org.jax.mgi.shr.graphs

Author: Joel Richardson
Created: March 14, 2005
Last Modified: Friday, June 10, 2005

1 Purpose of Document

This document describes the classes belonging to the org.jax.mgi.shr.graphs package and provides code examples for some common usage patterns.

2 Introduction

This package provides classes and interfaces for building and manipulating finite, undirected graphs. This package is a preliminary offering sufficient for implementing the MGI bucketizer, which needs only to build a graph and enumerate its connected components. The package will grow as time permits and need dictates.

This document assumes the reader is familiar with basic graph terminology. For an introduction to the (vast and fascinating) world of graphs, see any good algorithms or data structures text.

The main design points of this graph package are as follows:

- Any object (i.e., of type Object) can be a node. An object can be a node in any number of graphs. There is no special "Node" class. An object need not know that it is a node in a graph.
- Edges are not themselves objects; there is no Edge class or interface. Rather, graph methods that implement edge operations take two nodes as arguments.
- Edges may have "labels", a term we use generically to refer to an object (any Object) that "decorates" the edge. By default, edge labels are null, but can be set as desired.
- Each graph has a default edge label that it assigns when no other is specified; this default is null (by default!) but can be set as desired.
- Graphs are separated from traversals over them. The Graph class provides only basic structuring operations, such as to add a node or query for the neighbors of a node. Any operation that requires processing of all nodes and edges is considered a traversal.

3 Package contents

3.1 Overview.

Interface **Graph** defines the public interface that a graph implementation must provide. The interface defines basic node and edge manipulation functions.

Class **SimpleGraph** is a default implementation of the Graph interface. SimpleGraph uses a 2-level map representation. It has good performance characteristics for all

operations. It has good space characteristics as long as the graph is not too dense. (For dense graphs, an adjacency matrix implementation can be a better choice. This has not been implemented.)

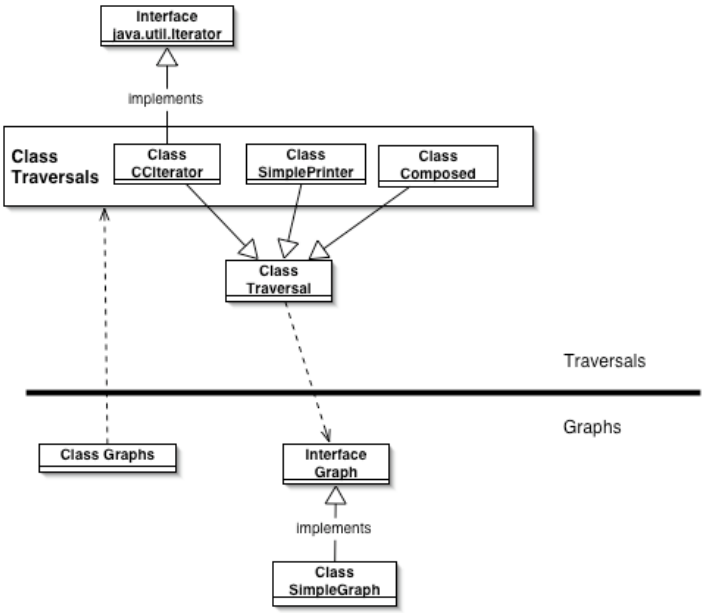
Class **Graphs** is a collection of useful static methods on graphs which are not part of the Graph interface. These methods are often simply wrappers around traversals defined in the Traversals class. The class will grow as more utility methods are added to this package.

Class **Traversal** encapsulates the process of visiting nodes and crossing edges in a graph and separates the traversal from the processing needed to accomplish a specific goal. A traversal can be set to be depth-first or breadth-first. Callbacks are invoked during a traversal. To implement a specific algorithm, one creates a subclass of Traversal, adds whatever state variables are needed, and overrides the desired callbacks.

Class **Traversals** is a container for the specific traversals included in this package. Each traversal is a static class inside Traversals. This class will grow as more useful traversals are implemented.

Class **SimplePrinter** (crudely) prints nodes and edges to a PrintStream.

Class **CCIterator** returns connected components of a graph, Iterator-style.



