The Oiler Tutorial

Samad Lotia

October 31, 2007

1 Introduction

Oiler, a pun on the name Leonhard Euler, is a high-performance Java library for operating with mathematical graph objects. While programming with Oiler requires a higher learning curve than other graph libraries, its performance is better and memory consumption is lower.

Oiler's performance gains and memory consumption reduction lies within its leveraging of Java's primitive types, especially the type `int`, over objects to manipulate the graph. Because of this, Oiler trades a lower learning curve for a higher performing graph library.

Oiler provides the ability to generically define the type of objects to be associated with nodes and edges. Any kind of object can be used to store information about nodes and edges. Objects associated with each node and edge can be quickly retrieved and updated as needed.

The purpose of this document is to serve as a tutorial for Oiler's features. It assumes familiarity with Java, object-oriented programming terminology, Java's generic programming features, and Java's built-in data structures in the `java.util` package.

2 Getting Started

Compiling To compile Oiler, issue `ant jar` in the `oiler` directory.

```
[~]
$ cd oiler/

[~/oiler]
$ ls
build.xml  lib  src  tests  tutorial

[~/oiler]
$ ant jar
...
BUILD SUCCESSFUL
Total time: 2 seconds
```

The file `oiler.jar` contains the entire Oiler library. Make sure to include this file in the Java class-path to compile and execute programs using Oiler.

Java Docs To produce the Java Docs for the Oiler library, issue the `ant docs` command in the `oiler` directory. The Java Docs is a reference for all of Oiler's classes, interfaces, and methods.

```
[~/oiler]
```

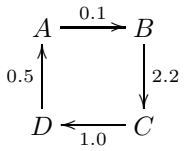
```
$ ant docs
...
BUILD SUCCESSFUL
Total time: 3 seconds
```

The file `docs/index.html` is the starting point for the Java Docs.

3 A first Graph

In the first example, a simple graph is created.

Figure 1: A simple graph



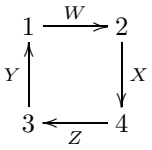
In the simple graph, nodes are `String`s ("A," "B," "C," and "D") and edges are `Double`s (0.1, 0.5, 2.2, and 1.0). The definition of the graph object looks like this:

```
Graph<String,Double> graph
```

In the above snippet, the first generic type parameter of `Graph` is `String`, and it is the object type associated with nodes. The second generic type parameter is `Double`, and it is the object type associated with edges.

One can easily use different node or edge types instead of `String`s or `Double`s.

Figure 2: A graph with different node and edge types



Looking at the graph with different node and edge types, the node type is `Integer`, and the edge type is `String`. The graph's definition looks like this:

```
Graph<Integer,String> anotherGraph
```

`Graph`'s first generic parameter is `Integer`, representing nodes, and the second generic parameter is `String`, representing edges.

Returning back to the simple graph example, the next step is to instantiate the graph with the `LinkedListGraph` implementation:

```
Graph<String,Double> graph = new LinkedListGraph<String,Double>();
```

`graph` is an empty graph without any nodes or edges. Now that there is an instance of the graph, it is possible to add nodes:

```
int nodeA = graph.addNode("A");
int nodeB = graph.addNode("B");
int nodeC = graph.addNode("C");
int nodeD = graph.addNode("D");
```

Notice `addNode()` returns an `int`. As stated in the Introduction, Oiler's performance derives from using primitive data types like `int`. Oiler represents all of its nodes as an `int`. These `int`s are called "node indices," and they are a way to ask the graph about its nodes. Each node in the graph has a unique node index. Because each node index is unique, testing whether two nodes are the same can be done as follows:

```
nodeX == nodeY
```

Edges also have edge indices that are unique to the graph. The following code demonstrates creating the edges for the graph.

```
int edge0 = graph.addEdge(nodeA, nodeB, 0.1, Graph.DIRECTED_EDGE);
int edge1 = graph.addEdge(nodeB, nodeC, 2.2, Graph.DIRECTED_EDGE);
int edge2 = graph.addEdge(nodeC, nodeD, 1.0, Graph.DIRECTED_EDGE);
int edge3 = graph.addEdge(nodeD, nodeA, 0.5, Graph.DIRECTED_EDGE);
```

`addEdge()` has three more parameters than `addNode()`. The parameters of `addEdge()` are:

1st parameter The source node of the edge.

