Marae: a Semiring-based Graphical Model Library for Java

Nicolas Stefanovitch

March 2015

1 Introduction

Graphical models are a very generic mathematical formulation that can describes algorithms in many field of mathematics, physics and operation research. The aim of this library is to propose an implementation of the algorithm than can be as reusable and generic as their mathematical formulation. This genericity is achieved is through the used of a Semiring object used as a parameter of the generic algorithms. Problems such as probabilistic inference, constraint optimisation, constraint satisfaction, parameter estimation can be dealt directly using the same code instead of implementing anew variation of the same basic algorithm for each applications. This library is based on the generic graph and maths library Moorea.

One has to keep in mind that graphical model are a graph based representation of computation and precedence constraints between computations. Using such an abstract approach is beneficial for two reasons : it is generic and it makes some algorithms clearer. However it is not always the case, for instance while the Vitterbi algorithm for decoding HMM can be represented as a graphical model, it is less efficient and less clear to understand than a straightforward implementation. Accordingly, the currently implemented algorithm for Markov chains and HMM do not use graphical models primitives.

2 Structure

This framework makes it easy to represent combine and marginalise discrete functions. These two operations are the basic building blocks of inference.

Additionally the framework provide means of building different kind of graphical models given a set of functions. Currently implemented are the junction tree and factor graph representation.

2.1 Example graphs

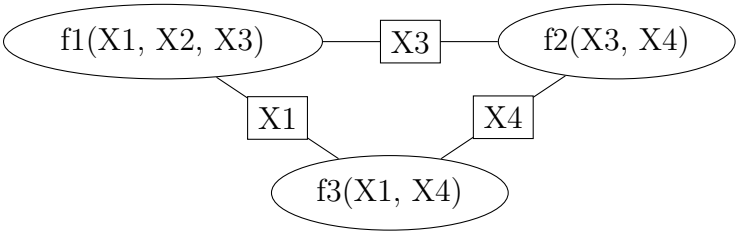
We illustrate this document using a set of 3 functions defined over a set of 4 discrete variables : f1(X1,X2,X3), f2(X3,X4), f3(X1,X4).

Legend : nodes of graphical models are noted either as ellipses or square. Mathematical they have the same type. The visual distinction indicates a difference in term of purpose : ellipse nodes are intended to store information, while square node are represent message exchange between ellipse nodes.

The text of a node, e.g. "X2, X3" represent the scope over which the node is defined. If there is bijection between a node and a function, this is indicated in the text of the clique with the name and scope of the function e.g. "f2(X3,X4)" . Functions allocated to a node are indicated bellow a node.

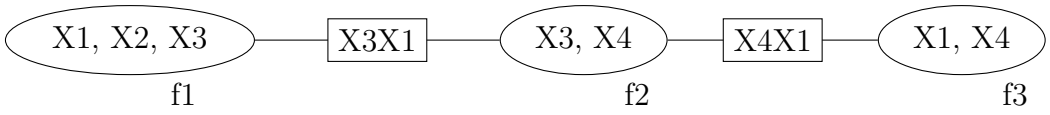
2.1.1 Clique graph

A clique graph has nodes which are groups of variables, which are connected to other clique nodes sharing the same set of variables. Simplest form is to have one clique per function.



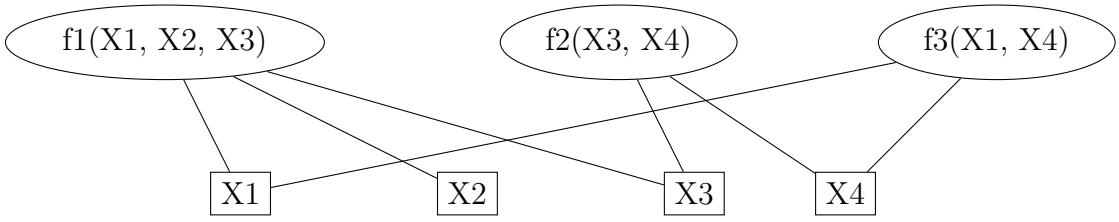
2.1.2 Junction tree

A junction tree is tree shape clique graph. It is constructed with dedicated algorithm transforming the Markov graph of the variables into this secondary graphical structure.



2.1.3 Factor graph

Factor graphs make a distinction between function nodes and variable nodes, isolating explicitly the latter has separate nodes. They are equivalent to a clique graph with one function per node and one node per variable, connecting the function node only to variable nodes, rather than to each others.



