Crema Documentation

Release 1.0

Apr 12, 2023

QUICK START

1 Installation 3

2 Getting Started 5

3 Requirements 7

4 Factors 9

5 Graphical Models 15

6 Inference Engines 17

7 Domains 19

8 Domain Iterators 23

9 Credal Model 27

10 Credal Inference 29

11 Bayesian Network 31

12 Bayesian Inference 35

13 Contact and Support 37



Crema (CREdal Models Algorithms) is an Java library for inference in credal networks. The main features of Crema are:

- Provides a simple API for the definition of credal networks.
- CREMA embeds exact and approximate algorithms for credal inference.
- Models can be loaded and exported in UAI-based format for credal networks.

CHAPTER

ONE

INSTALLATION

Crema can be easily included at any maven project. For this, add the following code in the pom.xml:

```
<repositories>
    <repository>
        <id>cremaRepo</id>
        <url>https://raw.github.com/idsia/crema/mvn-repo/</url>
    </repository>
</repositories>

<dependencies>
    <dependency>
        <groupId>ch.idsia</groupId>
        <artifactId>crema</artifactId>
        <version>0.2.1</version>
        <scope>compile</scope>
    </dependency>
</dependencies>
```

CHAPTER

TWO

GETTING STARTED

As a short introduction to Crema, let us consider the following code snippet, in which a credal network with 2 nodes is defined. Credal sets are specified by enumerating the extreme points or vertices. Finally, a conditional query is performed.

```
package examples.docs;

import ch.idsia.crema.core.ObservationBuilder;
import ch.idsia.crema.core.Strides;
import ch.idsia.crema.factor.credal.vertex.separate.VertexFactor;
import ch.idsia.crema.factor.credal.vertex.separate.VertexFactorFactory;
import ch.idsia.crema.inference.ve.CredalVariableElimination;
import ch.idsia.crema.model.graphical.DAGModel;


public class Starting {
    public static void main(String[] args) {
        double p = 0.2;
        double eps = 0.0001;

        /*  CN defined with vertex Factor  */

        // Define the model (with vertex factors)
        DAGModel<VertexFactor> model = new DAGModel<>();
        int A = model.addVariable(3);
        int B = model.addVariable(2);
        model.addParent(B,A);

        // Define a credal set of the partent node
        VertexFactor fu = VertexFactorFactory.factory().domain(model.getDomain(A),␣
↪Strides.empty())
                .addVertex(new double[]{0., 1-p, p})
                .addVertex(new double[]{1-p, 0., p})
                .get();

        model.setFactor(A,fu);


        // Define the credal set of the child
        VertexFactor fx = VertexFactorFactory.factory().domain(model.getDomain(B), model.
↪getDomain(A))
```

(continues on next page)

(continued from previous page)

```
                .addVertex(new double[]{1., 0.,}, 0)
                .addVertex(new double[]{1., 0.,}, 1)
                .addVertex(new double[]{0., 1.,}, 2)
                .get();

        model.setFactor(B,fx);

        // Run exact inference
        CredalVariableElimination inf = new CredalVariableElimination();
        inf.query(model, ObservationBuilder.observe(B,0), A);

    }
}
```

CHAPTER

THREE

REQUIREMENTS

3.1 System

Build the Crema library requires Java 11 or higher and Maven (https://maven.apache.org).

Tests have been done under Linux Ubuntu, Windows 10, and macOS with openjdk 11, 12, and 16. Continuous integration tests are done using Ubuntu Latest and JDK 11 via GitHub Actions.

3.2 Package Dependencies

Crema contains the dependencies shown below which are managed using Maven.

- ch.javasoft.polco:polco:jar:4.7.1:compile
- colt:colt:jar:1.2.0:compile
- com.github.quickhull3d:quickhull3d:jar:1.0.0:compile
- com.google.code.findbugs:jsr305:jar:3.0.2:compile
- com.google.errorprone:error_prone_annotations:jar:2.3.4:compile
- com.google.guava:failureaccess:jar:1.0.1:compile
- com.google.guava:guava:jar:28.2-jre:compile
- com.google.guava:listenablefuture:jar:9999.0-empty-to-avoid-conflict-with-guava:compile
- com.google.j2objc:j2objc-annotations:jar:1.3:compile
- com.joptimizer:joptimizer:jar:3.5.1:compile
- com.opencsv:opencsv:jar:5.2:compile
- commons-beanutils:commons-beanutils:jar:1.9.4:compile
- commons-cli:commons-cli:jar:1.4:compile
- commons-collections:commons-collections:jar:3.2.2:compile
- commons-logging:commons-logging:jar:1.2:compile
- concurrent:concurrent:jar:1.3.4:compile
- javax.validation:validation-api:jar:1.1.0.Final:compile
- junit:junit:jar:4.13.1:compile
- log4j:log4j:jar:1.2.14:compile