2nd parameter The target node of the edge.

3rd parameter The edge object.

4th parameter The edge type. This will be discussed in depth later.

4 Node and Edge Objects

When nodes and edges were created, its associated objects were passed in. When node A was created, the object `"A"` was passed in to the `addNode()` method. When edge 3 was created, the object `new Double(0.1)` was passed in to the `addEdge()` method. (Java implicitly added the `new Double` part; this is called "type boxing.")

The methods `edgeObject()` and `nodeObject()` retrieve objects associated with edges and nodes. The following example demonstrates its usage:

```
System.out.println("Node A's object: " + graph.nodeObject(nodeA));
System.out.println("Node B's object: " + graph.nodeObject(nodeB));
System.out.println("Node C's object: " + graph.nodeObject(nodeC));
System.out.println("Node D's object: " + graph.nodeObject(nodeD));

System.out.println("Edge 0's object: " + graph.edgeObject(edge0));
System.out.println("Edge 1's object: " + graph.edgeObject(edge1));
System.out.println("Edge 2's object: " + graph.edgeObject(edge2));
System.out.println("Edge 3's object: " + graph.edgeObject(edge3));
```

The output of the program is:

```
Node A's object: A
Node B's object: B
Node C's object: C
Node D's object: D
Edge 0's object: 0.1
Edge 1's object: 2.2
Edge 2's object: 1.0
Edge 3's object: 0.5
```

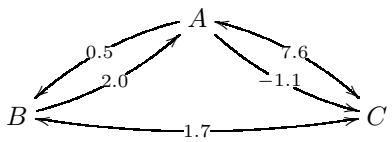
5 Edge Types

Oiler supports graphs with both undirected and directed edges. In the example above where edges were created, the `Graph.DIRECTED_EDGE` parameter was passed in. This told Oiler to create a directed edge. `Graph.UNDIRECTED_EDGE` could have also been passed in to the `addEdge()` method. To demonstrate Oiler's mixed edge capabilities, a mixed graph will be created.

The mixed graph can be represented in Oiler as:

```
Graph<String,Double> graph2 = new LinkedListGraph<String,Double>();
int nodeA = graph2.addNode("A");
int nodeB = graph2.addNode("B");
int nodeC = graph2.addNode("C");
int edge0 = graph2.addEdge(nodeA, nodeB, 0.5, Graph.DIRECTED_EDGE);
int edge1 = graph2.addEdge(nodeB, nodeA, 2.0, Graph.DIRECTED_EDGE);
```

Figure 3: A mixed graph. Lines with a single arrow represent directed edges. Lines with double arrows represent undirected edges.




```
int edge2 = graph2.addEdge(nodeA, nodeC, 7.6, Graph.UNDIRECTED_EDGE);
int edge3 = graph2.addEdge(nodeA, nodeC, -1.1, Graph.DIRECTED_EDGE);
int edge4 = graph2.addEdge(nodeB, nodeC, 1.7, Graph.UNDIRECTED_EDGE);
```

For directed edges, switching the source node with the target node when the edge is created changes its direction. In the following snippet, the direction of the edge is reversed:

```
int edge0 = graph2.addEdge(nodeB, nodeA, 0.5, Graph.DIRECTED_EDGE);
```

Switching the source and target nodes for undirected edges does not effect its directionality. (By definition, undirected edges lack directionality.) Creating `edge2` like this has no effect on the direction of the edge:

```
int edge2 = graph2.addEdge(nodeC, nodeA, 7.6, Graph.UNDIRECTED_EDGE);
```

Determining the edge type is done with the `edgeType()` method.