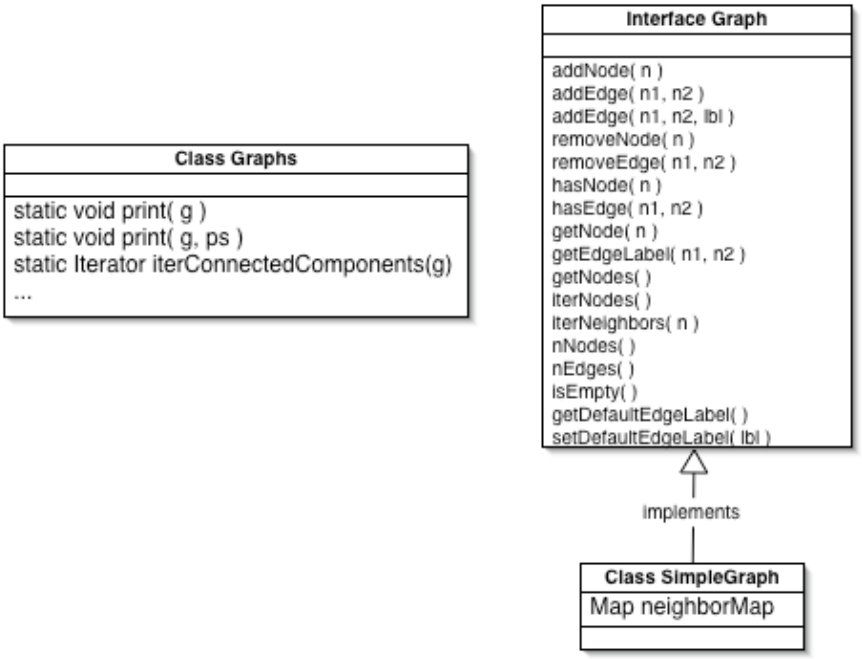
3.2 Graphs.

The following class diagram outlines the Graph interface. The essential building operations are addNode and addEdge. Both are idempotent; adding a node or edge that already exists has no effect. Edges are not manipulated directly (there is no edge object or "get edge" method); users can only test for the existence of edges, and get/set an edge's label. A graph has a default value that is assigned to the edge label when no explicit value

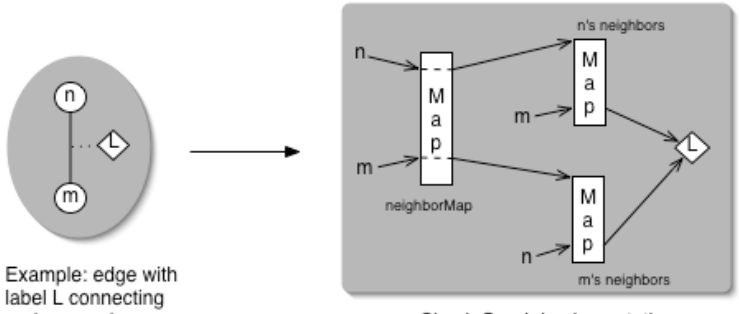
is given; this value is used by the addEdge(n1, n2) call. An explicit value can also be given by using the call addEdge(n1, n2, label).

Graphs do not support multi-edges (multiple edges between a pair of nodes).

Graphs do not allow self-edges (an edge that joins a node to itself). This is because it complicates the notion of neighbor and complicates traversals.



The SimpleGraph implementation represents a graph with a two-level map. The first level maps each node object to its neighbors, which are maintained in another map. The second level maps the neighbors of a given node to the (possibly null) label of the edge between them. If a node has no neighbors, its second level is simply an empty map.



Example: edge with label L connecting nodes n and m.

SimpleGraph implementation

Map objects == 1 + nNodes
Map entries = nNodes + 2*nEdges
Labels between 0 and nEdges

Because the implementation uses HashMaps, the following operations are all constant time: addNode, hasNode, removeNode, addEdge, hasEdge, removeEdge.

