Starting guide for using graph layout with JViews Diagrammer

Question

Do you have a starting guide that list those layouts, and describe the main parameters to use them?

Answer

IBM ILOG JViews Diagrammer provides a set of graph layout algorithms that take care of the automatic placement of the nodes of the diagram and/or ensure the links (connections between nodes) have optimal shapes. Now, we may wonder: with automatic arrangement of diagrams, humans (developers or end-users) have no role to play anymore? Well, don't worry, there is still place for humans taking decisions. These decisions concern basically two kind of choices that are in the hands of the developer (alternatively, if appropriate, the developer can offer these choices to the end-user thanks to configuration GUI):

1. The choice of the graph layout algorithm among those available: Hierarchical, Tree, UniformLengthEdges, TopologicalMesh, Bus, Circular, Grid, Short and Long Link layouts (the framework also allows developing your own algorithm or to combine existing layouts via the IlvMultipleLayout and IlvRecursiveLayout classes).
2. The configuration of each graph layout algorithm, thanks to a wide range of options allowing to fine-tune the display according to specific needs or taste.

Choosing the graph layout algorithm

Let's first focus on the first point: how to chose the graph layout algorithm that fits. In some cases, this choice is quite straight-forward, in other cases it is a bit trickier, and to guide your choice you can find below a short synthesis of information present at various places in the IBM ILOG Diagrammer documentation (Reference and User Manual), enriched with a few additional hints.

Hierarchical Layout

This algorithm organizes the nodes in horizontal or vertical levels, in such a way that the majority of the links flow uniformly in the same direction.
A typical example:
This algorithm is the obvious choice for representing hierarchies, that is for hierarchically-structured data (graphs) such as process flows, workflow diagrams, and so on. The key point about the data is whether the links (connections between nodes) are oriented, that is whether the direction of the link matters. If it matters, it is very likely that Hierarchical layout fits (however, in some cases the Tree layout might fit better, see below).

Tree Layout

The obvious choice for representing trees (such as organization charts), where each node has one single ancestor.

Typical examples:

Can also be used if the graph is not a (pure) tree, however in this case the Tree layout only takes care of the shape of part of the links (those that contribute to the pure tree part of the graph, the so-called "spanning tree" of the graph). The Tree Layout is also very fast. If you have huge graphs with millions of nodes, it is often better to find tree structures and visualize only those than to try any other layout algorithm that would be too slow and would not expose any reasonable structure of the huge graph.

Uniform-Length-Edges Layout

This is the force directed layout algorithm (sometimes also called Spring Embedder Technology) of JViews. Your graph does not represent a hierarchy, is not a tree and the orientation of links does not matter? Then Uniform-Length-Edges layout is likely to fit.

An example:
A remark, however. For large graphs (say, with thousands of nodes), this algorithm is much slower than Hierarchical or Tree layout. Therefore, for performance reasons, you may prefer using them for such graphs, even though the graph is not a Hierarchy nor a Tree.

**Topological-Mesh Layout**

The Topological-Mesh Layout produces very similar results to the Uniform-Length-Edges Layout, but it is specialized for meshes, that is, cyclic graphs without cut nodes and cut edges. (The graph-theoretical term is *biconnected graphs*). It does not take link orientations into account and treats the links as straight lines.

An example:

When the Topological-Mesh Layout cannot be applied but you need the same layout characteristics, try the Uniform-Length-Edges Layout.

**Bus Layout**

The Bus Layout is designed for Telecom applications where several nodes are connected to a bus. The bus is a stretchable element, and the algorithm calculates a snake-like shape for the bus.

An example:

**Circular Layout**

The Circular Layout is designed for Telecom applications where nodes are partitioned into clusters. The clusters are displayed as rings or stars and positioned in a tree like fashion relative to each other.

An example:
In most Telecom applications, the nodes represent network devices that have predefined cluster ids. Hence, the layout algorithm allows to specify the clusters. If no cluster ids are available, the layout algorithm is also able to calculate appropriate clusters from the graph topology.

**Grid Layout**

Obvious choice in case your graph has no links at all, or you want nodes to be placed on a grid or matrix while ignoring the links.

An example:

![kb2381_Grid.gif](kb2381_Grid.gif)

**Short and Long Link Layout**

Both are link layouts, that is they do not move the nodes, they only reshape the links in such a way that crossings and overlaps are reduced or avoided.

But why two different algorithms, Short link layout and Long link layout? The answer is that they have different (mostly complementary) characteristics, and, depending on the case, one or the other provides results that fit better the aesthetic and performance requirements. How to chose one or the other? The dumb rule is to try any of them; if it does not fit, try the other one (and explore the multiple options of each).