- net.sf.lpsolve:lp_solve:jar:5.5.2:compile
- net.sf.trove4j:trove4j:jar:3.0.3:compile
- net.sourceforge.csparsej:csparsej:jar:1.1.1:compile
- org.apache.commons:commons-collections4:jar:4.4:compile
- org.apache.commons:commons-csv:jar:1.3:compile
- org.apache.commons:commons-lang3:jar:3.4:compile
- org.apache.commons:commons-math3:jar:3.6.1:compile
- org.apache.commons:commons-text:jar:1.8:compile
- org.apiguardian:apiguardian-api:jar:1.0.0:test
- org.checkerframework:checker-qual:jar:2.10.0:compile
- org.eclipse.persistence:org.eclipse.persistence.asm:jar:2.6.2:compile
- org.eclipse.persistence:org.eclipse.persistence.core:jar:2.6.2:compile
- org.glassfish:javax.json:jar:1.0.4:compile
- org.hamcrest:hamcrest-core:jar:1.3:compile
- org.jgrapht:jgrapht-core:jar:1.1.0:compile
- org.junit.jupiter:junit-jupiter-api:jar:5.4.2:test
- org.junit.jupiter:junit-jupiter-params:jar:5.4.2:test
- org.junit.platform:junit-platform-commons:jar:1.4.2:test
- org.opentest4j:opentest4j:jar:1.1.1:test
- org.slf4j:slf4j-api:jar:1.7.7:compile

3.3 External Dipendencies

In order to compile Crema from source code, two dependencies not available in Maven repositories need to be installed manually.

lpsolve

```
mvn org.apache.maven.plugins:maven-dependency-plugin:3.1.2:get -DgroupId=net.sf.lpsolve -
↪DartifactId=lp_solve -Dversion=5.5.2 -Dpackaging=jar -DremoteRepositories=https://raw.
↪github.com/idsia/crema/mvn-repo/
```

polco

```
mvn org.apache.maven.plugins:maven-dependency-plugin:3.1.2:get -DgroupId=ch.javasoft.
↪polco -DartifactId=polco -Dversion=4.7.1 -Dpackaging=jar -DremoteRepositories=https://
↪raw.github.com/idsia/crema/mvn-repo/
```

CHAPTER

FOUR

FACTORS

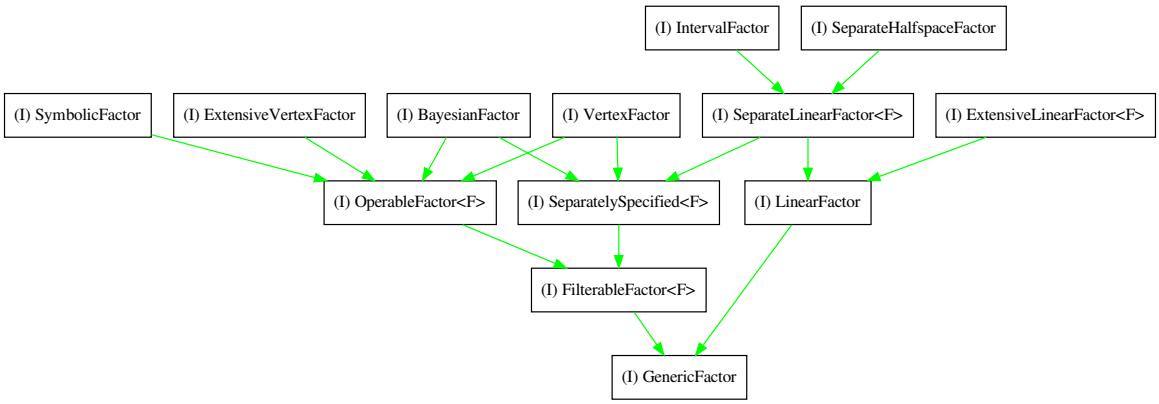
Table of Contents

- *Generic Factors Interfaces*
  - *Credal Factors*
  - *Bayesian Factors*
  - *Symbolic Factors*
- *Implementation*
- *Factory*
- *Conversion*

Crema supports different ways to represent the probability functions defined over the variables. A structure of different categorization and abstraction around factors have been implemented. At the top of this all we have the concept of `GenericFactor`.

The basic idea behind the whole class hierarchy is to have immutable implementation of different interface. As an example, a `VertexFactor` is an interface for many different implementation such as `VertexLogFactor` and `VertexFunctionFactor`. Inference algorithm should always work with the factor interfaces, such as `VertexFactor`. This let us hide the different kind of implementation and their complexity: to perform an inference the algorithm do not care how a factor implementation stores its data or if the data are generated by a function. This will grant Crema a high flexibility on multiple definitions of a factor.

4.1 Generic Factors Interfaces



The image above shows the main class hierarchy for the factors in Crema. The simplest definition of a factor is represented by the `GenericFactor` interface. This interface defines the two most basic methods of any factor: the `copy()` and the `getDomain()` methods.