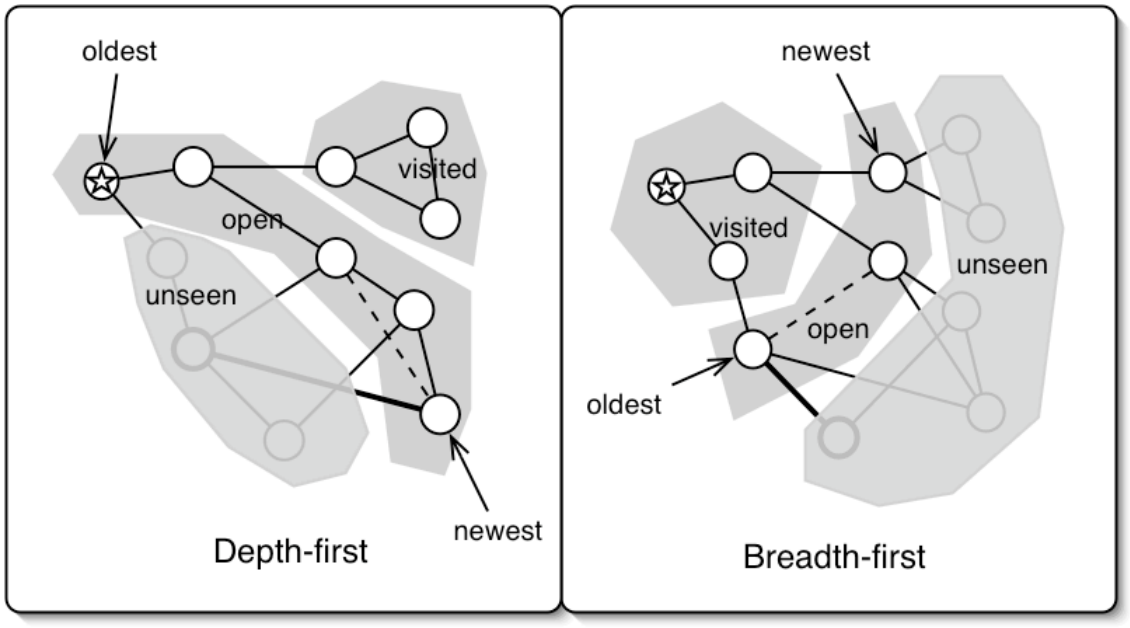
3.3 Traversals.

A traversal is a procedure that visits nodes and crosses edges in a graph. The Traversal class encapsulates this notion, and defines callback functions that are invoked at specific points in a traversal. The callbacks, implemented by subclasses, provide the "hooks" necessary to implement a specific algorithm.

Traversal run-state. The traversal class defines two run-states: `STATE_IDLE` and `STATE_ACTIVE`. At any point, a traversal instance is in one of these two states: it is `STATE_IDLE` from the time it is created until the traversal is initiated, `STATE_ACTIVE` while nodes and edges are being processed, and `STATE_IDLE` again after the traversal is done. Certain operations will throw an exception (`Traversal.StateException`) if invoked in the wrong state. The method `getState()` returns the current traversal state.

Traversal execution. A traversal begins with a graph and a set of starting nodes. If no starting nodes are given, the graph's node set is used. (By default we traverse the entire graph.) The traversal comprises a top-level loop and a "reach" procedure. The top-level loop iterates through the start list; if a node has not yet been visited, then it is passed to the reach procedure, which visits the node and everything reachable from it. Thus, each iteration of the top-level loop processes one connected component.

Traversal strategy. At any point during a traversal, the nodes in the graph can be divided into three groups: the nodes we have seen and are done with (the visited set), the nodes we have seen and are not done with (the open set), and the nodes we haven't seen at all (the unseen set). A traversal repeatedly picks a node from the open set. If that node has an unseen neighbor, the neighbor is added to the open set; if not, the node is moved to the visited set. This process is repeated until the open set is empty. Depth-first and breadth-first differ in which open node is chosen. In depth-first, the node most recently added to the open set is chosen; the open set is managed as a stack. In breadth-first, the node that has been open the longest is chosen; the open set is managed as a queue.



This diagram shows shapshots of a depth-first and a breadth-first traversal over the same graph, starting from the same node. In both diagrams, the starting node is marked with a star, the most recently added open node is marked "newest", the least recently opened is marked "oldest", and the node that will next be added to open is indicated with a thick edge. At any point in a DF traversal, the open set forms a path from the start node to the newest member of the open set; the traversal tries to elongate this path by adding an unseen neighbor of newest, backing up if there are none. A BF traversal spreads out in a wave from the starting node. The open set is the "wave front," and contains nodes the same distance (plus or minus one) from the start.