6 Querying the Graph

6.1 A word about IntIterator

Many classes store a collection of elements. A list represents an ordered collection of elements. A set represents an unordered and unrepeating collection of elements.

Iterators look at a collection without needing to know what kind of class stores the collection. Iterators are important in the Oiler library, as they are the primary means by which one can determine the nodes and edges in a graph, and what nodes and edges are adjacent to a specific node.

The `IntIterator` interface, provided by the `oiler.util` package, is important for querying the graph. It is analogous to the `Iterator` interface of the `java.util` package. The `IntIterator` interface has the following methods:

next() Returns the next `int` in the collection.

hasNext() Returns true if there are more `int`s left in the collection.

numRemaining() Returns the number of `int`s left in the collection.

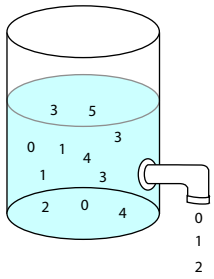


Figure 4: A collection of elements. A tank stores water, and a faucet "retrieves" the contents of the tank. One can use a faucet without knowing where the water comes from or how. An iterator is like a faucet, as it retieves the contents of a collection of elements. One can use an iterator without knowing where the elements come from or how.

6.2 Obtaining all the nodes and edges in the graph

Obtaining all the nodes and edges in the graph can be done with the `nodes()` and `edges()` methods. These methods both return `IntIterator`s. The following example demonstrates its usage:

```
IntIterator nodes = graph2.nodes();
while (nodes.hasNext())
{
  int node = nodes.next();
  System.out.println("Got node " + graph2.nodeObject(node));
}

IntIterator edges = graph2.edges();
while (edges.hasNext())
{
  int edge = edges.next();
  System.out.println("Got edge " + graph2.edgeObject(edge));
}
```

The output of the program looks something like this:

```
Got node A
Got node B
Got node C
Got edge 0.5
Got edge 2.0
Got edge 1.7
Got edge 7.6
Got edge -1.1
```

The order of the nodes and edges `IntIterator` returns is completely arbitrary.

6.3 Finding adjacent nodes and edges

Determining adjacent nodes and edges is done through the `adjacentNodes()` and `adjacentEdges()` methods. The following example demonstrates `adjacentEdges()`'s usage:

```
IntIterator adjEdges = graph2.adjacentEdges(nodeA, Graph.INCOMING_EDGE);
while (adjEdges.hasNext())
{
  int edge = adjEdges.next();
  System.out.println("Incoming edge of nodeA: " + graph2.edgeObject(edge));
}
```

In the above snippet, `adjEdges` will only return the edge with weight 2.0. This edge is the only one that is directed and is incoming to A.

Here is another example of `adjacentEdges()`:

```
IntIterator adjEdges = graph2.adjacentEdges(nodeA, Graph.OUTGOING_EDGE);
```

`adjEdges` returns edges with weight 0.5 and -1.1. These edges are the only edges that are directed and outgoing. The second parameter specifies the type of edge to inspect.

The following example's `adjEdges` returns all directed edges adjacent to A:

```
IntIterator adjEdges = graph2.adjacentEdges(nodeA, Graph.DIRECTED_EDGE);
```

The following example's `adjEdges` returns only undirected edges adjacent to A:

```
IntIterator adjEdges = graph2.adjacentEdges(nodeA, Graph.UNDIRECTED_EDGE);
```

The next example's `adjEdges` returns all edges, directed or undirected, adjacent to A:

```
IntIterator adjEdges = graph2.adjacentEdges(nodeA, Graph.ANY_EDGE);
```

The second parameter can be bitwise or'ed together. The following example demonstrates inspecting only incoming and undirected edges:

```
IntIterator adjEdges = graph2.adjacentEdges(nodeA,
                       Graph.INCOMING_EDGE | Graph.UNDIRECTED_EDGE);
```

The second parameter is optional. If the second parameter is omitted, Oiler calls the `defaultEdgeType()` method and fills its value for the second parameter.

