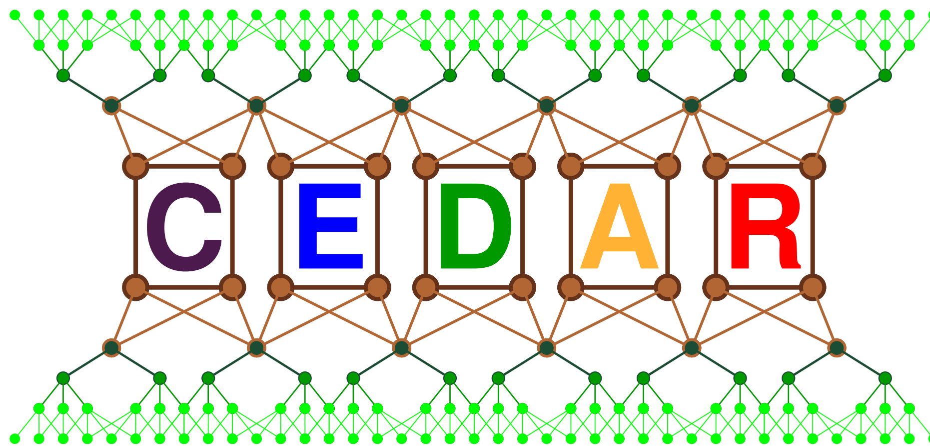


Technical Report Number 9

The `Cedar.Gdl` Java Library for the Generalized Distributive Law

Design and Implementation

Kevin Sancho and Hassan Aït-Kaci

July 2014

Publication Note

This report is based on the work done by the first author during his internship in the CEDAR Project toward the obtention of his MSc degree at the Université Claude Bernard Lyon 1, on a topic proposed by Prof. Hassan Aït-Kaci [32].

Contact information:

LIRIS - UFR d'Informatique
Université Claude Bernard Lyon 1
43, boulevard du 11 Novembre 1918
69622 Villeurbanne cedex
France

Phone: +33 (0)4 27 46 57 08
Email: kevin.steven.sancho@gmail.com
hassan.ait-kaci@univ-lyon1.fr

CEDAR Project's Web Site: cedar.liris.cnrs.fr

Copyright © 2014 by the CEDAR Project.

This work was carried out as part of the CEDAR Project (Constraint Event-Driven Automated Reasoning) under the *Agence Nationale de la Recherche* (ANR) Chair of Excellence grant N^o **ANR-12-CHEX-0003-01** at the *Université Claude Bernard Lyon 1* (UCBL). It may not be copied nor reproduced in whole or in part for any commercial purpose. Permission to copy in whole or in part without payment of fee is granted for non-profit educational and research purposes provided that all such whole or partial copies include the following: a notice that such copying is by permission of the UCBL, with an acknowledgement of the authors and individual contributors to the work; and all applicable portions of the copyright notice. Copying, reproducing, or republishing for any other purpose shall require a license with payment of a fee to the UCBL. **All rights reserved.**

CEDAR **Technical Report Number 9**

The `Cedar.Gdl` Java Library for the Generalized Distributive Law

Design and Implementation

Kevin Sancho and Hassan Aït-Kaci

kevin.steven.sancho@gmail.com, hassan.ait-kaci@univ-lyon1.fr

July 2014

Abstract

The Generalized Distributive Law (GDL) formulation and algorithm were proposed in 2000 by Srinivas Aji and Robert McEliece. The GDL is a parametric expression-evaluation optimization method that may be instantiated on any specific commutative semiring structure. It can be used to express various algorithms that have been independently designed in domains such as Information Theory, Digital Communications, Statistics, Artificial Intelligence, *etc.*, ... In this report, we describe the design and implementation of an abstract Java class library for the GDL. This library is a generic API meant to be instantiated with specific computation structures. It provides a runnable specification of the GDL in terms of abstract operations of a commutative semiring—*i.e.*, a set with an addition $(+)$ and a multiplication $(\times)$, the latter distributing over the former. This allows the generic efficient evaluation of expressions using a dynamic-programming two-way message-passing algorithm on an arborescent structure called a *Junction Tree*. We used our library to regenerate as running instances two algorithms described by Aji and McEliece: the Fast-Hadamard Transform (FHT) algorithm on any finite Abelian group, and Judea Pearl's Belief-Propagation Bayesian reasoner. We also used it to generate a new instance for Constraint Satisfaction.

Keywords: Abstract Java Library, Distributive Law, Junction Tree, Belief Propagation, Contraint Satisfaction

Table of Contents

1 Introduction **1**
- 1.1 Background . . . 1
- 1.2 Motivation . . . 2
- 1.3 Organization of contents . . . 2

2 Bayesian Networks **3**
- 2.1 Generalities . . . 3
- 2.2 The GDL in the literature . . . 8

3 The Generalized Distributive Law Algorithm **9**
- 3.1 The GDL . . . 9
- 3.2 Junction trees . . . 11
- 3.3 The message-passing algorithm . . . 16

4 The `Cedar.Gdl` Library Implementation **19**
- 4.1 The abstract core . . . 19
- 4.2 Classes implementing the GDL . . . 22

5 Original Contribution **23**
- 5.1 Constraint satisfaction with the GDL . . . 23
- 5.2 A modification of the GDL algorithm . . . 25
- 5.3 Modeling Allen's interval algebra with the GDL . . . 28

6 Implemented Instances of the `Cedar.Gdl` Library **29**
- 6.1 The Fast-Hadamard Transform . . . 30
- 6.2 Judea Pearl's belief propagation . . . 30
- 6.3 Constraint processing . . . 31
  - 6.3.1 Constraint-satisfaction problems . . . 32
  - 6.3.2 Constraint solving . . . 33

7 Conclusion **33**
- 7.1 Recapitulation . . . 33
- 7.2 Perspectives . . . 33

A Correctness of the construction of Section 5.1 **34**

B Correctness of the GDL Modification of Section 5.2 **34**
- B.1 First method . . . 34
- B.2 Second method . . . 35

1 Introduction

1.1 Background

The basic algebraic distributive law for a multiplicative operation $\times$ over an additive operation $+$ states that $(a \times b) + (a \times c) = a \times (b + c)$, for all a, b, and c. While the left-hand side contains two multiplications and one addition, the right-hand side contains only one addition and one multiplication. If one minds the number of operations, the difference may appear small in this case. But in many cases, the number of calculations saved can be important. Consider for example the expression:

$$\alpha(x, z) = \sum_{y,w \in A} f(x, y, z) \times g(x, w). \tag{1}$$

After transformation by distributivity, Equation (1) becomes:

$$\alpha(x, z) = \sum_{y \in A} f(x, y, z) \times \sum_{w \in A} g(x, w) \tag{2}$$

Equation (1) requires $|A|^2$ operations, while Equation (2) only requires $2 \times |A|$ operations.

The Generalized Distributive Law (GDL) [2] is a technique that uses distributivity to minimize the number of operations in expressions of a commutative semiring using a message-passing algorithm. This generalization leads to a large family of fast algorithms, such as Viterbi's algorithm [34], the Fast-Fourier Transform (FFT),[1] and many others. The interest of this algorithm comes from the fact that it applies to situations where the notion of addition and multiplication are abstracted. All computation is expressed in the appropriate framework of a *commutative semiring*.

DEFINITION 1 *A* commutative semiring *is a set $\mathcal{K}$ equipped with two binary operations $(+)$ and $(\times)$ such that:*

- $\langle \mathcal{K}, + \rangle$ *is a commutative monoid;* i.e.*:*
  1. *there is an identity element* 0 *such that* $\forall k \in \mathcal{K}$, $k + 0 = k$*;*
  2. $+$ *is associative;*
  3. $+$ *is commutative;*
- $\langle \mathcal{K}, \times \rangle$ *is a commutative monoid;* i.e.*:*
  1. *there is an identity element* 1 *such that* $\forall k \in \mathcal{K}$, $k \times 1 = k$*;*
  2. $\times$ *is associative;*
  3. $\times$ *is commutative;*
- $\times$ *distributes over* $+$*;* i.e.*:* $\forall a, b, c \in \mathcal{K}$, $(a \times b) + (a \times c) = a \times (b + c)$.

[1] https://en.wikipedia.org/wiki/Fast_Fourier_transform

Figure 1 shows examples of commutative semirings where $\mathcal{R}$ denotes any commutative ring, $\mathcal{R}[x]$ and $\mathcal{R}[x, y, \ldots]$ denote respectively univariate and multivariate polynomials with coefficients in $\mathcal{R}$, S is an arbitrary set, and $\mathcal{L}$ is any distributive lattice.