3 Inference

What makes the difference between different graphical models algorithms are the following : definition of the scope of the nodes and graphical structure joining them (given by the construction procedure), possible approximations (not covered here) and inference procedure. Inference consist in the precise scheduling of a set of two basic operations : combination and marginalisation. We first describe these, then the scheduling and discuss implementation choices in the next section.

3.1 Operations

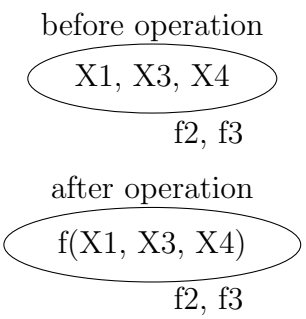
Notations : We illustrate this section with the following convention : scope of computed function is in **bold**, *actual computation to determine the value of the function are indicated in red. The examples are given with hypermatrix functions. The domain of a variable X is noted $D(X)$ and is assumed to be discrete. The computed function are given using a tuple representation, where the last element is the value of the function and the other n-1 elements are the values of the variables of this function.*

These operations applies to any semiring, however for clarity of presentation we present them using $+$ as the dot operator of the semiring and max *as the product operator. A generic description would require to write these respectively as $\oplus$ and $\otimes$.*

Combination This operation consist in merging assignment-wise a set of function using the dot operator of the semiring. This operation creates a new function whose scope is the union of the scope of the function to merge. Combination is the operation used to compute the set of information at a node.

In practice the merged function are of two origins : allocated functions (also called sub-functions), which are part of the original set of function used to build the graphical model, and messages, which are byproduct functions create by the marginalisation operation.The type of the function is exactly the same in both case.

$$f \leftarrow f2 + f3$$
$$f(X1, X3, X4) \leftarrow f2(X3, X4) + f3(X1, X4)$$
$$f = \{(\mathbf{x1}, \mathbf{x3}, \mathbf{x4}, f2(x2, x3) + f3(x1, x4)) : \forall x1 \in D(X1), \forall x3 \in D(X3), \forall x4 \in D(X4)\}$$



Marginalisation This operation consist in extracting a summary of function over a subset of it scope, merging the information over the rest of this scope using the sum operator of the semiring. Marginalisation is the operation used to compute messages from one node to another, using the set of information present at this node. This operation creates a new function whose scope is the intersection of the scopes of the node computing the message and of the destination node.

In the example bellow, we assume that there are two nodes, one containing only f1, and one containing only f2, whose respective scopes it the union of the scopes of their containing functions. The node containing f1 computes its messages to the other one. The intersection of their scope contains only one element : the variable $X3$.

$f \leftarrow \max_{X1,X2} f1$
$f(X3) \leftarrow \max_{X1,X2} f1(X1, X2, X3)$
$f = \{(\mathbf{x3}, \max f1(x1, x2, x3) : \forall x1 \in D(X1) \forall x2 \in D(X2)) : \forall x3 \in D(X3)\}$

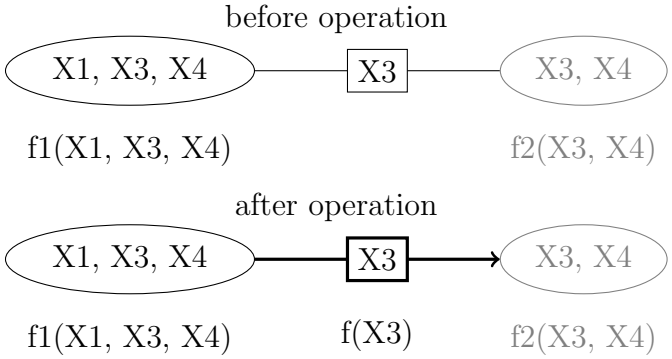


Fig : before : node (X3X4) asks node (X1X3X4) for a summary of its information over variable X3, after : node (X1X3X4) sends its summary, the function $f(X3)$ to node (X3X4).

3.2 Scheduling

Scheduling consist in defining the order of combination and marginalisation operation over a graphical model.

When the graph is acyclic, recursive scheduling starting from a root node permit to compute the correct marginal value for the variable of this node. Such a scheduling is guaranteed to terminate. When no caching is used, it is necessary to schedule inference from every nodes in the graphs. When caching is used, only to schedules are necessary : one from the leaves to the root then one from the root to the leaves.

When the graph is cyclic, recursive scheduling never terminates, a different scheduling heuristic and termination criterion must be used. The most common heuristic is to have a node update its information if its neighbours have updated they information. Two common termination criteria are : a fixed number of iteration, and a convergence detection. In the latter case, the algorithm is stopped either if receiving the information does not change

the state of a node, or if it changes it bellow a given threshold.

