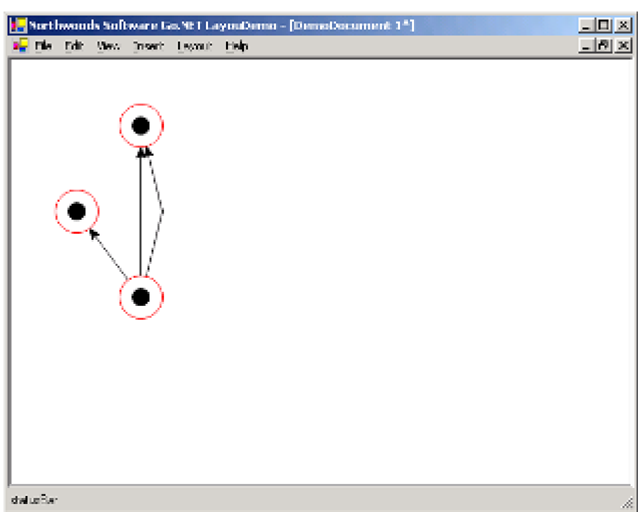


Figure 22. Example use of the LinkMinLength method

The Layout Demo sample overrides the **LinkMinLength** method as follows:

VB.NET:

```
Protected Overrides Function LinkMinLength(
     ByVal pLink As GoLayoutNetworkLink) As Integer
  Dim pFromNode As GoLayoutNetworkNode = pLink.FromNode
  Dim pToNode As GoLayoutNetworkNode = pLink.ToNode

  If Not pFromNode.GoObject Is Nothing AndAlso
       Not pToNode.GoObject Is Nothing Then
    Dim fromColor As Pen = CType(pFromNode.GoObject,
                                 BasicLayoutNode).Pen
    Dim toColor As Pen = CType(pToNode.GoObject, BasicLayoutNode).Pen
    If fromColor Is toColor Then
      Return 1 * MyBase.LinkMinLength(pLink)
    Else
      Return 2 * MyBase.LinkMinLength(pLink)
    End If
  End If
  Return MyBase.LinkMinLength(pLink)
End Function
```

C#:

```
protected override int LinkMinLength(GoLayoutNetworkLink pLink) {
  GoLayoutNetworkNode pFromNode = pLink.FromNode;
  GoLayoutNetworkNode pToNode = pLink.ToNode;

  if ((pFromNode.GoObject != null) && (pToNode.GoObject != null)) {
    Pen fromColor = ((BasicLayoutNode)(pFromNode.GoObject)).Pen;
    Pen toColor = ((BasicLayoutNode)(pToNode.GoObject)).Pen;
    if (fromColor == toColor) {
      return 1 * base.LinkMinLength(pLink);
    } else {
      return 2 * base.LinkMinLength(pLink);
    }
  }
  return base.LinkMinLength(pLink);
}
```

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

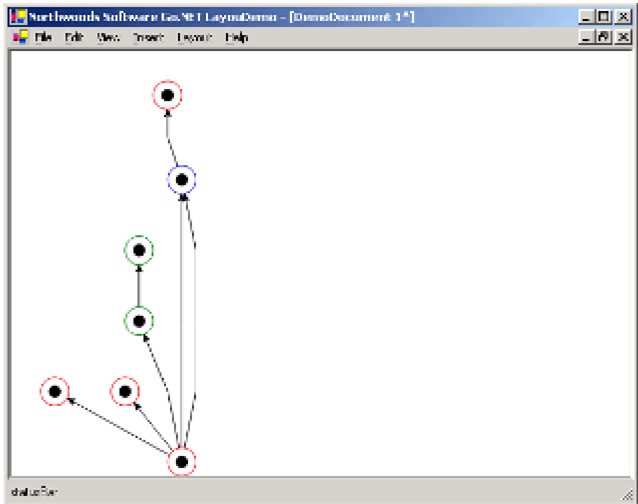


Figure 23. Another example use of the LinkMinLength method

The **LinkLengthWeight** method indicates the weight of the link. The Optimal Link Length Layering routine assigns nodes to layers such that the sum (Layer(U) – Layer(V)) * LinkLengthWeight(L) over all L = (U,V) is minimized. By default, all links have a LinkLengthWeight of 1. The **LinkLengthWeight** method can be overridden to increase the "importance" of a link, which means the link will be kept shorter. For example, compare the networks of Figure 24 and Figure 23.