	$\mathcal{K}$	$+$	0	$\times$	1
(1)	$\mathcal{R}$	$+$	0	$\times$	1
(2)	$\mathcal{R}[x]$	$+$	0	$\times$	1
(3)	$\mathcal{R}[x, y, \ldots]$	$+$	0	$\times$	1
(4)	$[0, +\infty)$	$+$	0	$\times$	1
(5)	$(0, +\infty]$	min	$+\infty$	$\times$	1
(6)	$[0, +\infty)$	max	0	$\times$	1

	$\mathcal{K}$	$+$	0	$\times$	1
(7)	$(-\infty, +\infty]$	min	$+\infty$	$+$	0
(8)	$[-\infty, +\infty)$	max	$-\infty$	$+$	0
(9)	$\{\mathfrak{false}, \mathfrak{true}\}$	$\mathfrak{or}$	$\mathfrak{false}$	$\mathfrak{and}$	$\mathfrak{true}$
(10)	$\mathbf{2}^S$	$\cup$	$\emptyset$	$\cap$	S
(11)	$\mathcal{L}$	$\wedge$	$\perp$	$\vee$	$\top$
(12)	$\mathcal{L}$	$\vee$	$\top$	$\wedge$	$\perp$

Figure 1: Examples of commutative semirings

1.2 Motivation

Since the essence of the Generalized Distributive Law is that it abstracts the algebraic structure of a commutative semiring, it makes sense to propose an abstract library architecture for it. This abstraction is the key for implementing several algorithms related to different domain in a single library. This can then provide a tool that could be used for diverse applications like inference and decision making in Bayesian Networks (BNs). Having one generic tool is not only a boon for creating new algorithms as GDL instances, but this also gives the opportunity to focus on optimizing the generic code once and for all independently of the specific nature of the semiring operations involved.

The challenge in designing and implementing a Java library for the GDL thus resides in enabling this genericity. It should be capable of handling a vast variety of semiring instances of all sorts. The solution is the creation of abstract classes that specify in an abstract manner the GDL algorithm. Using the library is through the creation of an instance for a specific commutative semiring. The work we report here describes a Java implementation and use of such an abstract library.

1.3 Organization of contents

The rest of this document is organized as follows. Section 2 is a review of Bayesian networks, presenting the context of the GDL and the state of the art related to this technology. Section 3 gives definitions necessary to understand the GDL, and what is required for this algorithm to be cast into a generic software architecture. Section 4 presents the architecture, and discusses the implementation, of the library. Section 5 is our original contribution as far as: using the GDL the context of constraint processing (Section 5.1); introducing modifications to the GDL in the representation and computation of a junction tree (Section 5.2); and, explaining how it is possible to use it for

qualitative temporal reasoning (Section 5.3). In Section 6, we discuss our implemented instances of the GDL: The Fast-Hadamard Transform (Section 6.1) and Judea Pearl's Belief Propagation (Section 6.2). In Section 7, we conclude: Section 7.1 recapitulates this work; and, Section 7.2 discusses potential perspectives opened by our design and implementation. We added an appendix containing the proof of correctness of the parts of our design that depart from the Aji-McEliece design.

2 Bayesian Networks

In [1] and [2], most of the GDL instances studied are algorithms related to decision under uncertainty such as with Bayesian networks. Thus, let us first explain what a Bayesian network is, and review some related tools used for learning, inference, and decision making, under uncertainty.

2.1 Generalities

A Bayesian network (BN) is a graph-based model that enables making decisions under uncertainty [23]. What makes BN models even more interesting is that they may be learned from data (parameters *and* structure). They are among the most successful technology for data mining because they allow *adapting* decision making dynamically to changing environments. Some particular forms of dynamic BN models, such as Hidden Markov Models [30], are very successful in pattern recognition of sequential data such as time series forecasting.

Recall that the *conditional probability* of a random event A *given* that another random event B has occurred is defined as:

$$p(A \mid B) \stackrel{\text{def}}{=} \frac{p(A \cap B)}{p(B)}.$$

From this definition, Bayes's Law follows, which states that:[2]

$$p(A, B) = (A \mid B) \times p(B) = p(B \mid A) \times p(A).$$

This generalizes to n events ($n \geq 2$) as follows:

$$\begin{array}{rcl} p(A_{\pi_1}, \ldots, A_{\pi_n}) & = & p(A_{\pi_1} \mid A_{\pi_2}, \ldots, A_{\pi_n}) \\ & \times & p(A_{\pi_2} \mid A_{\pi_3}, \ldots, A_{\pi_n}) \\ & \vdots & \\ & \times & p(A_{\pi_n}) \end{array}$$

for any permutation $\{\pi_1, \ldots, \pi_n\}$ of the set $\{1, \ldots, n\}$.

[2] By convention, $P(A, B) \stackrel{\text{def}}{=} p(A \cap B)$.

Recall also that two events A and B are deemed *independent* (noted $A \perp B$) whenever the probability of them simultanously occurring is equal to the product of probabilities of each occurring; that is:

$$A \perp B \text{ iff } p(A, B) = p(A) \times p(B).$$

In other words:

$$A \perp B \text{ iff } p(A \mid B) = p(A) \text{ iff } p(B \mid A) = p(B).$$

In essence, a Bayesian network is a graph-theoretic encoding of causal conditional independence [26, 10, 16]. Figure 2 shows an example of a Bayesian net.[3] This graph

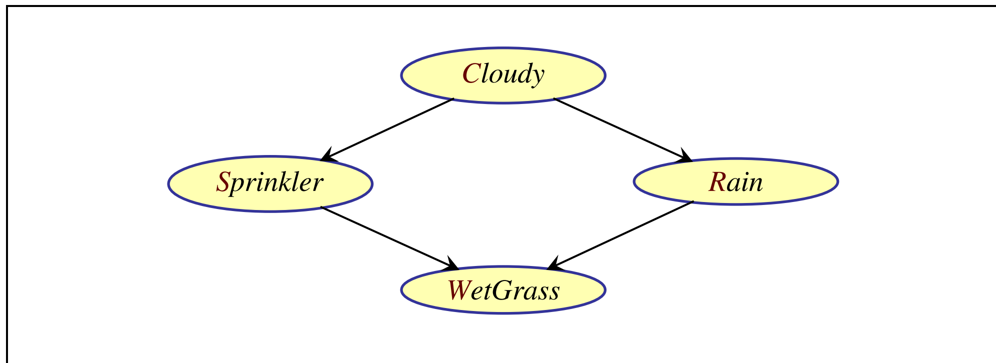


Figure 2: Example of Bayesian network

tells us that: (1) *R* and *S* are independent given *C*, and (2) *W* and *C* are independent given *R* and *S*. Figure 3 shows an example of causal conditional probability tables for the Bayesian network of Figure 2.

Conditional independence

The most important information leveraged by Bayesian reasoning is *conditional independence*. This is precisely where the "*graphical*" aspect of Bayesian networks comes into play as the graph structure of a BN encodes this information implicitly. For example, the graph structure shown in Figure 2 tells us that (1) *S* and *R* are independent given *C*, and (2) *W* and *C* are independent given *R* and *S*. Indeed, Bayes's rule gives the joint probability:

$$p(C, S, R, W) = p(C) \times p(S|C) \times p(R|C, S) \times p(W|R, C, S)$$

which is simplified by independence into:

$$p(C, S, R, W) = p(C) \times p(S|C) \times p(R|C) \times p(W|R, S).$$

[3] This "Wet Grass" example has become the standard all-replicated example in Bayesan tehcnology literature, pretty much as appending two linear lists for Functional Programming or Logic Programming. This is borrowed from [23].