That said, it is also useful to be aware of some characteristics of each, which help making the choice. In very brief, the names "Short Link layout" and "Long Link layout" refer to the fact that the first one fits better when most links are "short", that is they interconnect relatively close nodes, without too many other nodes placed as obstacles that would need to be avoided by the path of the link. It is the converse for "Long Link layout": it usually fits better when links connect distant nodes with many obstacles in-between.

A more in-depth comparison follows.

**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.
- Crossing 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 end points of a link.
- 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.
- Fast algorithm with low memory footprint.

An example:
Long Link layout

- Links are placed on a grid.
- Link-to-node crossings of orthogonal links are avoided, even if this introduces many bends.
- Overlapping of nodes and links is always avoided, while link crossing cannot always be avoided.
- Slower and uses more memory depending on how small is the grid spacing.

An example:

The two screenshots above hold for the same graph. At a quick look, the results of the two link layouts are pretty much similar, but a closer look, many details are different in terms of link-link crossings, link-node overlaps, symmetry of connection points with respect to the node box, number of link bends (how many turns).

Parameters that matter most for the graph layout algorithms

For most algorithms, the number of available options is very high, such they can be adapted to the variety of user's needs. Parts of the options are intended for expert users only, some are used more commonly, and only a few are very basic options that every user of the algorithm should be aware of. The reference and user manuals provide in-depth guidance about all parameters. Here, we highlight the most important options of each algorithm, those that nobody should miss when starting using the algorithm.

Hierarchical Layout

- `flowDirection`: left | right | top | bottom. This defines the direction towards which the diagram "flows".
- `globalLinkStyle`: orthogonal | polyline | straight. Different styles can be intermixed in the same diagram.
- A large number of dimensional parameters, which define various spacing options.
- And many others

Tree Layout

- `layoutMode`: free | level | balloon | radial | alternating radial | various variants of tip-over modes. The arrangement differs quite significantly depending on this parameter, it is therefore interesting to experiment the modes for choosing the one that fits better.
- `flowDirection`: left | right | top | bottom. Defines the direction towards which the diagram "flows".
- `globalLinkStyle`: orthogonal | straight. Both styles can be intermixed in the same diagram.
• and many others

**Uniform-Length-Edges Layout**

- layoutMode: incremental | non-incremental | fast multilevel. The most useful is the "fast multilevel" mode, because it is much faster for large graphs. The "incremental" mode helps in case it is important that the new arrangement is close to the previous arrangement, thus preserving the so-called user's "mental map".
  - preferredLinksLength: value. Increase the value to spread the nodes over a larger area, or decrease the value to get a more dense arrangement.
  - and many others

**Topological-Mesh Layout**

- nodesPlacementAlgorithm: slowGood | quickBad. Whether the mesh is arranged slowly with high quality, or quickly with lower quality
  - exteriorCycleId: value. The graph consists of several grid meshes and you can choose which one should be placed at the exterior border of the layout.

**Bus Layout**

- bus: node. Determines which node acts as bus. The bus node must be stetchable.
  - nodeComparator: comparator. Determines the order of the nodes along the bus.
  - various spacing and alignment options
  - and many others

**Circular Layout**

- clusteringMode: bySubgraphs | byClusterIds | Automatic. Determines how the clusters are specified: either each subgraph forms a cluster, or the clusters are specified per node by an id, or the clusters are automatically determined from the graph topology without any additional specification.
  - when the clustering mode is byClusterIds, specifying clusters, star centers, ordering index changes the layout most.
  - when the clustering mode is Automatic, parameters such as minimumClusterSize, maximumClusterSize, clusterByBiconnectivity influence how the clusters are found.
  - various spacing options
  - and many others

**Grid Layout**

- layoutMode: rows | columns | matrix with fixed total width | matrix with fixed total height. The nodes can be arranged either on rows, or on columns, or in a matrix (grid) for which either the total width or the total height is specified.
  - various spacing and ordering options
  - and many others

**Short Link Layout**

- globalLinkStyle: orthogonal | direct (different styles can be mixed in the same diagram)
  - various dimensional parameters, the most useful being linkOffset which defines the minimum offset between links connected to the same node
  - and many others

**Long Link Layout**

Starting guide for using graph layout with JViews Diagrammer
• `globalLinkStyle: orthogonal | direct` (different styles can be mixed in the same diagram)

• `horizontalGridOffset` and `verticalGridOffset: value`. This is a key dimensional parameter which defines how dense is the grid where link points are placed. A finer grid (lower grid size) usually improves the result, however at the price of a slower computation and of an increased memory footprint.
• various other dimensional parameters
• and many others

For going further, you can consult the following section of the manual: "Rogue Wave JViews Diagrammer > Programmer’s documentation > Using graph layout algorithms > Layout algorithms".