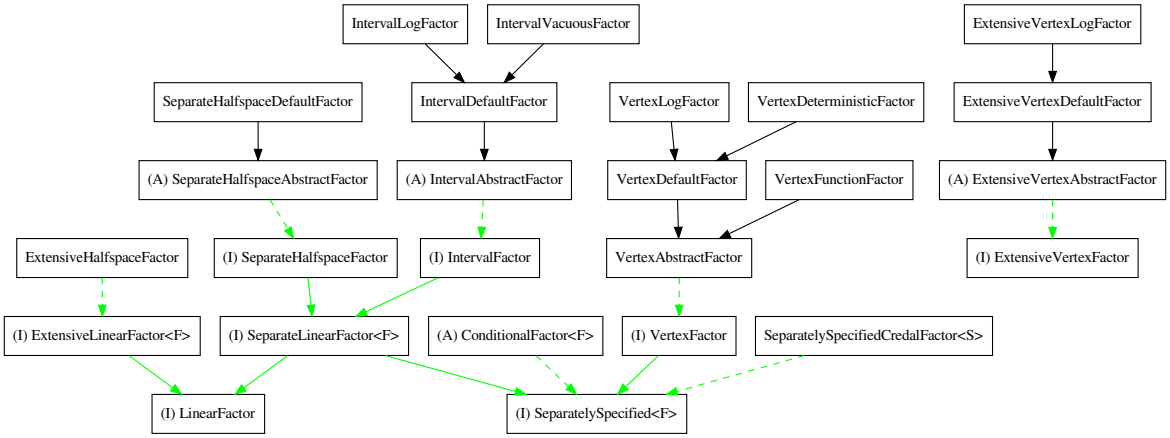
In Crema we have two main different kind of factors: `FilterableFactor<F>` and `LinearFactor`. The first is a group of factors that are able to perform the `filter(int, int)` operation on itself, while the second represents the group of factors defined with a linear problem.

Note: Note that these two groups are not separated: as an example, the `SeparateLinearFactor` is a particular type of factor that is defined with a linear problem but can also perform the filter operation.

Two other important groups are below the `FilterableFactor<F>`: the `OperableFactor<F>` and `SeparatelySpecified<F>` factors. The first defines the capability to perform operations such as `combine(factor)`, `marginalize(int...)`, `divide(factor)`, and `normalize(int...)`. These are all operation used by particular inference algorithm, in particular Bayesian-base algorithms. The second group, instead, defines the factors that have particular operations over their domains.

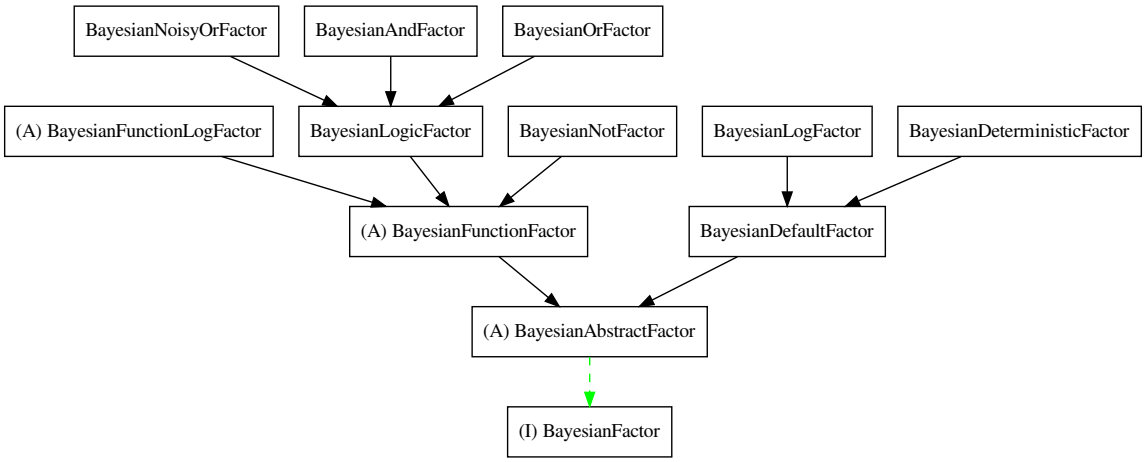
Below these main interfaces, we find the implementation of all the types of factors.

4.1.1 Credal Factors



The credal factors are the main factors that works with Crema. The idea of this library is to offer inferences algorithm to perform computation over these kind of factors. There we can find the most used factors, such as `VertexFactor` and `IntervalFactor` that are used to specify imprecise probability factors.

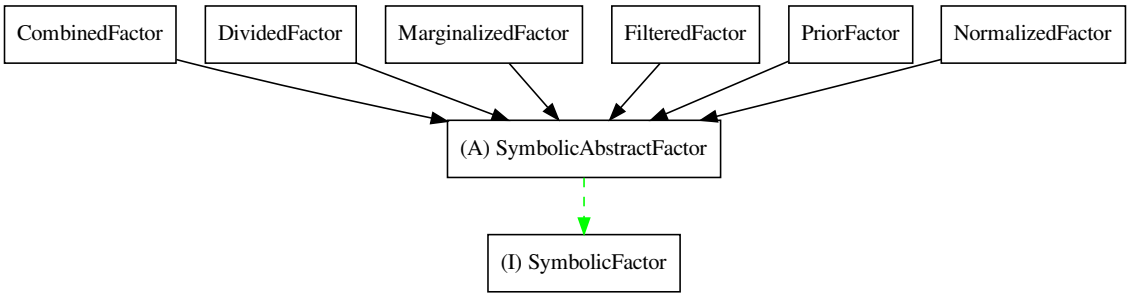
4.1.2 Bayesian Factors



`Bayesian Factors` are just a single type of factors that have an huge potential in Crema. These factors implements all the algebra to perform Bayesian inference over `BayesianNetworks` and other particular kind of models.