Tree edges, back edges, and cycles. A graph that contains no cycles is called a tree. In traversing such a graph, nodes in the open set have *uncrossed* edges connecting them to nodes in the unseen and visited sets, but never to another node in the open set[1]. Conversely, if a graph contains cycles, then those cycles reveal themselves as uncrossed edges between open nodes; that is, at some point the traversal will consider an uncrossed edge from an open node, and discover that the other node is already open. Such edges are called "back edges"; all other edges are called "tree edges". The above diagram shows two back edges, drawn as dotted lines. Note that a specific edge can be a back edge in one traversal and a tree edge in another (there are two examples in the diagram). The important point is that *some* edge in every cycle (assuming the graph has cycles) will be identified as a back edge. The method `isBackEdge()` returns true if the traversal is currently crossing a backedge, and false at all other times.

---

[1] Proof by contradiction: assume such an edge exists. I.e., there exist two open nodes, n and m, such that edge (n,m) exists. Because they are open, n and m are both reachable from the start node, s. But now there is a cycle: [s, …, n, m, …, s] (or [s, …, m, n, …, s]), which is a contradiction.

Callbacks. Callbacks are methods (overridden by subclasses) that are invoked during a traversal in order to customize node and edge processing. There are two sets of callbacks invoked at the top-level. The global (or graph-level) (or `go()`-level) methods, `gPre` and `gPost`, are invoked immediately before entering and upon completion of the loop, respectively. As well, the top-level callbacks, `tPre` and `tPost`, are invoked immediately before and after the reach phase, respectively, for each starting node. In pseudocode, the top-level looks like this:

```
def go():
      gPre()
      topLevelLoop()
      gPost()

def topLevelLoop():
      foreach node n in starting list:
          if n not visited:
             tPre(n)
             reach(n)
             tPost(n)
```

During a reach phase, the node and edge callbacks (`nPre, nPost, ePre, and ePost`) are invoked at specific points. The general rule is the `ePre` and `nPre` are called when a node is moved from unseen to open, and `nPost` and `ePost` are called when the node is moved from open to visited. Here's the pseudocode (note: in the following, "tp" stands for "traversal parent"):

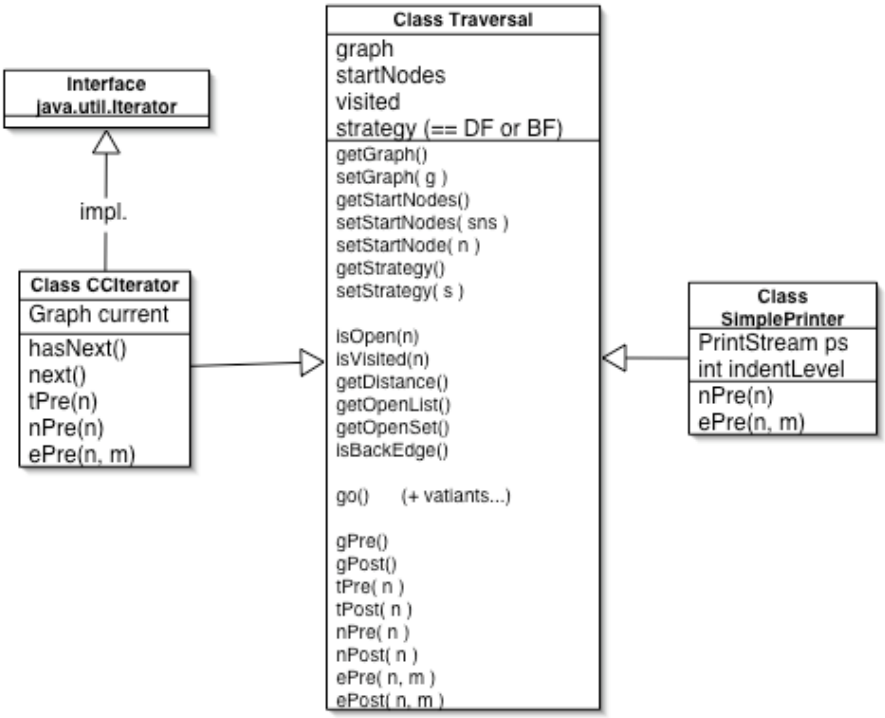
```
def reach( s ):
      Add s to open, with tp=null
      nPre(s)
      While open is not empty:
            Choose a node, n, from open.
            Find an unvisited neighbor, m, of n.
            If no such neighbor exists:
                  // Finished with n
                  nPost(n)
                  Move n to visited
                  if n's traversal parent is not null:
                        ePost(n's tp, n)
            Else If m is currently open:
                  // Back-edge
                  ePre(n, m)
                  ePost(n, m)
            Else:
                  // Add m to open
                  ePre(n,m)
                  Move m to open, with tp=n
                  nPre(m)
```