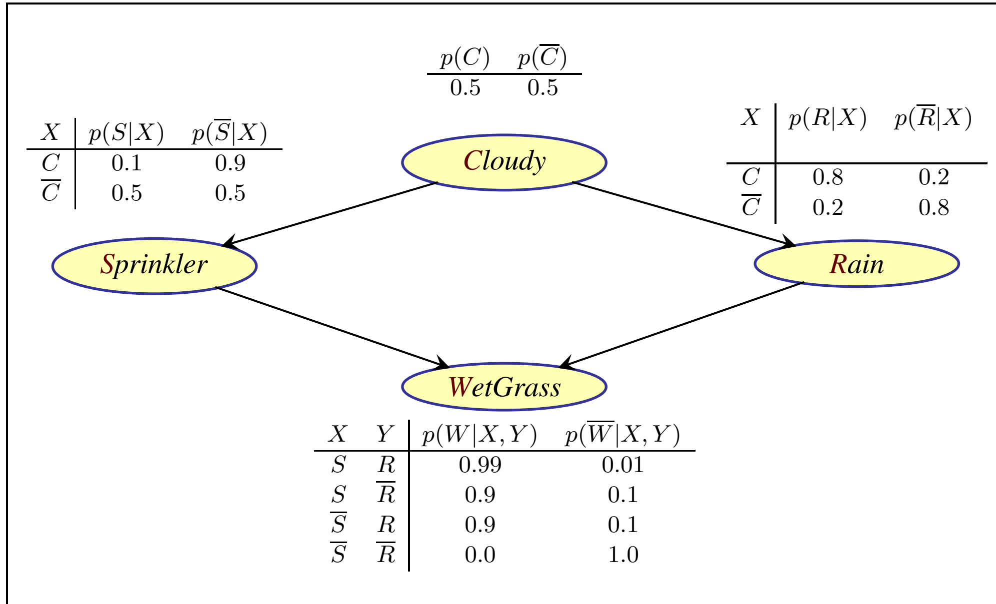


Figure 3: Example of Bayesian net's causal conditional probabilities

The probability *"marginalization"* is thus given by:

$$p(S|W) = \frac{p(S,W)}{p(W)} = \frac{\sum_{c,r} p(C=c,S,R=r,W)}{\sum_{c,s,r} p(C=c,S=s,R=r,W)}$$

$$p(S|W) = \frac{0.2781}{0.6471} = 0.4298$$

$$p(R|W) = \frac{p(R,W)}{p(W)} = \frac{\sum_{c,s} p(C=c,S=s,R,W)}{\sum_{c,s,r} p(C=c,S=s,R=r,W)}$$

$$p(R|W) = \frac{0.4581}{0.6471} = 0.7079.$$

Markov Blanket

As can be seen from our "wet grass" example, intuition may be easily fooled trying to determine *what* is independent of *what* given *what*—even in such a trivial causal graph! Fortunately, the wealth of formal research on the subject has made it possible to reduce this analysis to a very simple criterion. Indeed, conditional independence can be easily determined from the connectivity of a causal graph by computing each node's so-called *Markov blanket*. The Markov blanket of a node is defined as the set of nodes comprising the node's parents, its children, and its children's other parents.[4] That is, given a node X, its Markov blanket is defined as the set:

$$\partial_X \stackrel{\text{def}}{=} \textsc{Parents}(X) \cup \textsc{Children}(X) \cup \textsc{Parents}(\textsc{Children}(X)) \setminus \{X\}.$$

[4] http://en.wikipedia.org/wiki/Markov_blanket

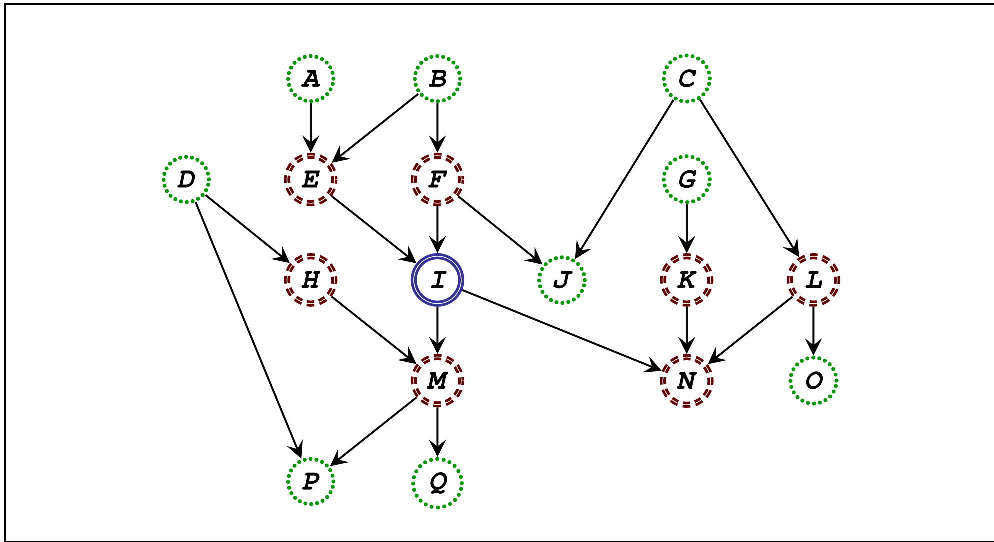


Figure 4: Example of a Markov blanket

For example, referring to the causal graph of the example in Figure 4, we obtain:

$$\text{PARENTS}(\boldsymbol{I}) = \{\boldsymbol{E}, \boldsymbol{F}\},$$

$$\text{CHILDREN}(\boldsymbol{I}) = \{\boldsymbol{M}, \boldsymbol{N}\},$$

and:

$$\text{PARENTS}(\text{CHILDREN}(\boldsymbol{I})) \setminus \{\boldsymbol{I}\} = \{\boldsymbol{H}, \boldsymbol{K}, \boldsymbol{L}\}.$$

Hence, the Markov blanket of the node $\boldsymbol{I}$ is:

$$\partial_{\boldsymbol{I}} = \{\boldsymbol{E}, \boldsymbol{F}, \boldsymbol{H}, \boldsymbol{K}, \boldsymbol{L}, \boldsymbol{M}, \boldsymbol{N}\}.$$

The Markov blanket splits each node X in the set $\mathcal{N}$ of nodes of a Bayesian network's causal graph partition $\mathcal{N}$ into three mutually exclusive components; namely, $\mathcal{N} = \partial_X \uplus \{X\} \uplus \partial_X^c$. The key result is that any node is independent of nodes outside its Markov blanket: $X \perp \partial_X^c \;\mid\; \partial_X$ [26, 16]. Using Bayes's rule, this allows the following simplification by conditional independence: $p(X \mid \partial_X, \partial_X^c) = p(X \mid \partial_X)$.

Belief revision

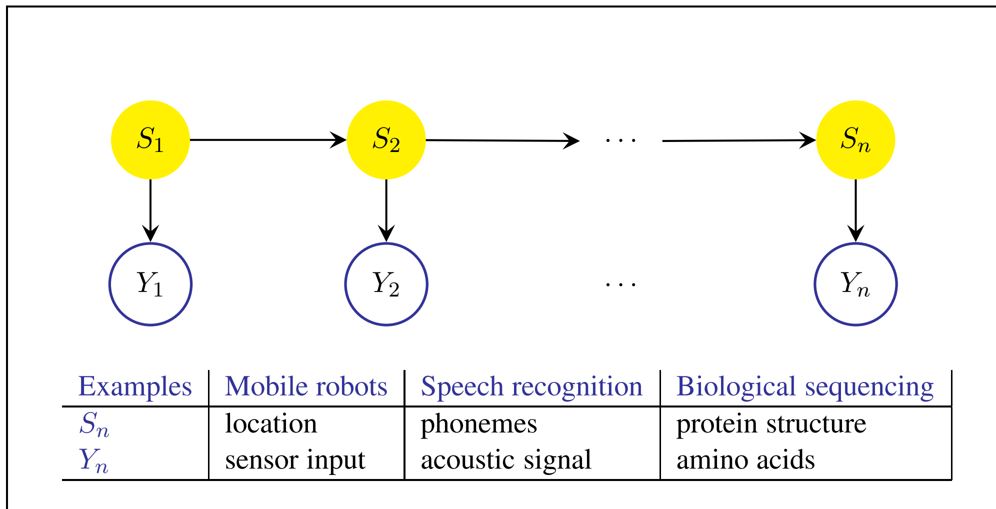
One of the most powerful capabilities offered by a Bayesian network is that it can adapt its knowledge according to accumulated evidence. This is known as *"explaining away"* since it is a form if *plausible reasoning* such that whenever several events are plausible causes of another one, say X, posterior evidence changes the likelihood of explanations for X. To see that with our example, let us suppose that it is observed that

(W) the grass is wet and that (R) it is raining. Then, this indicates that the posterior likelihood that (S) the sprinkler is on goes down as follows:

$$p(S \mid W, R) = \frac{p(S, W, R)}{p(W, R)} = \frac{\sum_c p(C = c, S, W, R)}{\sum_{c,s} p(C = c, S = s, W, R)} = 0.1945.$$

Causal learning

Yet another benefit of Bayesian networks is that they can be learned from data, thus circumventing the “expert belief assessment” problem [6]. Indeed, recent research in Data Mining has made great progress for learning Bayesian network (parameters *and* structure) from data [24].[5,6]



Examples	Mobile robots	Speech recognition	Biological sequencing
S_n	location	phonemes	protein structure
Y_n	sensor input	acoustic signal	amino acids

Figure 5: Example of an HMM’s hidden states and their observations

Dynamic Bayesian Networks

Bayesian networks also appear in particular specific instances (*e.g.*, Hidden Markov Models, Linear Dynamic Systems) that are very successful for pattern recognition of sequential data (*e.g.*, speech recognition [35], time series data [19]). Figure 5 shows an example of a Hidden Markov Model (HMM), a particular instance of a dynamic Bayesian network where the S_i’s are time-indexed “hidden” (*i.e.*, unobservable) states of a Markov process, and each Y_i is an observable random function of the corresponding hidden state S_i.

