Moorea: a Semiring-based Graph and Maths Library for Java

Nicolas Stefanovitch

March 2015

1 Another graph library, how can I benefit from it ?

Let us go straight to the matter before presenting the details of the library and start by answering this question, very likely the only one that matters to a new reader.

Mooreais based on two fundamental goals : reusability and ease of use. These concerns have been addressed in two parts :

- in the **design** of the library to provide easy to use and extend class. Because ease of use means providing the users with the right tool for the problem at hand, Moorea proposes several fundamentally different graph representation classes.

- in the use of **semirings** to provide generic algorithms applicable to a wide range of settings. Algorithms used to compute shortest paths, reachability, max capacity links all use numerical valuation of the edges of the graph and two different combination of these value : in fact they all share the same algebraic structure. Therefore, instead of coding three different algorithm to answer these three different problems, it is possible to use only one algorithm with three different semirings.

If you are not familiar with semirings please refer to [], and to look directly at an example of their use, check Section 4.1.

2 A graph and maths library, what do they actually contain ?

Moorea contains three subpackages : graphs, maths and misc.

In *graphs* are located the following subpackages :
— *algorithms* : basic graph algorithms
— *construction* : a wide range of graphs is supported
— *io* : to read and write graph on disk
— *measures* : contains a few graph measures

— *traversal* : contains traversal primitives based on node iterators

In *maths* are located the following subpackages :
— *algebra* : to represent algebraical objects, notably `Monoid` and `Semiring`
— *objects* : to represent mathematical object such as `Variable`, `Assignment` and `Function`
— *matrix* : to deal with matrix objects
— *hypermatrix* : to represent hypermatrix objects
— *lambda* : minimalist implementation of functional programming primitives
— *random* : helper functions for random number generation

In misc are located a number of helper functions, notably for logging, benchmarking, tuples, arrays and propose its own version of a bidirectional map, different from the one of Apache commons.

3 The graph library in more details

3.1 Graph implementation

The definition of a graph is a set of nodes and a set of edges between these nodes. From an implementation point of view, many way can be achieved to do so. Moreover one typically wants to include more information on the nodes and edges of the graph, going beyond this basic definition. The are also different ways of implementing these additional informations.

Because there are no graph representation that is in absolute better than another one in term of algorithmical efficiency, Moorea proposes three basic ways of representing graph, leaving the choice to the programmer to pick the one the most suitable : edges sets in `EdgeGraph`, matrix adjacency in `MatrixGraph`, and adjacency lists in `Graph`.

The library has been designed having also in view the future distribution of the graph structures, and as such focuses on adjacency list graph and aims to store as few information as possible inside the graph object : All information is contained inside the nodes. The class `Graph` is an abstract class that can is reused to create generic non oriented graphs `SimpleGraph` and generic oriented graphs `DirectedGraph`. Figure **??** describes the hierarchy of implemented classes generic class.

Information can be stored either in dedicated fields or in a map. The former approach requires creating a new class each time one wants to add more information to a node, this is the approach used in `ContainerGraph` and `WeightedGraph`. The latter approach is generic and does not require creating a new class in order to add a new information : only a new key has to be used. This is the approach used in `LabelledGraph` and `UniversalGraph`.

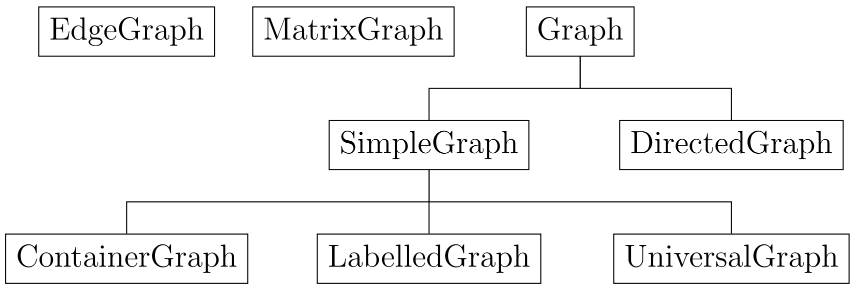

Fig. 1: Hierarchy of generic graph implementations in Moorea

3.1.1 Generic graph types

`Graph` : is an abstract class, it is used to represent operations common to non directed graphs. Because adjacency information is stored at the level of the nodes, it can be used as a basis for building directed graphs, representing access to their non oriented counterparts.