These factors also contains a special version of the bayesian factor: the `logic` factors. These factors implement a logic binary operation, such as `and`, `or`, or `not`, that can be used to implement logics in a Bayesian network.

4.1.3 Symbolic Factors



A `SymbolicFactor` is a special factor that does not perform any kind of operation. The use of these factors is to build a diagram of the operations so that it is possible to visualize the operations done and at the same time optimize and reuse them by changing the input factors.

Note: The `PriorFactor` is a special factor that can wrap any kind of `GenericFactor`. These are the inputs node of a workflow diagram produced by any inference algorithm that run over a `DAGModel` of `SymbolicFactors`.

4.2 Implementation

As stated before, the main idea is to let the algorithms work with a common interface that defines a factor while the implementation, in other words how the data are stored and managed for each type of factor, is hidden. For this reason, we have multiple implementations available for each factor interface.

Note: Since most of the implementation have a common set of fields and methods, the majority of these interfaces are first implemented in abstract classes. Then all the definition of factor extends this abstract class.

Across multiple hierarchies, we have some common way of implement a factor. As an example we can have `FunctionFactors`. These factors does not store the data in them but have a function (often a **lambda** function) that *generates* the requested data on the fly. One interesting implementation of this mechanics is available in the `BayesianLogicFactor` and the classes that extends this one. These logic factors implements a logic function and does not have any kind of storage, making them faster and more efficient at runtime.

Another common implementation pattern is to differentiate between factors in **log-space** and not. All the factors that are called like `*DefaultFactor` are the most simple implementation of a factor in a normal space. The factors that works and are optimizer for the **log-space**, instead, are called `*LogFactor`. Most of the factor interfaces offers two methods to access the values: one for log (as an example, `BayesianFactor#getLogValue(int)`) and one for normal space (following the example, `BayesianFactor#getValue(int)`).

4.3 Factory

Although all factors can be instantiated directly with the `new` keyord, many factor groups have a so called *factory* class. This is an helper class that simplify the build of the factors with helper methods and functions. All factor classes have the `factory()` static method that will instantiate the factory. All the methods of a factory can be chained together in a fluent way.

To obtain a factor once the factory setup is complete, just call one of the builder methods like `get()` or `log()`.

Note: Check the latest version of the JavaDoc to find more on this argument.

4.4 Conversion

In the package `ch.idsia.crema.factor.convert` we collected a conversion classes that can be used to convert one factor to another. These converter classes does not cover all the possible and doable combination. In certain cases, to perform a conversion, multiple converter need to be used.

CHAPTER

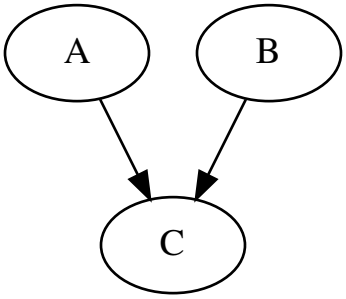
FIVE

GRAPHICAL MODELS

Crema includes a few packages to work with probabilistc graphical models. These include support the network representations, algorithms and modifiers.

5.1 Working with networks

As an exercise we will be creating a Bayesian Networks with 3 nodes connected in a V shape, as shown in the following picture.



Graphical Networks are implemented in the `models.graphical` package and they extend the `Graph` class. The class has a generic parameter to specify the concrete Factor used in order to express the probability models that parametrise the relationships between variables defined by the network.

There are currenlty 2 concrete implementations of graphical networks that differ in the underlying storage of the edges and nodes. From an inference and algorithmic point of view the actual implementation is irrelevant.

5.1.1 DAG Models

The main implementation for directed acyclic graphs is the `DAGModel` class. Crema uses JGraphT `SimpleGraph` object to store the actual graph.

For a Bayesian Network we will use a `BayesianFactor`.

```
DAGModel<BayesianFactor> model = new DAGModel<>();

model.addVariable(2); // C
model.addVariable(3); // A
model.addVariable(2); // B
```

Note: In its current implementation crema stores networks using a double adjacency lists. This is for each node in the network we store the collection of parents and children.

CHAPTER

SIX

INFERENCE ENGINES

Crema offers the generic interface `Inference<M, F>` to perform an inference on a model using the `query(model, evidence, variable)` method.

Note: If not specified otherwise, all the algorithms implementations are state-less; this means that each query are considered unique and there is no memory of the previous inference queries done with the same object.

The interface requires to specify two generic types: the input *model* type `M`, and the output factor type `F`.