In addition, HMMs can be “trained” on data in the manner of neural networks to enable forecasting. They have been used extensively and applied with great success in

[5] http://www.cs.cmu.edu/~awm/10701/slides/Param_Struct_Learning05v1.pdf
[6] http://www.autonlab.org/tutorials/

diverse fields such as signal decoding [20], speech recognition [35], geological exploration [28], stock trading [25], and genome analysis.[7]

During the past decade, this an important set of algorithms have been contributed; *e.g.*, Judea's Pearl Belief Propagation [27], the Expectation-Maximization algorithm [21], Viterbi's algorithm [34] and many others. All these can be cast as specific instances of the GDL algorithm.

The Expectation-Maximization algorithm was introduced in 1977 in a paper by Dempster, Laird, and Rubin [11]. This algorithm is used to find the maximum-likelihood parameters of a statistical model when the equation of the model itself cannot be solved directly. This algorithm is often used in computational biology applications since it allows drawing conclusions with incomplete sets of data. The Expectation-Maximization algorithm is indeed quite useful in domains where there is no guarantee that the data collection will produce complete sets of data. The algorithm is quite complex but it is possible to expose a commutative semiring underpinning the operations performed by the algorithm. Therefore, it becomes possible to use the GDL to solve this kind of problem.

Viterbi's algorithm is a Dynamic-Programming algorithm for finding the most likely sequence of hidden states in a Hidden Markov Model. It is broadly used for decoding convolutional codes—such as, for example, using Viterbi's Algorithm in order to decode bit-stream [34]. These algorithms are well-known in probabilistic deduction. They are widely used in causal learning [24], Sequential Data Analyisis [30, 35, 19, 25], belief revision [16, 27, 26], probabilistic-logic programming [9], and many more application areas [28], *etc.*, ...

Having a single *abstract* library capable of implementing all these algorithms offers a truly versatile parametric application generator to reproduce quickly specific efficient computation methods over commutative semirings. It can also provide an effective experimental toolset for a large set of AI applications, including where reasoning under uncertainty must be handled.

2.2 The GDL in the literature

Several papers [18, 12] have been published that use, or refer to, the GDL algorithm.

In [18], the GDL is extended to exact solutions. This paper presents an algorithm called the "Sum-Product Algorithm" that is similar to the GDL. This algorithm is also a generic message-passing method but uses factor graphs instead of junction trees.[8] However, during the message passing algorithm, some operations can become intractable. A solution to handle such cases is provided that allows attaining exact results. Interestingly, we will show that this algorithm can be expressed as a slight modification of the GDL as originally proposed by Aji and McEliece. Indeed, we also propose improvements of the GDL and apply it to new use cases.[9]

[7] http://genomics10.bu.edu/bioinformatics/kasif/bayes-net.html
[8] See Section 3.2.
[9] See Section 5.

In [12], the focus is on the statistical and information-theoretic aspects of Hidden-Markov Processes (HMPs). It is shown how to generalize HMPs under some conditions. Consequently, new algorithms were proposed for universal decoding of HMPs. The GDL is relevant to HMPs for two reasons: (1) it is a generic algorithm; (2) it can be made to model a large class of HMPs.

These small examples demonstrate that references to the GDL can be found in the literature. However, none reports the use of a generic GDL package to be used and reused for diverse purposes. In the future, it is thus hoped that, thanks to generic libraries like the `Cedar.Gdl`, the GDL could be used more systematically as the basis of many new instantiations based only on the properties of an underlying commutative semiring structure.

To recapitulate, the GDL algorithm can be applied to many use cases. But as far as we know, there exists no implemented generic library for it that is able to deal with many different structures and problems using the same abstract methods. The design and implementation of the `Cedar.Gdl` abstract library is an effective means for experimenting with specific commutative semiring algebraic structures corresponding to specific use cases in order to optimize the evaluation of expressions. Depending on the specific semiring operations of a given use case, evaluating such expressions corresponds to carrying out some kind of reasoning such as decision making under uncertainty, predictive modeling in sequential random processes, *etc.*, ... Thus, proposing a tool such as the `Cedar.Gdl` library opens the way to exploring new use cases with minimal investment while keeping the number of core operations performed minimal. This investment amounts to the sheer specifying two concrete operations of a communtative semiring and the concrete data structures they act upon, all of which instantiate the abstract API provided by the `Cedar.Gdl` library.

3 The Generalized Distributive Law Algorithm

This section is an illustrated, step-by-step, explanation of the Generalized Distributive Law. After a brief definition of the GDL, we list the requirement for using the GDL's algorithm and how to attain them. Then, we explain the algorithm itself. In the following subsections, we use the use-case example of the Fast-Hadamard Transform for illustration purposes.

3.1 The GDL

In essence, the GDL computes "sums of products"; *e.g.*, expressions of the form:

$$\alpha(x, z) = \sum_{x_k \in S_k}^{i \leq k \leq m} \prod_{y_l \in T_l}^{i \leq l \leq n} \phi(x_1, \ldots, x_m, y_1, \ldots, y_n) \tag{3}$$

where $\prod$ and $\sum$ correspond to additive and multiplicative laws of a commutative semiring.

The GDL uses a finite set $\{x_1, \ldots, x_n\}$ of *variables*, where each variable x_i takes values in a finite discrete domain D_i, for $i \in \{1, \ldots, n\}$. We define the *local indices* as subsets of the set of the first n natural numbers; namely, $\mathcal{I} \stackrel{\text{def}}{=} \{I_1, \ldots, I_m\}$ such that $I_k \subseteq \{1, \ldots, n\}$, for $k = 1, \ldots, m$.

Given a local index, we define a *local domain* as follows.

DEFINITION 2 (LOCAL DOMAIN) *Let $I = \{I_1, \ldots, I_r\}$ be a local index; we define the* local domain*:*

$$D_I \stackrel{\text{def}}{=} D_{I_1} \times \ldots \times D_{I_r}$$

such that the I-indexed r-tuple of variables $x_I \stackrel{\text{def}}{=} \langle x_{I_1}, \ldots, x_{I_r} \rangle \in D_I$.

With this, we can now express the GDL algorithm in terms of two sets of abstract functions. One set is taken as parameters—the so-called *local kernels*.

DEFINITION 3 (LOCAL KERNEL) *A local kernel is a function $\alpha_i : D_{I_i} \to \mathcal{K}$, where $\mathcal{K}$ is a commutative semiring.*

Another set of functions are defined in terms of the local kernels defined as given by Definition 3—the so-called *global kernels*. Given a local index $I = \{I_1, \ldots, I_r\}$:

DEFINITION 4 (GLOBAL KERNEL) *A* global kernel *is a function $\beta : D_1 \times \ldots \times D_n \to \mathcal{K}$ defined as the product of local kernels:*

$$\beta(x_1, \ldots, x_n) \stackrel{\text{def}}{=} \prod_{i=1}^{m} \alpha_i(x_{I_i}).$$

Still in the context of a given a local index $I = \{I_1, \ldots, I_r\}$, we define the notion of *marginalization* of a global kernel as follows:

DEFINITION 5 (MARGINALIZATION FUNCTION) *The* I_i-marginalization *function is the sum over complemented partial sets of indices I_i of the global kernel function values (also called* "(i-th) local objective function"*):*

$$\beta_i : D_{I_i} \to \mathcal{K} \ : \ \beta_i(x_{I_i}) \stackrel{\text{def}}{=} \sum_{x_{I_i^c} \in \mathcal{D}_{I_i^c}} \beta(x_1, \ldots, x_i, \ldots, x_n).$$

With this setup, the GDL may then be formulated as a variable-elimination algorithm for computing local objective functions given local kernels on local domains.[10]

[10] The expression *bucket elimination* has been used to denote the basic junction tree technique [9, 10]. This is because it maximizes the number of eliminated variables per "pivot step."