`SimpleGraph` : it is the simplest instantiation of a non directed graph using adjacency lists, themselves implemented with ArrayList. It provides no information apart from adjacency and node id. Nodes can be accessed either trough id or index.

`ContainerGraph` : a graph whose node contain an unique object of the same given type. The object contained in the graph can not access by default to the reference of the node that contain them, and therefore to the adjacency information. Nodes can accessed either through id, index or reference of the object they contain.

`LabelledGraph` : a graph whose nodes contains objects of any type (not necessarily the same type as in a ContainerGraph). Both nodes and edges can be assigned label of any type and marks. Nodes can accessed either through id, index or reference of the object they contain.

`UniversalGraph` : a graph whose properties can be parametrised at construction, specifically, whether it is : directed, weighted, labelled.

`EdgeGraph` : is based on a representation of edges using a set of Edge objects.

`MatrixGraph` : is based on a matrix adjacency representation. The matrix is provided by the `maths` package. The type of the element of the matrix is parametrized. As a consequence it is possible to straightforwardly represent directed graphs and weighted graph using this representation.

3.1.2 Ad-hoc graph types

`WeightedGraph` : a weighted graph with an unique weight of a given type on edges.

```
class CityNode extends
    SimpleNode {
    String name;
    int population;
}

CityNode n;

n.name =
    "Papeete";
n.population =
    26000;
```

(a) SimpleNode

```
class CityData {
   String name;
   int population;
}


ContainerNode<CityData> n;

n.getContent().
    name = "Papeete";
n.getContent().
    population = 26000;
```

(b) ContainerNode

```
String keyName =
    "name";
String keyPopulation =
    "population";


LabelledNode n;

n.setLabel(keyName,
    "Papeete");
n.setLabel(keyPopulation
    26000);
```

(c) LabelledNode

Fig. 2: Data structure declaration, node declaration and information access for different node implementation : (a) ad-hoc node extending SimpleNode, (b) ad-hoc data structure using ContainerNode, (c) ad-hoc node label using LabelledNode

`WeightedEdgeGraph` : same as WeightedGraph, using a Edge representation instead.

`SpatialGraph` : a graph whose nodes contains n-dimensional spatial coordinates.

3.1.3 Choosing a graph implementation

One has first to decide whether he prefers efficiency of execution or easiness to develop. The classes `LabelledGraph` and `UniversalGraph` are very close to what the NetworkX library propose for python. It is very easy to attach and manipulate information on a node or an edge, however, the memory and execution footprint is heavy.

The fastest implementation is to create a derived class of `SimpleNode` (or `DirectedNode` for directed graphs). This way the fields of the node directly contains the relevant information, possibly using simple types. A compromise can be made using the `ContainerGraph`. Figure 2 presents 3 different coding styles using 3 different node implementations.

3.1.4 Basic node and edge operations

The single most important method is `getNeighbours()` in `Node` to get the undirected adjacency list of a node. Figure 3 summarises the different nodes and edges operations available in the generic graph implementations. Node can basically be accessed through their reference, their id, their index in the list of nodes of the graph. More advanced graph structure can index nodes by their content. Hashmaps stored at the level of the graph make these conversion efficient. Edge can be accessed the same way. Note that `ContainerGraph` and `LabelledGraph` propose additional operations on nodes and edges not appearing in the Figure, only basic operations are covered.

Node operations methods, where *operation* ∈ get, has, remove	
`SimpleGraph`	*operation*Node(Node node)
`SimpleGraph`	*operation*NodeById(int id)
`SimpleGraph`	*operation*NodeByIndex(int idx) (not in EdgeGraph)
`ContainerGraph`	*operation*NodeByContent(K content)
`LabelledGraph`	*operation*NodeByObject(Object object)
Edge operations methods, where *operation* ∈ create, has, remove	
`SimpleGraph`	*operation*Edge(Node node1, Node node2)
`SimpleGraph`	*operation*EdgeById(int id1, int id2)
`SimpleGraph`	*operation*EdgeByIndex(int idx1, idx2)
`ContainerGraph`	*operation*EdgeByContent(K content1, K content2)
`LabelledGraph`	*operation*EdgeByObject(Object object1, Object object2)

Fig. 3: Node and edge operation for the different generic graph implementations

3.2 Graph generation algorithms

Generation is done through the instantiation of a `GraphFactory` object that produces graphs of any type with a specific structure. Graph generation can be done either directly or using a mini language that specifies a string encoding of the graph structure, calling the `generateGraph(String graphCode)` of GraphFactory. Supported structure are given in Table 4.