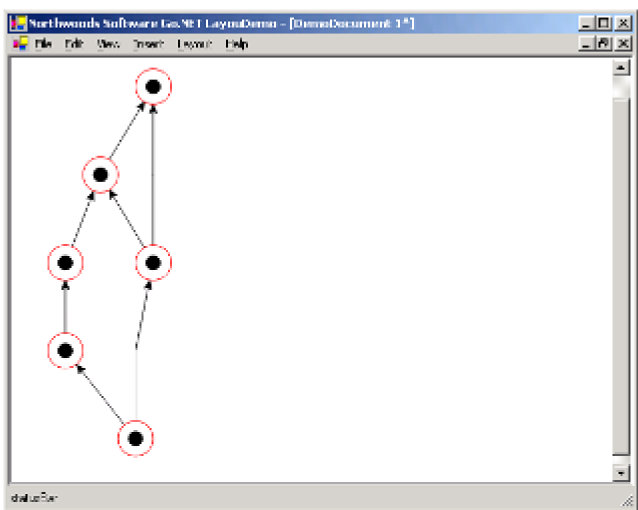


Figure 24. Graph before using LinkLengthWeight

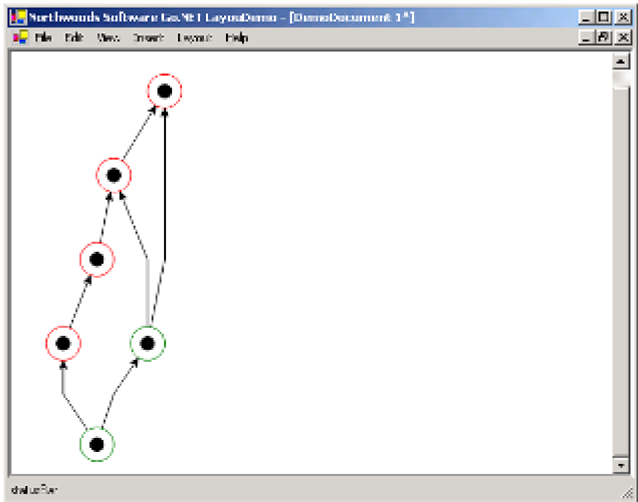


Figure 25. Graph after using LinkLengthWeight

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

Tree Layout

Most of the properties that govern how trees are laid out are properties of **GoLayoutTreeNode**. Thus each node of the tree can have its own **Angle**, **Alignment**, **NodeSpacing**, et al.

By default all **GoLayoutTreeNode**s get their properties initialized from either **GoLayoutTree.RootDefaults** or **GoLayoutTree.AlternateDefaults**. (Note again that for simplicity and ease of use, the "node" properties on **GoLayoutTree** actually get and set the same-named properties on **GoLayoutTree.RootDefaults**.)

AssignTreeNodeValues

But if you want to specify some properties for particular **GoLayoutTreeNode**s, you cannot assign the properties of the whole **GoLayoutTree**. Instead you need to override the **AssignTreeNodeValues** method. This method is called for each **GoLayoutTreeNode**, so you can decide if it represents a **GoNode** that you care about, and if so, what properties to assign to the argument **GoLayoutTreeNode**.

```
protected override void AssignTreeNodeValues(GoLayoutTreeNode n) {
```

```
    base.AssignTreeNodeValues(n);
    MyNode x = n.GoObject as MyNode;
    // a MaxGenerationCount of 1 means there are children but no
    // grandchildren
    if (x != null && x.ClosePack && n.MaxGenerationCount == 1) {
      n.NodeSpacing = 10;
      n.RowSpacing = 20;
      int cols = (int)Math.Ceiling(Math.Sqrt(n.ChildrenCount));
      // use n.Width if Angle is 90 or 270
      n.BreadthLimit = (n.Height+n.NodeSpacing)*cols;
      if (cols >= 3) n.Alignment = GoLayoutTreeAlignment.Start;
    }
  }
```

As this example shows, you can look at the **GoLayoutTreeNode.GoObject** to decide whether to provide any custom property values.

Note also that you can make use of the statistical properties of **GoLayoutTreeNode**, such as **Level**, **DescendentCount**, **MaxGenerationCount**, and **MaxChildrenCount** in order to make decisions regarding how you want that tree to lay out.

Sorting

Each **GoLayoutTreeNode** has the chance to specify the ordering of its children. By default that order is just the order in which the **Children** are listed.

If you set the **Sorting** property to **GoLayoutTreeSorting.Forwards**, and if the **GoObject** associated with each **GoLayoutTreeNode** is an **IGoLabeledPart**, the **GoLayoutTree.SortTreeNodeChildren** method will sort the array of **Children** by the **IGoLabeledPart.Text** strings with a case-insensitive comparison. Since **GoNode** implements **IGoLabeledPart**, this will work to sort most nodes by the text of their **Label**s.