3.2 Junction trees

The GDL message-passing algorithm can only be performed if the elements of S can be organised into a junction tree [13]. Therefore, the first step is to obtain such a tree for the desired instantiation.

DEFINITION 6 (JUNCTION TREE) *A tree T is a junction tree if: $\forall S_i S_j \in T$, $S_i \cap S_j$ is a subset of every vertex on the path from S_i to S_j, and only one path exists between S_i and S_j.*

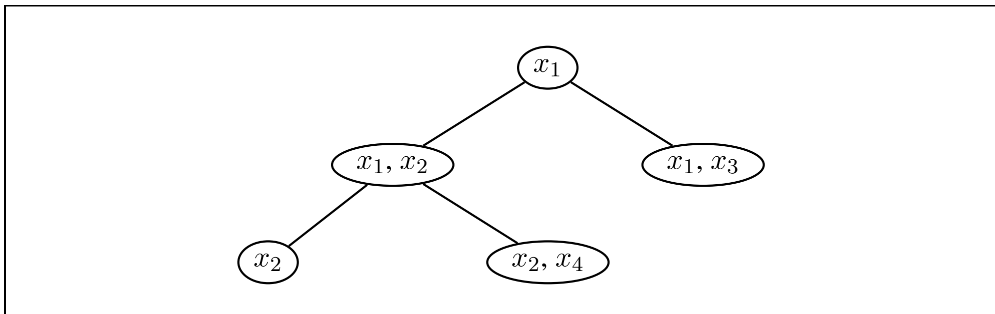


Figure 6: A junction tree

Let us take an example that will lead us to the Fast-Hadamard Transform (FHT). The FHT is used in many domains as Images Processing [15]. Let $x_1, x_2, x_3, y_1, y_2, y_3$ be variables taking value in the binary set $\{0, 1\}$, and $f(y_1, y_2, y_3)$ a real-valued function. With this, we can define the problem displayed in Table 1.

Local Domain	Local Kernel
$\{y_1, y_2, y_3\}$	$f(y_1, y_2, y_3)$
$\{x_1, y_1\}$	$(-1)^{x_1 y_1}$
$\{x_2, y_2\}$	$(-1)^{x_2 y_2}$
$\{x_3, y_3\}$	$(-1)^{x_3 y_3}$
$\{x_1, x_2, x_3\}$	1

Table 1: GDL formulation for the Fast-Hadamard Transform

The first step, toward building a junction tree is the creation of a local domain tree using the list of local domains and kernels.

DEFINITION 7 (LOCAL DOMAIN TREE) *A local domain tree is a tree in which every local domain is the label of a vertex, and a link exists between two vertices v_i and v_j if share at least one variable. Such a link is weighted by the number of variables in common.*

In the `Cedar.Gdl` Java library, once the n local domains and local kernels are defined, a quick algorithm will put every local domain in a node. Then, it looks for common variables between them and creates a link if there are any common variables. To every link is associated a weight equal to the number of identical variables in the pair of vertices that it connects. For our example, this leads to the graph shown in Figure 7.

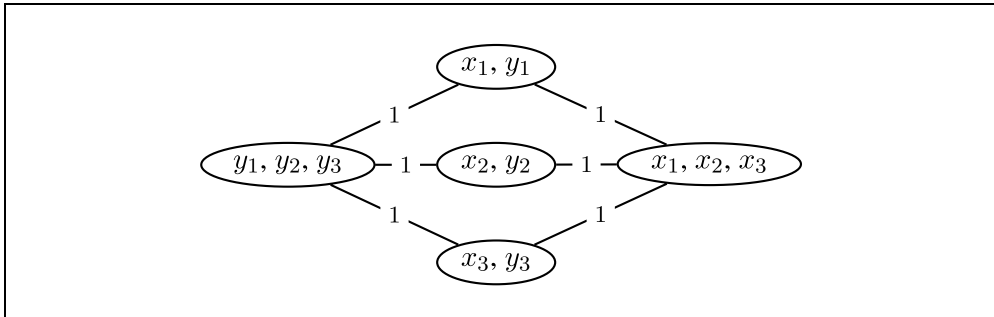


Figure 7: The local domain graph associated to Table 1.

The local-domain graph is used as the basis for building a maximal-weight spanning tree. Several such algorithms exist; *e.g.*, as Prim's Algorithm [14]. All have their strengths and weaknesses. However, we decided to opt for Kruskal's Elimination Algorithm [17]. This is for essentially three reasons. First, this algorithm fits our internal data representation. Second, it works well. Finally, it simplifies the implementation of heuristics, adapted to the context of the GDL.

Indeed, a maximal-weight spanning tree is in general not unique. While a specific spanning tree behaves well on certain problems, it may also behave so wo well, or even quite badly, on others. In the context of the GDL, the goal is to minimize the calculation during the message-passing algorithm. Therefore, the heuristics must be adequate with this goal. A link on a variable v_i which takes values in a set A_i is more interesting to keep than a link on a variable v_j, if $|A_j| > |A_i|$. With the heuristics described in the Generalized Distributive Law, Kruskal's Algorithm performed on Figure 7, leads to the Maximal-Weight Spanning tree in Figure 8.

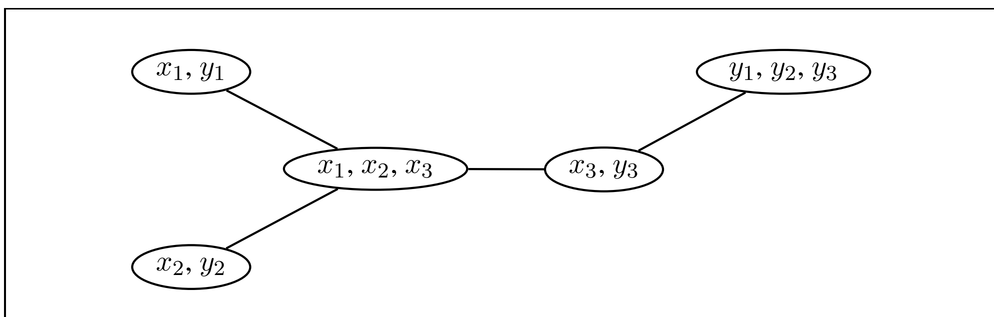


Figure 8: A maximal-weight spanning tree associated to Figure 7

At this point, two situations are possible. Either the maximal-weight spanning tree is already a junction tree, in which case there is nothing further to do. Or it is not a junction tree, in which case another algorithm must be used to find a junction tree. To know whether or not a maximal-weight spanning tree is indeed a junction tree, a simple test may be performed. Before we describe it, let us first define some notations.

The maximal-weight spanning tree contains M vertices, $v_1, \ldots, v_M$ (one per local domains). Every vertex v_i contains a set of variables S_i (of cardinality $|S_i|$). Using this notation we can define:

$$w^* = \sum_{i=1}^{M} |S_i| - n \tag{4}$$

In words, w^* is the sum of the cardinality of all the local domains minus the number of different variables. We also define $w_{\max}$ as the weight of the maximal-weight spanning tree (*i.e.*, the sum of the weight of all the edges). Once this is done, a simple test checks whether this is junction tree. If $w^* = w_{\max}$, then any maximal-weight spanning tree is a junction tree, and the message-passing algorithm can be performed. However, if $w^* > w_{\max}$, more work is needed to find a junction tree.

For the Fast-Hadamard Transform corresponding to Figure 8, $W^* = 3+2+2+2+3-6 = 6$. However, $w_{\max} = 4$ because all the edges of the maximal-weight spanning tree have a weight equal to 1. Thus, the maximal-weight spanning tree of Figure 8 is not a junction tree. When this happens, the GDL uses another method to find a junction tree based on the construction of a *moral graph*. This is explained next.

DEFINITION 8 (MORAL GRAPH) *A moral graph is a graph with n vertices, one for every variable $x_1, \ldots, x_n$, with an edge between two vertices if and only if there is a local domain which contains both of them.*

In the `Cedar.Gdl` Java library, an algorithm similar to the creation of the local domain graph is performed. For the FHT example of Table 1, it leads to the tree shown in Figure 9.