Below an example on how to run an inference on a model built with `BayesianFactors` using the `BeliefPropagation` inference algorithm. This is the simplest way to run an inference.

```
Inference<DAGModel<BayesianFactor>, BayesianFactor> inf = new BeliefPropagation<>();
BayesianFactor factor = inf.query(model, A);
```

Note how the inference engine works on a model of type `DAGModel<BayesianFactor>` and that the output of the inference is an object of type `BayesianFactor`.

6.1 Evidence

In Crema, an evidence is just a map, an object of type `TIntIntMap`. If no evidence is needed, then the `Inference<M, F>` interface offers an utility `query(model, variable)` method without the need to pass an empty map.

6.2 Multiple queries

Crema offers other kind of inference interfaces. These interfaces are intended to offer a more optimized way to perform multiple and joined queries.

Note: In te current version, there are no algorithms that support and implement these interfaces. If an algorithm offers a special way to perform these queries, it will be required to instantiate it as its own class instead of using the `Inference<M, F>` interface.

CHAPTER

SEVEN

DOMAINS

Table of Contents

- *Domain interface*
- *SimpleDomain*
- *DomainBuilder*
- *Strides*
  - *Creating Strides*
    - *Working with Strides*

7.1 Domain interface

Domains in Crema are located in the `ch.idsia.crema.model` package. They are all instances of the `Domain` interface. This simple interface declares basic methods to query the domain about variables and their cardinality.

```
Domain domain = ...;
domain.getSizes();
domain.getVariables();
```

Note: Returned arrays should never be modified!

7.2 SimpleDomain

The simplest implementation of the `Domain` interface is the `SimpleDomain`. This class encapsulates two integer arrays. One with the variable labels and one with their cardinality.

```
Domain domain = new SimpleDomain(
                new int[]{1, 4, 6}, // variables 1, 4, 6
                new int[]{3, 2, 3}  // the corresponding cardinalities
);

assertTrue(domain.contains(6));
```

Warning: When creating a `SimpleDomain` the list of variables must be sorted! Crema will **not** automatically sort them, but for some operations will assume they are.

7.3 DomainBuilder

While creating a `SimpleDomain` by passing the arrays of variables and their sizes is possible and valid, a slightly more friendly method is available using the `DomainBuilder`. Laveraging the ellipses of Java the `DomainBuilder` class avoids the explicit creation of the arrays as shown in the following example.

```
Domain dom = DomainBuilder.var(1, 4, 6).size(3, 2, 3);
```

7.4 Strides

A more sophisticated and more frequently used implementation of the `Domain` interface is the `Strides` class. In addition to the arrays of variables and their cardinality, this class caches the cumulative sizes of the variables in the provided order. The access to this additional array is seldomly required by the end-user. They are mostly required to index parts of a probability table.

The `Strides` class offers a much richer set of functionalities both related to the domain itself and the aforementioned indexing of probability tables.

7.4.1 Creating Strides

We we first look at how `Strides` instances can be created conveniently.

Note: The variable's cardinalities are accumlated starting from the variable at index 0.

```
Strides domain = new Strides(
                 new int[]{1, 4, 6}, // variables 1, 4, 6
                 new int[]{3, 2, 3}  // the corresponding cardinalities
);
```

Again, just as with the `SimpleDomain`, creating the object specifying the arrays is valid, but not the most readable solution. The following example shows an alternative way of creation where variables are added along with their cardinality.

```
Strides other = Strides.as(1, 3).and(4, 2).and(6, 3);
```

Alternative ways to create strides are based on operations on them. Generally Domains are considered unmutable objects and any alteration will result in a new instance.

```
// remove variable 4 and 6
Strides smaller = domain.remove(4, 6);
```

A number of common set operations are available:

- union
- intersect

- remove

Working with Strides

CHAPTER

EIGHT

DOMAIN ITERATORS

When interacting with factors and working with indexing of items in the library you will definitely need to address the issue of the variables ordering and iterators.

In our implementation when an iterator over a domain is requested it will return an instance of an `IndexIterator`. This Java iterator will visit the different instantiations of the variables by means of an integer index. This index enumerates all the possible configurations of the domain's variables, sorted order with the variable at index 0 being the least significant.

In its original ordering and domain the index will simply be an increasing integer value. In the following example we show this on a domain defined on the binary variables 0 and 2 and the ternary variable 3.

```
Strides domain = Strides.var(0,2).and(3, 2).and(2,3);
IndexIterator iterator = domain.getIterator();
while(iterator.hasNext()) {
    System.out.print(iterator.next() + " ");
}
```

which will output:

```
0 1 2 3 4 5 6 7 8 9 10 11
```

Asking the domain we can olso convert this index to the actual states for the variables. This can be achieved with a call to `getStatesFor` and for the code above will generate the sequence of states configurations shown in the following table:

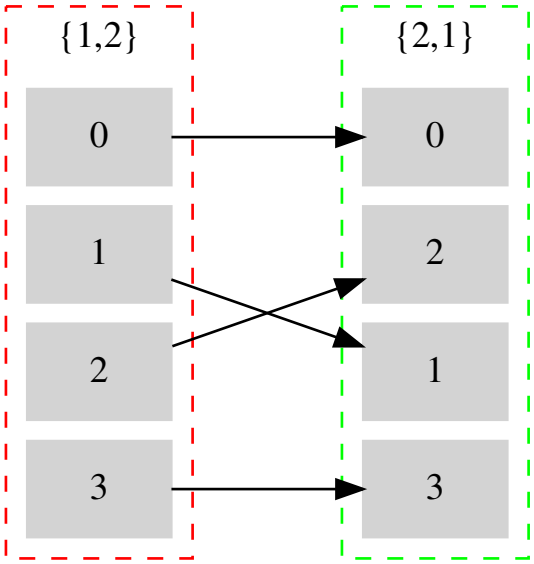
Offset	Variable		
	0	2	3
0	0	0	0
1	1	0	0
2	0	1	0
3	1	1	0
4	0	2	0
5	1	2	0
6	0	0	1
7	1	0	1
8	0	1	1
9	1	1	1
10	0	2	1
11	1	2	1

This is obvioulsy quite a usesless use of an iterator. An simple increasing integer would be enough. The really interesting use of iterators arises when we want to index a domain fixing some variable, reordering them or even using a larger domain. These uses are all explored in further detail hereafter.

8.1 Modified variable order

Crema uses mostly a global ordering of the variables. This, however, is not always the most natural and confortable way to index data. To overcome this issue crema offers in some methods to iterate over the indices using a different ordering of the variable.

So if one has a domain that is defined over two binary variables $\{1, 2\}$, a reordered iterator over the same domain but with inverse order, will visit all the indices in the order shown below.



In code creating this iterator can be done directly from the Strides class, as shown in the following code snippet:

```
Strides domain = Strides.var(1,2).and(2,2);
IndexIterator iter = domain.getReorderedIterator(new int[] {2,1});
int original = 0;
while (iter.hasNext()) {
    int offset = iter.next();
    // some operation using the updated order
    data[offset] = input[original++];
}
```

8.2 Wider domain

Another useful way to traverse a domain is to expand it with attitional variables. In such configuration the iterator will not move for different instantiations of these additional variables.

In the following code snipped a domain over variable 0 is visited moving both variable 1 and variable 0.

```
Strides domain = Strides.var(1,3);
Strides bigger_domain = Strides.var(1,3).and(0,2);

IndexIterator iter = domain.getIterator(bigger_domain);

int target = 0;
while (iter.hasNext()) {
    int offset = iter.next();
    // some operation using the offset
    data[target++] = input[offset];
}
```

In the following table we show the evolution of the `offset` within the original domain for the different states configurations of the two variables of the extended domain.

target	Var 0	Var 1	offset
0	0	0	0
1	1	0	0
2	0	1	1
3	1	1	1
4	0	2	2
5	1	2	2

8.3 Filtered Iterators

One final way to address indexing is by conditioning on some variables. In this setting the domain will have some variables fixed to some state while the others are going to be iteratred.

In the following example we will take a domain over 3 variables (2 binary and a ternary one) and iterate over it blocking one of the binary variable.

```
Strides domain = Strides.var(2,3).and(0,2).and(3,2);
IndexIterator iter = domain.getFiteredIndexIterator(0,1);
while (iter.hasNext()) {
    int offset = iter.next();
    System.out.println(offset);
}
```

Offset	Variable		
	0	2	3
1	1	0	0
3	1	1	0
5	1	2	0
7	1	0	1
9	1	1	1
11	1	2	1

CHAPTER

NINE

CREDAL MODEL

9.1 Credal Set Specification

For the definition of a credal set, the domains should be first specified. Discrete variable domains in Crema are managed with objects of class `Strides`. Then, for the definition of a credal set defined by its vertices, create an object of class `VertexFactor` as shown below.

```
// Define the domains
Strides strides_left = DomainBuilder.var(0).size(3).strides();
Strides strides_right = Strides.empty();

double p = 0.2;

// define a marginal vertex factor
VertexFactor f0 = VertexFactorFactory.factory().domain(strides_left, strides_right)
                .addVertex(new double[]{p, 0, 1 - p})
                .addVertex(new double[]{0, p, 1 - p})
                .get();
```

Similarly, a conditional credal set can be define as shown in the following code.

```
// define a conditional vertex factor
strides_left = DomainBuilder.var(1).size(2).strides();
strides_right = DomainBuilder.var(0).size(3).strides();

VertexFactor f1 = VertexFactorFactory.factory().domain(strides_left, strides_right) //␣
↪K(vars[1]|[0])
                // when adding the extreme points, value of the conditioning variables␣
↪should be specified
                .addVertex(new double[]{0.4, 0.6}, 0)
                .addVertex(new double[]{0.2, 0.8}, 0)

                .addVertex(new double[]{0.3, 0.7}, 1)
                .addVertex(new double[]{0.4, 0.6}, 1)

                .addVertex(new double[]{0.3, 0.7}, 2)
                .addVertex(new double[]{0.4, 0.6}, 2)

                .get();
```

