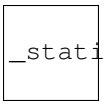
Crema Documentation

Release 1.0

Aug 18, 2021

QUICK START



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

GETTING STARTED

As a short introduction to Crema, let us consider the following code snippet, in which an 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))
                .addVertex(new double[]{1., 0.,}, 0)
                .addVertex(new double[]{1., 0.,}, 1)
                .addVertex(new double[]{0., 1.,}, 2)
                .get();
```

(continues on next page)

(continued from previous page)

```
        model.setFactor(B,fx);

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

    }
}
```

CHAPTER

TWO

REQUIREMENTS

2.1 System

Crema requires Java 11 or higher and maven (https://maven.apache.org). Tests have been done under Linux Ubuntu and macOS with openjdk 11 and 12.

2.2 Package Dependencies

Crema contains the dependencies shown below which are transparently managed with 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

CHAPTER

THREE

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>0.1.6</version>
        <scope>compile</scope>
    </dependency>
</dependencies>
```

CHAPTER

FOUR

MODEL DEFINITION

4.1 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.

```
///// code 1 .... LINE 13


// 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
```

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

```
/// code 2


// 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)
```

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

```
        .get();


```

(continues on next page)

(continued from previous page)

```
// code 3

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
```

4.2 Credal Network Specification

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

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



// Define the structure

DAGModel<VertexFactor> cnet = new DAGModel<>();
```

CHAPTER

FIVE

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.

```
                .addVertex(new double[]{0.4, 0.4}, 1)
                .addVertex(new double[]{0.2, 0.8}, 2)
                .addVertex(new double[]{0.1, 0.9}, 2)
                .get();
cnet.setFactor(X0,fa);
// set up the inference and run the queries
```

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`.

```
                .constraint(new double[]{0,1}, Relationship.LEQ, 0.9, 2)
                .get();
cnet.setFactor(X0,fa);

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

CHAPTER

SIX

BAYESIAN INFERENCE

Crema provides useful algorithm for precise inference on Bayesian networks.

6.1 Belief Propagation

The `BeliefPropagation` inference algorithm works on the `BayesianFactors` of a `BayesainNetwork`.

First instantiate the inference algorithm object using the model. The inference engine will build an internal `JunctionTree` that will be used for the following queries. Then remember to call `fullPropagation()` to update the model. This will return the posterior of a variable considered the root of the internal `JunctionTree`.

```
// perform a full update
factor = bp.fullPropagation(model, A);
```

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:

```
BayesianFactor pA = bp.query(model, A);

// Inference with evidence

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

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)`:

```
BayesianFactor pAB0 = bp.query(model, evidence, A);

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

Full example:

```
package examples;

import ch.idsia.crema.factor.bayesian.BayesianFactor;
import ch.idsia.crema.factor.bayesian.BayesianFactorFactory;
import ch.idsia.crema.inference.bp.BeliefPropagation;
import ch.idsia.crema.model.graphical.BayesianNetwork;
import gnu.trove.map.hash.TIntIntHashMap;
```

(continues on next page)

(continued from previous page)

```
public class BeliefPropagation {

        public static void main(String[] args) {
                /* Define your Bayesian Network model */

                BayesianNetwork model = new BayesianNetwork();
                int A = model.addVariable(2);
                int B = model.addVariable(2);
                int C = model.addVariable(2);

                model.addParent(B, A);
                model.addParent(C, A);

                // define the Bayesian Factors
                BayesianFactor[] factors = new BayesianFactor[3];

                factors[A] = BayesianFactorFactory.factory().domain(model.
↪getDomain(A))
                                .data(new int[]{A}, new double[]{.4, .6})
                                .get();
                factors[B] = BayesianFactorFactory.factory().domain(model.getDomain(A,
↪ B))
                                .data(new int[]{B, A}, new double[]{.3, .7, .7, .3})
                                .get();
                factors[C] = BayesianFactorFactory.factory().domain(model.getDomain(A,
↪ C))
                                .data(new int[]{C, A}, new double[]{.2, .8, .8, .2})
                                .get();

                // Assign factors to model
                model.setFactors(factors);

                // Instantiate the inference algorithm over BayesianFactors using the␣
↪model
                BayesianFactor factor;

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

                // perform a full update
                factor = bp.fullPropagation(model, A);

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

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

                // Simple Inference

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

                // Inference with evidence

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

(continues on next page)

(continued from previous page)

```
                evidence.put(B, 0);

                BayesianFactor pAB0 = bp.query(model, evidence, A);

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

                BayesianFactor pAB0C1 = bp.query(model, evidence, A);
        }
}
```

CHAPTER

SEVEN

FACTORS

Credo 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`.

CHAPTER

EIGHT

DOMAINS

Table of Contents

8.1 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!

8.2 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 = new SimpleDomain(
int[]{1, 4, 6}, // variables 1, 4, 6
int[]{3, 2, 3}  // the corresponding cardinalities


True(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.

8.3 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.

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

8.4 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.

8.4.1 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.

```
s domain = new Strides(
int[]{1, 4, 6}, // variables 1, 4, 6
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.

```
s 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.

```
ove variable 4 and 6
s smaller = domain.remove(4, 6);
```

A number of common set operations are available:

- union
- intersect
- remove

Working with Strides

CHAPTER

NINE

GRAPHICAL MODELS

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

9.1 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.

9.1.1 DAG Models

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

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

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

addVariable(2); // C
addVariable(3); // A
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

TEN

BAYESIAN NETWORK EXAMPLE

Lets start with an example of Bayesian Network. Later we will look into more detail how to create Credal Networks and how to work with factors directly.

We will create a vary small Bayesian Network and perform some simple query. The network will contain 3 variables connected in a V-shape as shown in the following figure

CHAPTER

ELEVEN

CONTACT AND SUPPORT

notes/../_static/img/idsia.png

Crema has been developed at the Swiss AI Lab IDSIA (Istituto Dalle Molle di Studi sull'Intelligenza Artificiale). 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.