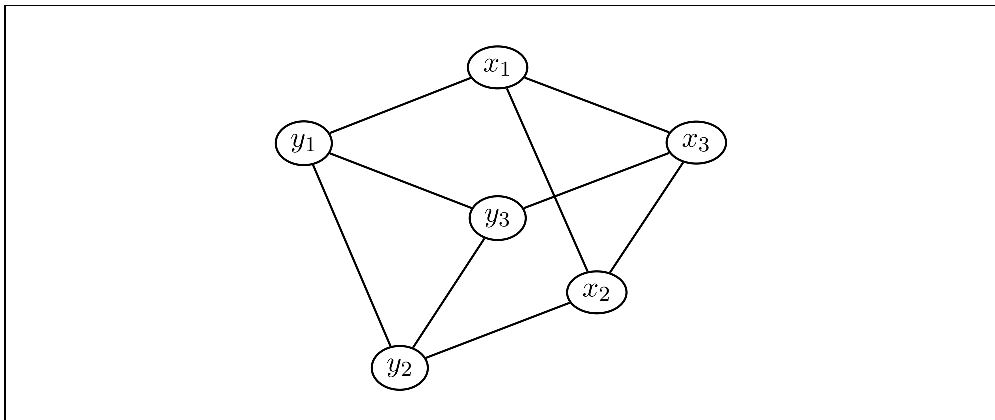


Figure 9: The moral graph of the variables in Table 1

Once the moral graph is created it is needed to triangulate it (*i.e.*, make it chordal).

DEFINITION 9 (CHORD) *A chord is an edge between two non-consecutive vertices in a cycle.*

DEFINITION 10 (CHORDAL GRAPH) *A graph is chordal if and only if every cycle of length greater than three has a chord.*

Therefore, we need to add edges until it satisfies Definition 10. The triangulation of a moral graph is an *NP*-Hard problem. Many algorithms are known, but none of them is optimal in all cases. In [5], the authors discuss an approximation for a 3-way vertex cut. It is of interest because its purpose is to reduce the sizes of the maximal cliques.

DEFINITION 11 (CLIQUE) *A clique consists of a subset of vertices such that every two vertices in the subset are connected by an edge.*

For the GDL algorithm, reducing the size of the maximal clique is of the utmost importance. From this, will depend the size of the domains on which will be performed the calculations, and thus the number of operations. However, the GDL is applied to diverse cases such as the FHT, and Judea Pearl's Belief Propagation. Thus, we use Tarjan Elimination algorithm because it is more convenient for specifying heuristics in the computation of the moral graph triangulation (see Algorithm 1). This flexibility allows to specify our specific triangulation heuristics, with the purpose of reducing the size of the maximal cliques.

Result: A chordal graph
Input: an undirected graph $G = (V, E)$;
while *Not all the nodes are dead* **do**
  Pick any living node $X \in V$;
  Connect all the neighbors of V (*i.e.*, create a clique with V and all its neighbors);
  Mark X as dead;
end

Algorithm 1: Tarjan's elimination algorithm

In this algorithm, the order in which the nodes are picked is important. To deal with this issue efficiently, we propose the heuristic described in Algorithm 2.

This heuristic makes the (locally) optimal node choice at every stage of the triangulation. However, it can also be expensive. Nevertheless, it significantly reduces the calculation during the message-passing algorithm. However, sometimes it can be useful to spend less time on the triangulation of the moral graph. In such a case, a less costly heuristic can be used.

The implemented Tarjan's Elimination Algorithm with the heuristic defined in Algorithm 2, yields the graph in Figure 10. Once we have a triangulated moral graph, we

while *Not all the nodes have been picked* **do**
  For every node V, count the number of edges needed for creating a clique with its neighbors;
  Create an ordered list of nodes using these numbers as scores;
  if *at least one node has a score of 0* **then**
    Pick all the nodes with a score of 0 and perform the Tarjan's Elimination algorithm on them;
  else
    Pick the node with the lowest score and perform the Tarjan's Elimination algorithm on it;
  end
end

Algorithm 2: Heuristic for node choice

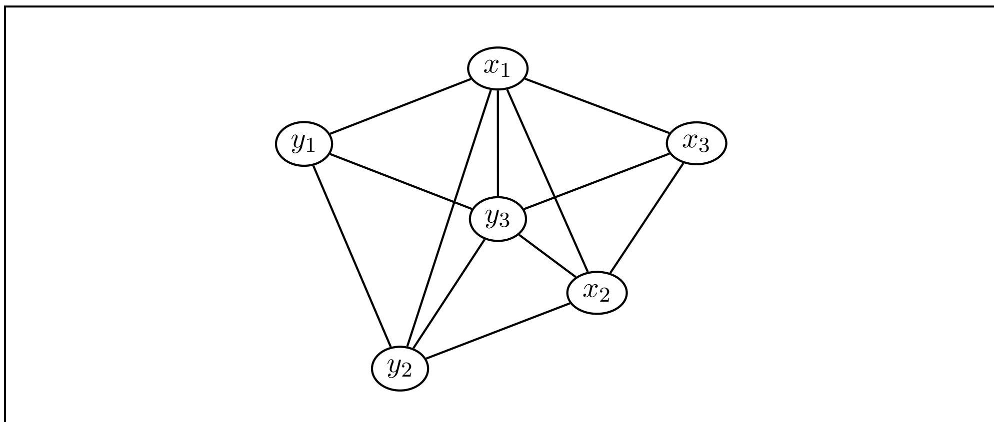


Figure 10: The triangulated moral graph corresponding to Figure 9

create a tree using the cliques of the chordal moral graph as vertex labels. In the Fast-Hadamard Transform example, the triangulated moral graph can be decomposed into three cliques: $\{x_1, x_2, x_3, y_3\}$, $\{x_1, x_2, y_2, y_3\}$ and $\{y_1, y_2, y_3, x_1\}$. These cliques can be ordered into the tree described in Figure 11.

Now, every original local domain is a subset of at least one of the created cliques. Thus, a local domain can be linked to one of them. The result is a graph with the created vertices from the cliques, as core and the original local domains as leaves (see Figure 12).

Then, the vertices from the cliques must be associated with one of the local domain linked to them. In some cases some created vertices cannot be associated with a local domain. We discuss in detail this case later, but let us suppose for now that such an association is possible. However, in general, the resulting tree is not unique. There is no specific algorithm given in [2] nor in [1] for this stage.

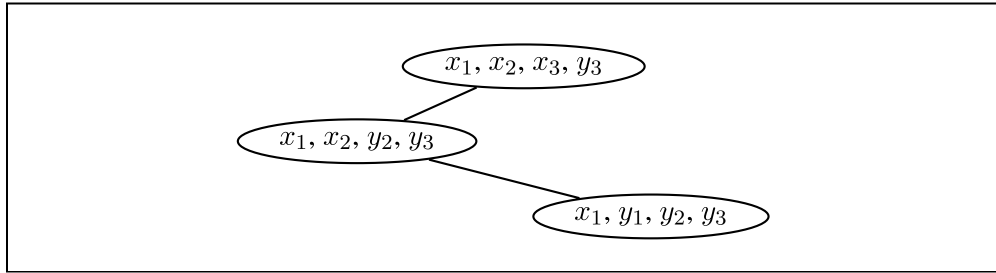


Figure 11: A tree for the cliques in the triangulated moral graph of Figure 10

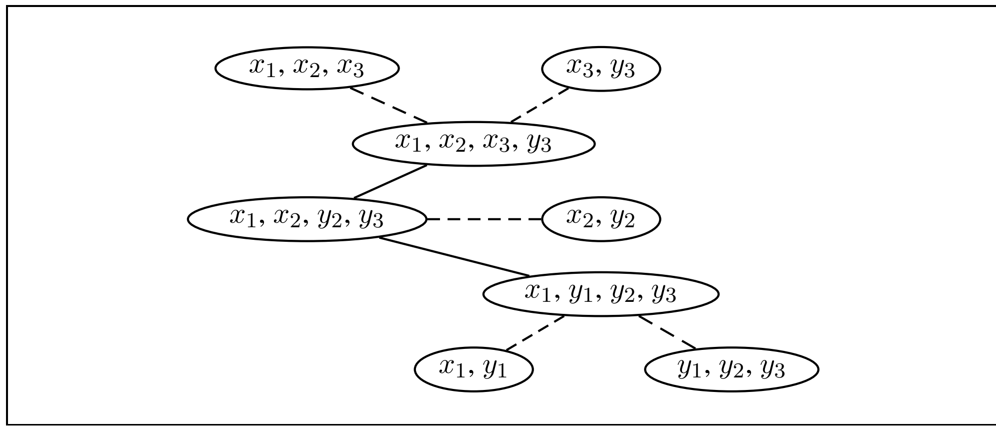


Figure 12: The local domains corresponding to Figure 11

To address this issue, we developed our own algorithm. Its method is similar to a scheduling association algorithm. The goal is to allow as many created cliques as possible (ideally all), to get associated with a local domain. When a vertex containing a local domain is merge into a vertex from a clique, the local domain becomes extended by the additional variables in the vertex from the clique. The result is a junction tree with each vertex associated to a local kernel and a local domain (see Figure 13).