Other heuristics can be proposed, such as scheduling in priority the most informative nodes.

4 Implementation

The core object of Marae is the class `GMNode`. This is an abstract class with two abstract methods : `summariseInformation()`, corresponding to the marginalisation operation, and `mergeInformation()`, corresponding to the combination operation. This very class is used a basic building block for different graphical models. Application of graphical models for specific domain need to derive concrete class extending from `GMNode`.

4.1 Core classes

GMNode The class `GMNode<V,K>` is parametrized with `V` the type of the object of the scopes and `K` is the type of the codomain of the functions used to built graphical model (e.g. `Boolean` in the case of constraint satisfaction, `Double` in the case of probabilistic inference). The design choice made is to represent only clique node (the ellipse nodes), and store messages nodes (the square nodes) inside cliques nodes. As a consequence sent it is necessary to keep track of communication information inside the clique node.

Fields :

— `List<V> scope` : the scope of this node. Note that the parametrized type `V` doesn't need to be a variable, and can be any object. However in most application, it would correspond to mathematical variables, hence the name.

— `K information` : this is the combined information of all the pieces of information received by the node. It needs to be recomputed explicitly each time new information is available. Marginalisation proceeds with this object.

— `Map<GMNode<V,K>,K> receivedMessages` : set of message received by the node and the associated sender.

Methods :

return type	signature	description
`K`	`summariseInformation(List<V>)`	(abstract)
`K`	`mergeInformation()`	(abstract)
`void`	`updateMessage(K, GMNode<V,K>)`	
`List<K>`	`getSubFunctions()`	
`K`	`getInformation()`	
`Map<GMNode<V,K>,K>`	`getReceivedMessages()`	

GMNodeInitialiser `GMNode` are very generic and they are the entity manipulated by the algorithms. However, for specific application it is necessary to initialise derived nodes of

`GMNode`. This is the aim of `GMNodeInitialiser<C extends GMNode>`, an abstract class with only one abstract method `configureClique()`.

4.2 Extension for hypermatrices

Logical clauses, probability tables, weighted constraints, code words can all be represented using the hypermatrix. Having an algorithm that manipulates generic hypermatrices rather than specific instances makes it more generic. In Marae a specific instance of `GMNode`, the concrete class `HyperMatrixGDLNode<K>` precisely fit that purpose.

HyperMatrixGDLNode This class manipulates hypermatrices functions and posses one additional field of semiring type. It reuses the `HyperMatrix` and `Semiring` classes from the Moorea library.

HyperMatrixGDLNodeInitialiser Is used to initialise `HyperMatrixGDLNode` and requires to be initialised with a semiring.

Example The power of Marae reside in the simplicity of computing messages between cliques for any applications with generic methods.

```
HyperMatrixGDLNode c1 = ...; // scope: e.g. X1 X2
HyperMatrixGDLNode c2 = ...; // scope: e.g. X2 X3
HyperMatrix<DiscreteVariable, Integer> hm;
hm = c1.mergeInformation(); // scope: e.g. X1 X2
hm = c1.summariseInformation(c2); // scope: e.g. X2
```

5 Objects and Algorithms

5.1 Junction tree

`JunctionTree<C extends GMNode>` : contains two variables a graph of type `Graph<C>` and a hashtable. The hashtable is used to store information needed during the building process.

`JunctionTreeConstruction` : contains several functions used to build a junction tree from a set of functions. The only method needed from the outside of the class is :

return type	signature
`JunctionTree<C>`	`createJunctionTree(List<Function>, GMNodeInitialiser<C>)`

`JunctionTreeInference` : contains methods to generate different updates schedule and to execute them.

return type	signature	comment
List<C>	getUpdateScheduleRecursivelyFromNode(Graph<C>, C)	
void	performSequentialUpdateSchedule	
void	performParalleUpdateSchedule	not implemented
void	asynchronousUpdate(GMNode)	not implemented

Exemple

5.2 Factor graph

FactorGraph<C extends GMNode> : contains three variable a graph of type Graph<C>, and two lists of type List<C>, respectively for variable nodes and function nodes.

FactorGraphConstruction

return type	signature
FactorGraph<C>	createFactorGraph(List<Function>, GMNodeInitialiser<C>)

FactorGraphInference

return type	signature	comment
void	updateNTimesCylcicaly(FactorGraph<C>, int)	

Example