Crema also allows the specification of credal sets by defining its constraints. This is done with the class `SeparateHalfspaceFactor`.

```
SeparateHalfspaceFactor f0_constr = SeparateHalfspaceFactorFactory.factory().
↪domain(strides_left, Strides.empty())

                // add constraints
                .constraint(new double[]{1., 1., 0.,}, Relationship.EQ, p)
                .constraint(new double[]{0., 0., 1.,}, Relationship.EQ, 1 - p)

                // normalization constraint
                .constraint(new double[]{1., 1., 1.,}, Relationship.EQ, 1)

                // positive constraints
                .constraint(new double[]{1., 0., 0.,}, Relationship.GEQ, 0)
                .constraint(new double[]{0., 1., 0.,}, Relationship.GEQ, 0)
                .constraint(new double[]{0., 0., 1.,}, Relationship.GEQ, 0)
                .get();
```

9.2 Credal Network Specification

For defining a credal network, create an object of class `SparseModel`, specify the structure of the graph and associate the factors.

```
// Define the structure
DAGModel<VertexFactor> cnet = new DAGModel<>();
int X0 = cnet.addVariable(3);
int X1 = cnet.addVariable(2);
cnet.addParent(X1, X0);

// Set the factors
cnet.setFactor(X0, f0);
cnet.setFactor(X1, f1);
```

CHAPTER

TEN

CREDAL INFERENCE

Crema provides exact and approximate inference algorithms over credal networks. For the exact one, create an object of class `CredalVariableElimination` and run the query. The result is an object of class VertexFactor.

```
// set up the inference and run the queries
CredalVariableElimination inf = new CredalVariableElimination();
VertexFactor res1 = inf.query(cnet, ObservationBuilder.observe(X0, 0), X1);
VertexFactor res2 = inf.query(cnet, X0);
```

Approximate inference can be done by means of linear programming. For this, create the an object of class `CredalApproxLP` and then run the query. Note that the output is an `IntervalFactor`.

```
// set up the inference and run the queries
CredalApproxLP<SeparateHalfspaceFactor> inf = new CredalApproxLP<>();
IntervalFactor res1 = inf.query(cnet, ObservationBuilder.observe(X0, 0), X1);
IntervalFactor res2 = inf.query(cnet, X1);

double[] lbound = res1.getLower();
double[] ubound = res1.getUpper();
```

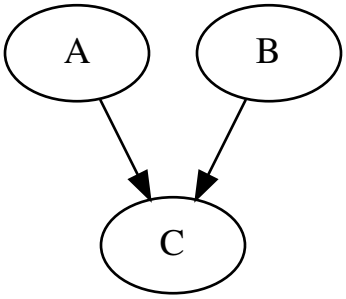
CHAPTER

ELEVEN

BAYESIAN NETWORK

Lets start with an example of *Bayesian Network*. We will create a very small model and perform some simple query.

The network will contain 3 variables connected in a V-shape as shown in the following figure:



The first thing to do is to declare our Bayesian Network, the variables, and assign the parents for each variable. In this case, all variables are binary.

Note: The `BayesianNetwork` is just a wrapper class of the `DAGModel<BayesianFactor>` class.

```
BayesianNetwork model = new BayesianNetwork();

// variables declaration
int A = model.addVariable(2);
int B = model.addVariable(2);
int C = model.addVariable(2);

// parents assignments
model.addParent(C, A);
model.addParent(C, B);
```

For each variable, the model assign a domain. We need such information to build the factors.

```
Domain domA = model.getDomain(A);
Domain domB = model.getDomain(B);
Domain domC = model.getDomain(C, A, B);
```

Finally, there we build a factors for each of the variables. In this example, we show an overview of the many possible ways to instantiate a factor.

```
BayesianFactor[] factors = new BayesianFactor[3];

factors[A] = new BayesianDefaultFactor(domA, new double[]{.8, .2});

factors[B] = BayesianFactorFactory.factory().domain(domA)
                .set(.4, 0)
                .set(.6, 1)
                .get();

factors[C] = BayesianFactorFactory.factory().domain(domC)
                .set(.3, 0, 0, 0)
                .set(.7, 0, 0, 1)
                .set(.5, 0, 1, 0)
                .set(.5, 0, 1, 1)
                .set(.4, 1, 0, 0)
                .set(.6, 1, 0, 1)
                .set(.6, 1, 1, 0)
                .set(.4, 1, 1, 1)
                .get();

// factor assignment
model.setFactors(factors);
```

Factor A We instantiate a new `BayesianDefaultFactor` from the domain and an array of double values.

Factor B We use the factory to set the probabilities for states `0` and `1`.

Factor C We use the factory to set the whole joint probability table for variable `C` using the states of all the variables in the domain. In order: `C`, `A`, `B`. Compare the order with the variable order in the definition of `domC`.