However you can provide a custom **IComparer** for a **GoLayoutTreeNode** by assigning an **IComparer** to the **GoLayoutTree.Comparer** property.

```
[Serializable]
public class FlagsComparer : System.Collections.IComparer {
  public FlagsComparer() { }

  public int Compare(Object x, Object y) {
    GoLayoutTreeNode m = (GoLayoutTreeNode)x;
    GoLayoutTreeNode n = (GoLayoutTreeNode)y;
    IGoGraphPart a = m.GoObject as IGoGraphPart;
    IGoGraphPart b = n.GoObject as IGoGraphPart;
    if (a != null) {
      if (b != null) {
        int aflags = a.UserFlags;
```

```
        int bflags = b.UserFlags;
        return (aflags < bflags) ? -1 : ((aflags == bflags) ? 0 : 1);
      } else {
        return 1;
      }
    } else {
      if (b != null)
        return -1;
      else
        return 0;
    }
  }
}
```

Then you could either use this comparer for many tree nodes:

```
    GoLayoutTree layout = new GoLayoutTree();
    layout.Comparer = new FlagsComparer();
    . . .
```

Or you could assign it for particular tree nodes in an override of **AssignTreeNodeValues**. For example, to change how all of the parent nodes in the third layer sort their children:

```
  protected override void AssignTreeNodeValues(GoLayoutTreeNode n) {
    base.AssignTreeNodeValues(n);
    if (n.Level == 2) {
      n.Sorting = GoLayoutTreeSorting.Reverse;
      n.Comparer = new FlagsComparer();
    }
  }
```

As another example, if you want to customize the **LayerSpacing** or **NodeIndent** for "last parent" nodes based on the size of the node, you can do something like:

```
  protected override void AssignTreeNodeValues(GoLayoutTreeNode n) {
    base.AssignTreeNodeValues(n);
    if (n.MaxGenerationCount == 1) {
      n.NodeIndent = n.Height+20;
      n.LayerSpacing = 20-n.Width/2;
    }

  }
```

Port Spots

To improve the tree layout of single-port nodes such as **GoBasicNode** and **GoIconicNode**, the **GoLayoutTree.SetPortSpots** method sets the values of

GoPort.FromSpot and **GoPort.ToSpot** to force links to come out or go into the ports in certain directions at certain locations, according to the **GoLayoutTree.Angle**.

So for a tree whose **Angle** is zero and whose **Path** is the default **GoLayoutTreePath.Destination**, the **GoPort.FromSpot** should normally be **GoObject.MiddleRight**, and the **GoPort.ToSpot** should normally be **GoObject.MiddleLeft**. This is usually the best to reduce the likelihood of links crossing over adjacent nodes.

However, you can easily either avoid setting any port spots or set them to node-specific values. Set **SetsPortSpot** to false to avoid setting the port spot for the parent node; set **SetsChildPortSpot** to false to avoid setting the port spot for the children. To specify a particular spot, set the **PortSpot** and/or **ChildPortSpot** properties, respectively. (As with all tree node properties, you can set this either in the **GoLayoutTree.RootDefaults** or **GoLayoutTree.AlternateDefaults** to cover all of the nodes, or you can set this for particular **GoLayoutTreeNode**s in an override of **GoLayoutTree.AssignTreeNodeValues**.)

Remember that the **PortSpot** and **ChildPortSpot** properties are only effective if the port can support links coming in or going out at the desired spots. The ports on **GoBasicNode, GoIconicNode,** and **GoBoxNode** do support links coming in or going out at any direction. For most nodes with multiple ports, the ports are designed to go at specific directions, so setting the port spots would not make sense. Thus the **SetPortSpots** method sets port spots only for those ports that are the only **GoPort** for its node.

Threads

GoLayout does not explicitly make use of any threads. All of the computation is performed on the thread which calls **PerformLayout**. That means that if your Windows Forms application has a button or menu item command that constructs a **GoLayout** and then calls **PerformLayout**, the application user interface is blocked until the **PerformLayout** call returns.

You can easily put the layout computation in a worker thread. This allows your application to remain responsive, and lets you be able to implement commands that abort the layout computation. You just need to set **GoLayout.View** to your **GoView** so that **Progress** events and positioning of document objects can be done on that view's thread.

C#:

```
  // the event handler to start a layout
  private void button1_Click(object sender, System.EventArgs e) {
    if (myThread == null) {
      myThread = new Thread(new ThreadStart(this.LayoutAsynch));
```

```
    myThread.Start();
  }
}

private Thread myThread = null;  // the thread running PerformLayout

private void LayoutAsynch() {
  GoLayoutForceDirected layout = new GoLayoutForceDirected();
  layout.Progress += new GoLayoutProgressEventHandler(layout_Progress);
  layout.Document = goView1.Document;
  layout.View = goView1;  // the Control to Invoke when InvokeRequired
  layout.PerformLayout();
  myThread = null;
}

void layout_Progress(object sender, GoLayoutProgressEventArgs e) {
  // substitute your own informational mechanisms...
  SetStatusMessage(((int)(e.Progress*100)).ToString() + "% done");
}

// an event handler to cancel the layout
private void goView1_KeyDown(object sender, KeyEventArgs evt) {
  if (evt.KeyCode == Keys.Escape) {
    if (myThread != null) {  // don't have multiple layouts running
      myThread.Abort();
      myThread = null;
      SetStatusMessage("layout aborted");
    }
  }
}
```