Spatial graphs are generated using a different class : `SpatialGraphFactory` that does not support string encoding. It currently contains only own generation method that randomizes the positions of the nodes and recursively assemble them in connected component by increasing an accessibility radius parameter.

3.3 Graph algorithms

Because currently only a handful of algorithms are implemented, they are regrouped in coherent classes. Refer to Table 5 for the list of available algorithms. Any algorithm contributions is warmly welcomed :).

`GraphExtraction` regroups all algorithms that aims at extracting a part of a graph, for instance, connected components and spanning trees.

`ShortestPath` regroups all algorithms dedicated to shortest path computation, currently only Dijkstra and Floyd-Warshall.

basic graphs	
empty	`empty`
grid (rectangle)	`grid_<width>_<height>`
grid (square)	`gridsq_<side>`
chain	`grid_<length>_1`
ring	`cycle_<length>`
	`cycle_<length>_<thickness>`
star	`star_<size>`
tree	`tree_<depth>_<nb childs>`
tree (directed)	`treedir_<depth>_<nb childs>_<root to child>`
complete	`kn_<nb nodes>`
advanced graphs	
complete bipartite	`kmn_<nb nodes1>_<nb nodes2>`
hypercube	`hyperc_<nb dimensions>`
hierarchical	`hierarchical_<nb node base>_<nb levels>`
regular ring latice	`regringlat_<nb nodes>_<nb neighb>`
random graphs	
Watts-Strogatz	`ws_<nb node>_<K>_<beta>`
Erdos-Renyi	`er_<nb nodes>_p<link probability>`
	`er_<nb nodes>_d<average degree>`
scale-free	`sf_<nb nodes>`

Fig. 4: Supported graph structure and corresponding string encoding

`GraphConvertion` regroups all algorithms dedicated to converting graph from one representation to another.

`GraphTraversal` does simplistic graph traversal, advanced traversal is performed using iterator to be found in `graphs.traversal`.

`GraphMeasure` contains all graph measure algotihms, currently only height and diameter are implemented.

`GraphDisplay` contains one method per generic graph type to display graphs of this kind in the console.

`GraphIO` contains methods to read and write graph to the disk in various formats.

4 The maths library in more details

The maths library is here to collected mathematical objects to represent semirings, to support the graphical library with the representation of matrices, and ultimately to collect representation of any mathematical objet necessary to do optimisation (variable, scope, discrete function, hypermatrix).

4.1 Semiring

To put it in a nutshell with a simple example, the added value of semiring is to replace this bit of code :

```
val = Math.max(x,y+z)
```

by the following one :

```
val = sr.dotOperator.apply(x,sr.sumOperator.apply(y,z))
```

Which is of tremendous interest when such code appear in the middle of a large function : It allows to parametrize the action of this function, making it extremely reusable. Thanks to this generality, instead of writing anew the function for different operations, it is necessary to only provide it with a different semiring.

But how useful is it ? And why would I be interested in using a semiring instead of lambda expression to parametrize my code ?

Everywhere in mathematics where dynamics programming is used or where mathematical distribution applies [1] the algebraical structure of the computations at hand is that of a semiring. Lastly, semirings are implemented using lambda expressions, which is the reason why a minimalist support for functional programming is implemented in Moorea.

Tell me more about it !

A semiring is the definition of a set of element, not necessarily numerical, a combination operation on the set called "sum operator" with a neutral element, a marginalisation operator on the set called "dot operator" with a neutral element and which distributes over the sum operator.

If you are doing optimisation or inference semiring should be of interest for you. Some instances of semirings are the following : $(\mathcal{R}^+, +, *, 0, 1)$ allows to compute the probability of an event, $(\mathcal{R}^+, \max, *, 0, 1)$ computes the event of maximal probability, $(\mathcal{R}, \max, +, 1, 0)$ is the constraint optimisation semiring, $(\mathcal{R}, \min, +, -\infty, 0)$ is the shortest path semiring, $(\mathcal{B}, \vee, \wedge, true, false)$ is the accessibility semiring and the constraint satisfaction semiring. Code developed using a semiring representation automatically applies to all these problem. Please refer to [] for more semirings applications.