Full example:

```
package example;

import ch.idsia.crema.core.Domain;
import ch.idsia.crema.factor.bayesian.BayesianDefaultFactor;
import ch.idsia.crema.factor.bayesian.BayesianFactor;
import ch.idsia.crema.factor.bayesian.BayesianFactorFactory;
import ch.idsia.crema.model.graphical.BayesianNetwork;

/**
 * Author:  Claudio "Dna" Bonesana
 * Project: crema
 * Date:    27.10.2021 11:33
 */
public class BayesianNetworkExample {
        public static void main(String[] args) {
```

(continues on next page)

(continued from previous page)

```
                // [1] model declaration
                BayesianNetwork model = new BayesianNetwork();

                // variables declaration
                int A = model.addVariable(2);
                int B = model.addVariable(2);
                int C = model.addVariable(2);

                // parents assignments
                model.addParent(C, A);
                model.addParent(C, B);

                // [2] domains definitions
                Domain domA = model.getDomain(A);
                Domain domB = model.getDomain(B);
                Domain domC = model.getDomain(C, A, B);

                // [3] factor definition
                BayesianFactor[] factors = new BayesianFactor[3];

                factors[A] = new BayesianDefaultFactor(domA, new double[]{.8, .2});

                factors[B] = BayesianFactorFactory.factory().domain(domA)
                                .set(.4, 0)
                                .set(.6, 1)
                                .get();

                factors[C] = BayesianFactorFactory.factory().domain(domC)
                                .set(.3, 0, 0, 0)
                                .set(.7, 0, 0, 1)
                                .set(.5, 0, 1, 0)
                                .set(.5, 0, 1, 1)
                                .set(.4, 1, 0, 0)
                                .set(.6, 1, 0, 1)
                                .set(.6, 1, 1, 0)
                                .set(.4, 1, 1, 1)
                                .get();

                // factor assignment
                model.setFactors(factors);

                // [4] end
        }
}
```

CHAPTER

TWELVE

BAYESIAN INFERENCE

Crema provides useful algorithms for both precise and approximate inference on Bayesian networks and graphs.

Table of Contents

- *Exact Inference*
  - *Variable Elimination*
  - *Belief Propagation*
- *Approximate Inference*
  - *Sampling*
  - *Loopy Belief Propagation*

12.1 Exact Inference

12.1.1 Variable Elimination

The `VariableElimination` inference algorithm uses a given *elimination sequence* in order to perform the inference. Each elimination sequence depends on the structure of the model and the variable to query.

The implementation of this algorithm in Crema need an *algebra* to work. If this algebra is available externally, it is possible to use the `VariableElimination<F>` implementation; while if the existing `FactorAlgebra` is enough for the used factor, the wrapper class `FactorVariableElimination<F>` can be used.

12.1.2 Belief Propagation

The `BeliefPropagation` inference algorithm works by analyzing the model and build a `JunctionTree`. Then it will use the *message passing* algorithm to performing the inference.

Each call to the `query()` method will build a new `JunctionTree` from zero.

To perform an inference on a variable, as an example if you want the marginal of `P(A)`, use the `query()` method as in the example below:

```
// P(A)
BayesianFactor pA = inf.query(model, A);
```

If you want to use evidence, you need to create first a `TIntIntHashMap` that will include the state of the various variables, in the belo case we query for `P(A | B=0)`:

```
// P(A | B=0)
TIntIntHashMap evidence = new TIntIntHashMap();
evidence.put(B, 0);

BayesianFactor pAb0 = inf.query(model, evidence, A);

// P(A | B=0, C=1)
evidence = new TIntIntHashMap();
evidence.put(B, 0);
evidence.put(C, 1);

BayesianFactor pAb0c1 = inf.query(model, evidence, A);
```

This algorithm offers other ways to perform an inference. It is possible to build such tree once and query multiple variables at the same time. First instantiate the inference algorithm object. The inference engine will build an internal `JunctionTree` that will be used for the following queries.

```
BeliefPropagation<BayesianFactor> bp = new BeliefPropagation<>();
```

Then remember to call `fullPropagation()` to update the tree. This will return the posterior of a variable considered the root of the internal `JunctionTree`. This root variable is also the query variable.

```
factor = bp.fullPropagation(model, A);

// Perform the distribution step
bp.distributingEvidence();

// Perform the collection step
factor = bp.collectingEvidence(A);
```

12.2 Approximate Inference

12.2.1 Sampling

Crema offers two implementation of `StochasticSampling` for `BayesianFactor`: the `LogicSampling` and the `LikelihoodWeightingSampling`. These sampling algorithms have different levels of precision based on the number of iterations performed.

12.2.2 Loopy Belief Propagation

This is an approximate version of the `BeliefPropagation`: is uses the same *message passing* algorithm but without the burden to build a *junction tree*. The performance, and quality, of the algorithm can be managed by the number of iterations to execute.

CHAPTER

THIRTEEN

CONTACT AND SUPPORT



Crema has been developed at the Swiss AI Lab IDSIA (Istituto Dalle Molle di Studi sull'Intelligenza Artificiale), a not-for-profit research institute for Artificial Intelligence.

The members of the development and research team are:

- David Huber (david@idsia.ch)
- Rafael Cabañas (rcabanas@idsia.ch)
- Alessandro Antonucci (alessandro@idsia.ch)
- Marco Zaffalon (zaffalon@idsia.ch)
- Claudio Bonesana (claudio@idsia.ch)

If you have any question, please use Github issues.