FastGraph Example Note

Table of Contents

Introduction

JGraph provides a large range of functionality in the core library aimed at the majority of users and use cases. To make JGraph as simple as possible for the bulk of people, there are not excessive switches and configuration for each of the features. This can mean that users seeking high performance can be disappointed by the library as-is, in terms of both speed and memory footprint. These notes and the FastGraph example are mostly aimed at users looking to produce a fairly simple application that loads a huge graph initially and can afford to disable certain features for performance advantages. Though, many of the ideas can be adapted for other uses.

JGraph adds new functionality through extension, but also disables functionality through extension. By providing method hooks for sub-classes it is possible to make a wide range of changes without changing the core library. The FastGraph example demonstrates the disabling of various features of JGraph in order to improve performance and memory footprint. The majority of users do not use the features mentioned and so it is important to check their are switched off. Some of the memory footprint improvements are achieved using more common features being disabled and consideration should be given at a design level as to whether or not the features are required. However, enabling and disabling the features are generally very simple changes, so late changes to these area are not generally an issue.

A lot of changes went into JGraph 5.5.1 relating to performance and memory footprints, if these issues are of concern it is worth upgrading.

Performance Issues

- Use the simplest router for edges you can afford to
- Turn visible ports off, that is the ports are not always visible on all vertices at all times. If you have a large number of vertices painting the ports takes time. See the FastGraph constructor in the example code for the call (`setPortsVisible(false)`).
- Turn the grid off and snapping to grid (`setGridEnabled(false)`). Snapping a large number of vertices takes time.
- Turn selection upon insertion off, selecting a large insertion can cause a large performance hit. There are two methods in the GraphLayoutCache that deal with this, depending on whether or not the cache is partial. If in doubt disable both:

  ```
  graph.getGraphLayoutCache().setSelectsAllInsertedCells(false);
  graph.getGraphLayoutCache().setSelectsLocalInsertedCells(false);
  ```

- Consider turning double buffering off

  ```
  setDoubleBuffered(false);
  ```

Double buffering creates flicker-free graphics at a often very large performance penalty. Consider whether or not your application is less useable because of flickering or the user has to wait 20 seconds for a redraw? Dynamic use of double buffering is also an option when there is a large amount on the screen turn it off and turn it on for simple graph or detailed zooms on large graphs.

- Consider whether or not you need multiple independent views in your graph. Inserting directly into the model insert() method (same applies to the edit() method) is quicker than inserting into the GraphLayoutCache. If there is no view local information (i. e. no information that is only to go into certain views) inserting into the model is perfectly valid. Even if you need independent views, large initial insertions that are common to all views can go into the model, later changes can be via the GraphLayoutCache.

Memory Issues

Memory footprint not only affects whether or not the application can run without an out of memory message, but also reducing the memory footprint considerably improves performance by reducing the number of constructor calls and causing the application to be less likely to use swap memory space on the host machine. If memory issues are encountered consider changing the heap space available to the application, for example, the -Xms and -Xmx VM command line arguments set the initial heap size and maximum heap size respectively.

```
-Xms64m    sets the initial heap to 64MB
-Xmx256m   sets the maximum heap to 256MB
```

- Consider changing initial capacities of commonly used collections, in particular attribute maps. Think about the lowest value of capacity that shouldn't result in a reallocation for the majority of cells. Remember to consider that maps are resized when their capacity reaches the load factor proportion of the capacity.

- Try to avoid begin and end decorations on edges if you have a large number of edges. The drawing points of each one need to be stored.

- A large difference in memory footprint for huge insertions can be gained from disabling undos. If the graph is empty before a huge insert the application could work out how to restore the empty graph rather than store the large amount of information in the undo. FastGraph demonstrates how this can be done.

  FastGraphModel has a variable called `undoDisabled`, simply set to true in this example. You might want to set this to true for a large insert call and then enable undos for the other graph operations, for example. This custom model overrides insert() and handleAttributes(). insert() uses the flag to avoid indicating that an undoable event has occurred and handleAttributes() calls the superclass method if undos are allowed, otherwise does not create the undo like its parent method.

  If undo is disabled for a large initial insert there is another memory saving that can be gained in the cell views. FastGraph provides custom views for ports, vertices and edges. The getCellAttributes() in the parent AbstractCellView class clones the attribute map on each refresh so that the original attributes are not changed in-place. The downside of changing attributes in place is that undos are not created correctly, but if they are disabled, this isn't a problem. The custom view versions of getCellAttributes() in the custom view check whether or not undo is enabled and, if not, they use the model cell's attributes instead of performing the clone for every attribute map of every cell.

- If your application will not have views that use view-local attributes (where view attributes override the models) then the attributes member variable of cell views does not need to be used and does not need to be allocated space. The custom cell view in FastGraph overrides the createAttributeMap() hook to return the static empty attribute map AttributeMap.emptyAttributeMap. By overriding changeAttributes() and setAttributes() to work directly on the allAttributes map all cell view information will be stored there and attributes will be unused and unallocated. Note that if undos are left disabled then allAttributes will always be the actual model cell attribute map.

So one attribute map per cell view can be saved by not allowing view-local attributes for that cell type and another attribute map per cell view can be saved by not allowing undos. The memory footprint saving is considerable for large numbers of cells but this needs to be weighed up against the loss of functionality in an application.

The JGraph development team have a great deal of experience in profiling JGraph-based applications and tuning JGraph for quick and large performance improvements. If you are considering getting for application tuned by the experts contact sales@jgraph.com for a quote.