The library provide a number of basic operator that can be combined in order to build one's semiring of interest. For instance here we define the constraint optimization semiring :

1. under some additional conditions, which are the one of a semiring

```
Semiring<Integer> sr = new Semiring<Integer>(
    // set
    Integer.class,
    // combination operator and neutral element
    new SumOperator<Integer>(Integer.class),
    (Integer) 0,
    // marginalisation operator and neutral element
    new MaxOperator<Integer>(Integer.class),
    (Integer) 1
);
```

4.2 Matrix

Matrices are represented in the `Matrix` class, which contains two dimensional array of a parametrized type. Algorithms for matrices are located in the class `MatrixAlgorithms` which currently propose only three functionalities : displaying a matrix, getting the optimal element of a matrix and saving to a file.

Matrices are currently used only as a mean to represent adjacency matrix of a graph and to perform shortest path computations using matrix exponentiation in `ShortestPaths`. As a consequence, fundamental operation such as matrix additions and multiplication are not implemented. A `MatrixValueIterator` is used to iterate over the values. The `MatrixFactory` class allows to create matrices and fill them using an iterator, which is used to generate random matrices.

4.3 Hypermatrix

Hypermatrices are n-dimensional matrices. A convenient implementation of hypermatrices is done using hashtable, where a string key represent the index of element. An `Hypermatrix` is a `Function` defined over a scope of `DiscreteVariable` (see next subsection for more details). The class `DiscreteFunctionIterator` allows to iterate over all the values of an hypermatrix.

Hypermatrix algorithms are located in `HyperMatrixAlgorithms`. It contain code to display an hypermatrix, to merge a set of hypermatrices into a single hypermatrices, and to summarise an hypermatrix over a subset of its variables. These operations are necessary notably for probabilistic inference and constraint optimization.

4.4 Mathematical objects

A `Variable` is identified with an unique id. A `DiscreteVariable` is a `Variable` with a cardinality. A `Scope` is a list of `Variable`, they are generated using the class `ScopeFactory`. An `Assignment` is map of `Variable`. A `Function` is an abstract class defined by its domain and codomain and propose an abstract method to get the value of the function given an assignment. These mathematical objects are the fundamental bricks to build a clean hypermatrix implementation.

5 Algorithm implementation examples

We illustrate the implementation of algorithms and use of semirings with the Dijkstra algorithm. We present two different implementations with `WeightedGraph` and `LabelledGraph`. Their usage is quite different, as LabelledGraph allows to store data relative to the graph nodes and edges directly inside the graph, while WeightedGraph doesn't allow this. As a consequence, the algorithm using WeightedGraph has to manage a set of map, while these are directly managed by LabelledGraph. Thanks to the semiring these algorithm can be used to compute : shortest path, longest paths, accessibility and max capacity link.

5.1 Dijkstra using WeightedGraph

```
public static <N extends Comparable> void dijkstraSemiring(
    WeightedGraph<N> g, Node s, Node t, Semiring<N> semiring) {

    List<WeightedNode<N>> nodes = g.getNodes();

    Operator<N> sumOperator = semiring.getSumOperator();
    Operator<N> dotOperator = semiring.getDotOperator();

    Map<Node,N> eval = new HashMap<Node, N>(nodes.size());
    Set<Node> mark = new HashSet<Node>(nodes.size());
    Map<Node,Node> pred = new HashMap<Node, Node>(nodes.size());

    Comparator comp = new Comparator<LabelledNode>() {
        public int compare(LabelledNode n, LabelledNode m) {
            return eval.get(n).compareTo(eval.get(m));
        };
    };

    PriorityQueue<Node> heap = new PriorityQueue(nodes.size(),comp);

    for(Node n : nodes) {
        eval.put(n,semiring.getDotNeutralElement());
    }
    eval.put(s,semiring.getSumNeutralElement());
    pred.put(s,null);
    mark.add(s);

    heap.addAll(nodes);
    while(heap.size()>0) {
        WeightedNode n = (WeightedNode) heap.poll();
        for(WeightedNode a : (List<WeightedNode>) n.getNeighbours()) {
            boolean updateHeap = false;
            if(!mark.contains(a)) {
                mark.add(a);
                eval.put(a, sumOperator.apply( eval.get(n), g.getEdgeWeight(n,a)));
                pred.put(a,n);
                updateHeap = true;
            } else {
                N vn = eval.get(n);
                N va = eval.get(a);
                N c = g.getEdgeWeight(n,a);
                if(dotOperator.apply(sumOperator.apply(vn, c),va) != va) {
                    eval.put(a, sumOperator.apply( vn, c));
                    pred.put(a,n);
                    updateHeap = true;
                }
            }
            if(updateHeap) {
                heap.remove(a);
                heap.add(a);
                updateHeap = false;
            }
        }
    }
}
```