Aborting a traversal. Normally, a traversal will continue until all nodes reachable from any starting node have been visited. A traversal may wish to terminate early for various reasons. The `abort()` method may be used for this purpose. If a callback calls `abort()`, the traversal is immediately returned to the idle state, and control passes to `gPost()`

which can do any post-processing. The method `wasAborted()` can be used to tell whether the traversal was terminated early or not.

Traversal results. A traversal may produce a result, which is returned by the `go()` methods, as well as by the `getResult()` method. By default, a traversal returns `null`. A traversal's result is set by the `setResult()` method. Traversal results may be get and set at any point before, during, or after a traversal, and its use is completely up to the specific traversal subclass.

Class diagram. The following class diagram outlines the interface of the `Traversal` class, as well as two concrete traversal classes which are included (as static inner classes) in class Traversals.



Utility methods. The Traversal class provides several methods that are used internally and may be used by callbacks. `isOpen` and `isVisited` test whether a given node is currently in the open or visited sets, respectively. `getDistance` returns the depth of the current stack (in depth first) or the current "generation" number (in breadth first). Other methods provide (read-only) access to the current open/visited/unseen sets. (*Not all of these are implemented yet.*)

Note that `Traversal` is not an abstract class. It implements all callbacks as empty functions.

Also note that a Traversal should **not** update the underlying graph (by adding/removing nodes/edges) while it is active. Such updates are not prevented by the current

implementation, but the results are unpredictable. Instead, the traversal should accumulate lists of changes (e.g., edges to delete), then apply them in `gPost`.

4 Configuration

At this time, there are no configuration parameters or environment variables defined or used by this package.

5 Examples

The following code shows an example of building up a graph then printing its connected components. In this example, the nodes are simple Strings, and there are no edge labels.

```
Graph g = new SimpleGraph();
g.addnode("A");
g.addNode("B");
…
g.addEdge("A", "G");
g.addEdge("K", "M");
…
CCIterator iter = new Traversals.CCIterator(g);
while(iter.hasNext()){
      Graph cc = (Graph) iter.next();
      Graphs.print(cc);
}
```

The next example shows a simple traversal class. It presumes a depth-first traversal, and simply prints each node as it is encountered, indented by the node's depth in the traversal. We maintain the current indentation as a StringBuffer; we override the callback `gPre` to ensure it contains "" at the start of traversal. When we encounter a new node, n, the callback `nPre` prints n and lengthens the indentation by a space. The descendants are then printed, and when we exit the node, `nPost` removes the space.

```
public class MyPrinter extends Traversal {
      private StringBuffer indent;
      protected void gPre(){
            this.indent = new StringBuffer("");
      }
      protected void nPre(Object n){
            System.out.print(this.indent);
            System.out.println(n);
            this.indent.append(" ");

      }
      protected void nPost(Object n){
            this.indent.deleteCharAt(0);
      }
}
```

6 To do

There are many possible extensions/followup projects.

- Add priority-first traversal

- More traversals. Minimal spanning tree. Shortest path. Bi-connected components. Better printers/formatters. …
- Integration with GraphViz. E.g., read/write DOT files. Run dot (and friends).
- Add directed graphs (and traversals) to package.
- Add adjacency-matrix implementation, designed to handle dense graphs.
- Add to Graph interface/SimpleGraph implementation:
    - test for subgraph
    - graph union/intersection/difference
    - more convenience methods, e.g., add bunches of nodes/edges
- Other thoughts
    - be able to limit/enforce the type of nodes on a per-graph basis?
    - currently unsynchronized. Define a SynchronizedGraph wrapper?