`adjacentNodes()` behaves the same way `adjacentEdges()` does, except it returns indices of nodes at the other end of the edge. It can return the same node multiple times if there are multiple edges connecting the same nodes.

7 Using Oiler's Data Structures

Java supplies many data structures in the `java.util` package. A major obstacle with these data structures is they cannot store primitive data types like `int`. To accomodate this obstacle, Oiler provides several classes for storing `int`s.

7.1 IntArray

`IntArray` is a class that has a resizeable array of `int`s. This class behaves very similarly to `java.util.ArrayList`. The principle methods are `add()`, `get()`, and `set()`. The following example demonstrates the flexibility of `IntArray`:

```
IntArray array = new IntArray();
// The author's birthday
array.add(7);
array.add(26);
array.add(1985);
System.out.println("The array has " + array.size() + " elements");
for (int i = 0; i < array.size(); i++)
  System.out.println("The int at position " + i + " is " + array.get(i));

// The Fibonacci sequence
array.set(0, 0);
array.set(1, 1);
array.set(2, 1);
array.set(3, 2);
array.set(4, 3);
array.set(5, 5);
array.set(6, 8);
array.set(7, 13);
System.out.println("The array has " + array.size() + " elements");
for (int i = 0; i < array.size(); i++)
  System.out.println("The int at position " + i + " is " + array.get(i));
```

The output of the above snippet is:

```
The array has 3 elements
The int at position 0 is 7
The int at position 1 is 26
The int at position 2 is 1985
The array has 8 elements
The int at position 0 is 0
The int at position 1 is 1
The int at position 2 is 1
The int at position 3 is 2
The int at position 4 is 3
The int at position 5 is 5
```

```
The int at position 6 is 8
The int at position 7 is 13
```

7.2 IntHashSet

This class stores a set of `int`s. It is behaves similarly to `java.util.HashSet`. Its principle methods are `add()`, which adds `int`s to the set, and `iterator()`, which returns an iterator for all the `int` in the set. The following example demonstrates its usage:

```
IntSet set = new IntHashSet();
set.add(0);
set.add(1); // Adding 1 for the first time
set.add(1); // Adding 1 for the second time
set.add(2);
set.add(3);

System.out.println("The set has " + set.size() + " elements");
System.out.print("The set contains: "
IntIterator iterator = set.iterator();
while (iterator.hasNext())
  System.out.print(iterator.next() + " ");
System.out.println();
```

The output of the above example is:

```
The set has 4 elements
The set contains: 0 1 2 3
```

Notice 1 was printed only once. This is because every `int` in the set is unique. Adding 1 the second time had no effect on the set.

7.3 IntIntHashMap

This class stores a map, where the keys and values of the map are `int`s. It behaves similarly to `java.util.HashMap`. Its principle methods are `put()`, which inserts a key-value pair into the map, and `get()`, which returns the value associated with a key. The following example demonstrates using this class:

```
IntIntMap map = new IntIntHashMap();
// The Fibonacci sequence
map.put(8, 21);
map.put(5, 5);
map.put(9, 33);
map.put(7, 13);
map.put(6, 8);
map.put(10, 54);
```

```
System.out.println("The 5th number in the Fibonacci sequence: " + map.get(5));
System.out.println("The 6th number in the Fibonacci sequence: " + map.get(6));
System.out.println("The 7th number in the Fibonacci sequence: " + map.get(7));
System.out.println("The 8th number in the Fibonacci sequence: " + map.get(8));
System.out.println("The 9th number in the Fibonacci sequence: " + map.get(9));
System.out.println("The 10th number in the Fibonacci sequence: " + map.get(10));
```

The output of the above example is:

```
The 5th number in the Fibonacci sequence: 5
The 6th number in the Fibonacci sequence: 8
The 7th number in the Fibonacci sequence: 13
The 8th number in the Fibonacci sequence: 21
The 9th number in the Fibonacci sequence: 33
The 10th number in the Fibonacci sequence: 54
```