This section has addressed issues concerning the existence and creation of a junction tree. Obtaining a junction tree is, at an implementation level, the most important part of this abstract library. Now that we have a junction tree, let us have a closer look at the message-passing algorithm.

3.3 The message-passing algorithm

Given a well-defined junction tree, the GDL works as a message-passing algorithm [20]. It is summarized in Figure 14.

The GDL iteratively updates a table $\mu_{i,j} : D_{I_i \cap I_j} \rightarrow \mathcal{K}$ for each node I_i, for all $I_j \in \boldsymbol{N}(I_i)$, where $\boldsymbol{N}(I)$ is the set of all neighbors of I. The $\mu_{i,j}$ table contains "messages" passed from from node I_i to node I_j—each node "sends a message" to a neighbor when it has received one from all its other neighbors *first "upward" then "downward."*

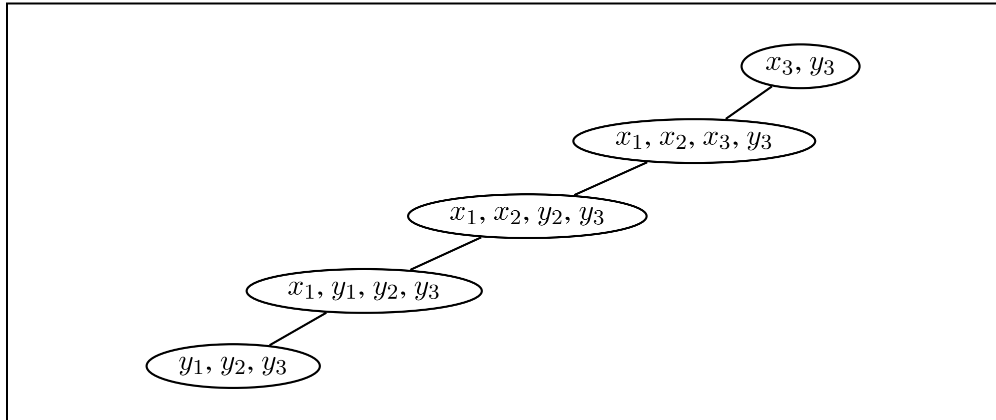


Figure 13: A junction tree for the local domains in Figure 12

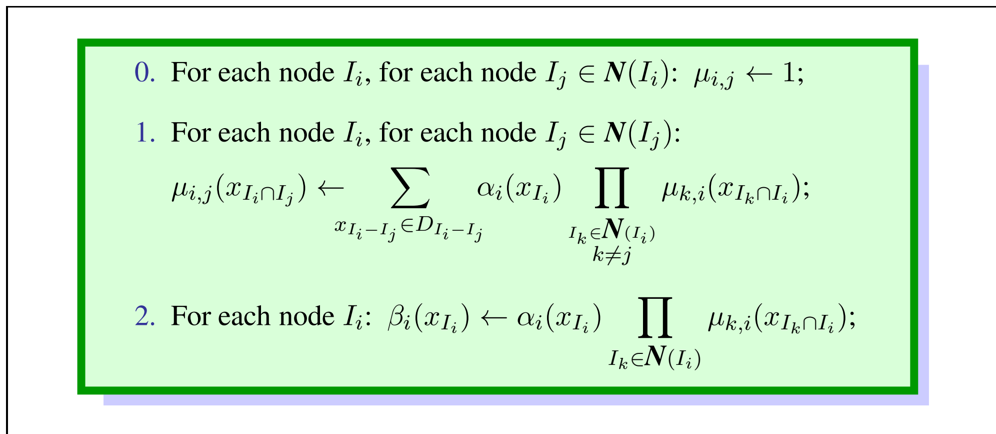
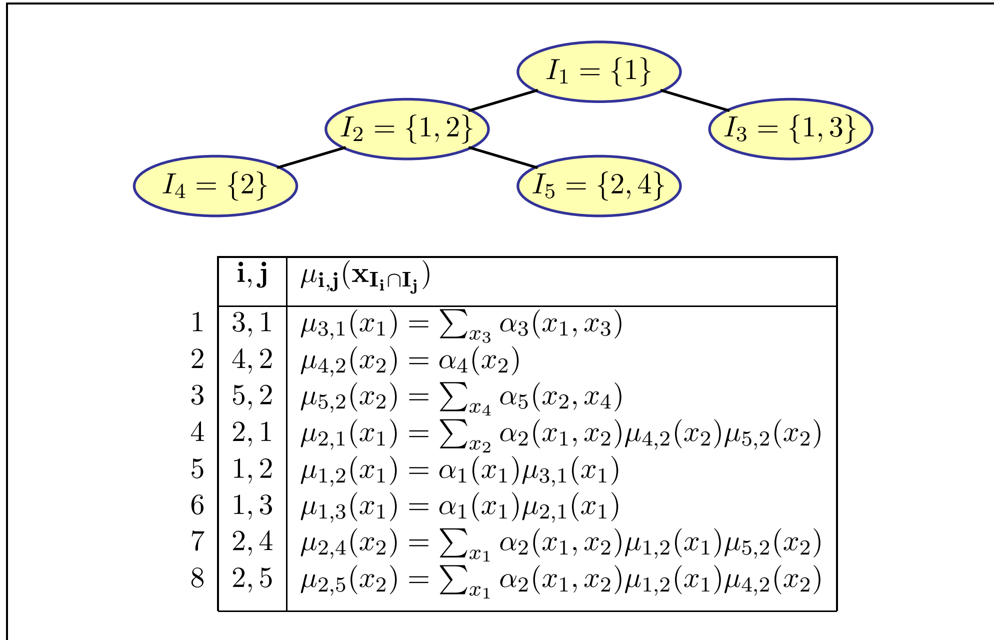


Figure 14: The GDL message-passing algorithm

The effect of the GDL message-passing algorithm of Figure 14 is illustrated in Figure 15 for the junction tree shown.

The algorithm exchanges messages between the vertices of the junction tree until enough messages have been send. When this situation is met a vertex is able to calculate its state that correspond to the global kernel for the corresponding local domain.

There are two cases: the single-vertex problem and the all-vertex problem. In the first one, all the messages are sent toward one specified vertex. The purpose is to ensure that the vertex receives a message from all the other vertices of the graph by the end. In the all-vertex problem, messages are exchanged in all the directions until all the vertices of the junction tree had received all the messages. However, in both cases the fundamental principle of the message-passing algorithm stays the same. The only information that changes is the number and destination of the messages. When a vertex



	$\mathbf{i,j}$	$\mu_{\mathbf{i,j}}(\mathbf{x}_{\mathbf{I_i} \cap \mathbf{I_j}})$
1	$3,1$	$\mu_{3,1}(x_1) = \sum_{x_3} \alpha_3(x_1,x_3)$
2	$4,2$	$\mu_{4,2}(x_2) = \alpha_4(x_2)$
3	$5,2$	$\mu_{5,2}(x_2) = \sum_{x_4} \alpha_5(x_2,x_4)$
4	$2,1$	$\mu_{2,1}(x_1) = \sum_{x_2} \alpha_2(x_1,x_2)\mu_{4,2}(x_2)\mu_{5,2}(x_2)$
5	$1,2$	$\mu_{1,2}(x_1) = \alpha_1(x_1)\mu_{3,1}(x_1)$
6	$1,3$	$\mu_{1,3}(x_1) = \alpha_1(x_1)\mu_{2,1}(x_1)$
7	$2,4$	$\mu_{2,4}(x_2) = \sum_{x_1} \alpha_2(x_1,x_2)\mu_{1,2}(x_1)\mu_{5,2}(x_2)$
8	$2,5$	$\mu_{2,5}(x_2) = \sum_{x_1} \alpha_2(x_1,x_2)\mu_{1,2}(x_1)\mu_{4,2}(x_2)$

Figure 15: Example of the effect of the GDL message-passing algorithm

i is required to send a message to a vertex j the following rule is used:

$$\mu_{i,j}(x_{S_i} \cap x_{S_j}) = \sum_{x_{S_i} \setminus x_{S_j} \in A_{S_i} \setminus A_{S_j}} \alpha_i(x_{S_i}) \prod_{\substack{v_k \ \textbf{adj} \ v_i \\ k \neq j}} \mu_{k,i}(x_{S_k} \cap x_{S_i}) \tag{5}$$

Note that Equation (5) uses the notation u **adj** v ("*adjacent*") for two vertices u and v to mean that there exists an edge between u and v. Using it is equivalent to the conditions on indices used in Figure 14. This is because the formula computes the product of messages received from adjacent vertices except for the one to which the message is destined. Hence, the condition v_i **adj** v_k and $k \neq j$ whenever i is the current vertex and j is the recipient of the message.