5.2 Dijkstra using LabelledGraph

```
public static void dijkstraLabelled(
    LabelledGraph g, LabelledNode s, LabelledNode t, final Object weightKey) {

    List<LabelledNode> nodes = g.getNodes();

    List<Node> p = new LinkedList<Node>();

    final Object markKey = "marked";
    final Object evalKey = "evaluation";
    final Object predKey = "predecessor";

    g.setAllNodeLabels(evalKey, Integer.MAX_VALUE);
    g.setAllNodeLabels(predKey, null);
    g.removeAllNodeMarks(markKey);
    Comparator comp = new Comparator<LabelledNode>() {
        public int compare(LabelledNode n, LabelledNode m) {
            return (int) n.getNodeLabel(evalKey) - (int) m.getNodeLabel(evalKey);
        };
    };

    PriorityQueue<Node> heap = new PriorityQueue(nodes.size(), comp);

    g.setNodeLabel(s, evalKey, 0);
    g.setNodeMark(s, markKey);

    heap.addAll(nodes);

    while(heap.size()>0) {
        LabelledNode n = (LabelledNode) heap.poll();
        for(LabelledNode a : (List<LabelledNode>) n.getNeighbours()) {
            boolean updateHeap = false;
            if(!a.hasNodeMark(markKey)) {
                a.setNodeMark(markKey);
                g.setNodeLabel(a, evalKey,
                    (int) n.getNodeLabel(evalKey) +
                    (int) g.getEdgeLabel(n, a, weightKey));
                g.setNodeLabel(a, predKey, n);
                updateHeap = true;
            } else {
                int vn = n.getNodeLabel(evalKey);
                int va = a.getNodeLabel(evalKey);
                int c = n.getEdgeLabel(a, weightKey);
                if(vn+c<va) {
                    a.setNodeLabel(evalKey, vn + va);
                    a.setNodeLabel(predKey, n);
                    updateHeap = true;
                }
            }
            if(updateHeap) {
                heap.remove(a);
                heap.add(a);
                updateHeap = false;
            }
        }
    }
}
```

6 Graph algorithms list

GraphExtraction	
List<Node>	getConnectedComponent(Graph, Node)
List< ? extends Graph>	getConnectedComponents(Graph)
Graph<N>	getLartgestConnectedComponent(Graph<N>)
<G extends Graph<N>>	getLargestConnectedComponentFromList(List<G>)
Graph	getSpanningTree(Graph, GraphNodeIterator)
Tupple2<Graph,BidiMap>	minimalSpanningTreeKruskall(WeigthtedGraph<N>)
ShortestPaths	
Tupple2<List<Node>, N>	dijkstra(WeightedGraph<N>, Node, Node, Semiring<N>)
N[][]	getAllSPFloydWarshall(Graph<Node>, Semiring<N>)
N[][]	getAllSPFloydWarshall(N[][], Semiring<N>)
GraphConvertion	
EdgeGraph	graphToEdgeGraph(Graph<K>)
WeightedGraph<Integer>	WeightedGraphFromGraph(Graph< ? extends Node>)
WeightedGraph<Integer>	WeightedGraphFromGraph(Graph< ? extends Node>, Integer)
WeightedEdgeGraph	weigthedGraphToWeightedEdgeGraph(WeightedGraph)
N[][]	adjacencyMatrixFromGraph(Graph<Node>,Class<N>,N,N,N)
moorea.graphs.traversal	
	GraphNodeIterator
	GraphNodeIteratorBFS
	GraphNodeIteratorBoundedBFS
	GraphNodeIteratorDFS
	GraphNodeIteratorRandom
	GraphNodeIteratorAdHoc
GraphDisplay	
void	disp(Graph)
void	disp(ContainerGraph)
void	disp(LabelledGraph)
GraphIO	
void	toAdjacencyList(Graph, String)
void	toDot(Graph, String)
void	toDot(Graph, String, Map<Node, String>)
void	toDotDirected(Graph, String)
void	toGephi(Graph, String)

Fig. 5: List of implemented graph algorithms