When i is required to send a message to j, it calculates the product of its local kernel with all the messages it haves previously received. The rule is that a vertex only send a message when it haves received a message from all its neighbors. Therefore, the messages start at the leaves and progress inward in the tree. For the single-vertex problem, the junction tree found in the previous section must be oriented. The first node must be the vertex for which we want the result. Transform a non-directed junction tree into an oriented junction tree is not difficult. In the library `Cedar.Gdl` a simple recursive method do the transformation. For the single-vertex message passing, we use the recursive algorithm described in Algorithm 3.

Because this is for the single-vertex problem, the messages are sent in one direction only. Furthermore, the junction tree is directed. This allow this basic algorithm to perform quite well. However, in most cases, it is the all-vertex problem that is used. This is because its complexity is only four times that of the single-vertex problem. The

The current vertex is named v_c, its father v_f, and the list of its children V_i;
Start at the vertex for which the algorithm must be performed;
Enumerate the number of message n_m received, and v_c;
if v_f *received at least* $|V_i| - 1$ *messages* **then**
  Send a message to v_f using Definition 5;
else
  Perform Algorithm 3 recrusively on all the children in V_i;
end

Algorithm 3: Heuristic for the "*single-vertex*" message-passing algorithm

rules are the same but this time the tree is not directed. This is because the messages are exchanged in both directions, up and down the tree. Therefore, the algorithm described in Algorithm 3 can no longer be used. For the all-vertex problem, we use Algorithm 4.

In both the single-vertex and the all-vertex problem, when a vertex has received all its expected messages, it calculates its state using the formula:

$$\sigma_i(x_{S_i}) = \alpha_i(x_{S_i}) \prod_{v_k \textbf{ adj } v_i} \mu_{k,i}(x_{S_k} \cap x_{S_i}) \tag{6}$$

This is simply the multiplication of the local kernel by the product of all the previously received messages. This state of the vertex is the objective function at the local domain associated to the current vertex. At this point the GDL's algorithm terminate. The `Cedar.Gdl` library has obtained a simplification of the GDL standard on the instantiated semiring.

4 The `Cedar.Gdl` Library Implementation

The following sections give an overview of the implementation of the `Cedar.Gdl` library in Java. These sections are not a tutorial or a user manual. The focus here is on the way the classes goes together and how they interact between each over.

4.1 The abstract core

This section introduces the abstract classes and explains how they allow to express instances of GDL problems.

In mathematics, every computation can be reduced to a succession of operations between two atoms. For example:

$$\sum_{i=1}^{3} x_i = (x_1 + x_2) + x_3 \tag{7}$$

Let v_c denote the current vertex, and V_n the list of v_c's neighbors ;
Create a list with all the vertices of the junction tree and call it L_v;
For every vertex in the list, create a 2-row table, each row of size the number of the vertex's neighbors. This table is used to determine for any given vertex which vertices have already sent a message to it, and which it has already sent a message;
For all the vertices in the list, let N_m be the number of messages received.;
while *All the message have not been sent* **do**
  index = 0;
  while *index* $< \ |L_v|$ **do**
    v_c = the vertex at the position index in L_v;
    if v_c *received at least* $|V_n| - 1$ *messages* **then**
      Search in the table to which vertices v_d, it must send a message (v_c must have received messages from all the other vertices of V_n, and not already sent a message to v_d);
      Send a message to v_d and update the tables of v_d and v_c;
    else
      Go to the next vertex in the list;
    end
    index++;
  end
end

Algorithm 4: Heuristic for the "*all-vertex*" message-passing algorithm

Equation (7) can be seen as, an operation $+$ between the atoms x_1 and x_2, and an operation $+$ between the result of the previous operation and x_3. We adapt the idea into an abstract class, `Abstract_Atom`. This class is used in the library to represent a mathematical element or an operation. `Abstract_Atom` is the most basic class of the library `Cedar.Gdl`. Its purpose is to required the others class of the library to implement specific methods. Having common methods to all the classes allow to call them without knowing the real type of the class. It is this abstraction that allows to use generic semiring. Because it is abstract, anyone who would like to implement a class extending `Abstract_Atom` would have to implement the `toString()`, `calculate()` and `returnVariableUsed()` methods. The method `toString()` need to be instantiated in order to have text feedback of the result of the GDL. This method must return a `String` that describes the class (*e.g.*, for a function,"f" of two variables the `toString()` method could return "f(x,y)").

As displayed in Figure 16, two classes extend `Abstract_Atom`: `Abstract_Function` and `Abstract_Operation`. The class `Abstract_Operation` must be extended by every class used in the `Cedar.Gdl` library and representing an operation. The purpose of this class is to hide the class `Abstract_Atom` from the user. Because it extends `Abstract_Atom` and does not implement the three functions mentioned earlier, any class extending `Abstract_Operation` must implement these methods.

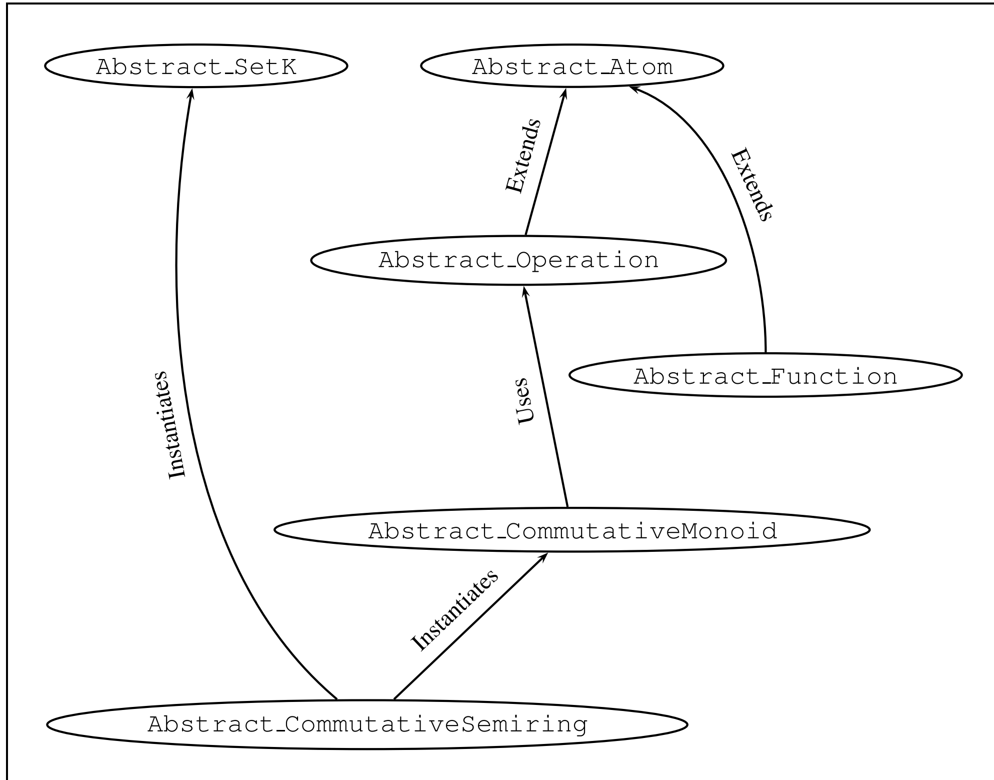


Figure 16: Existing relations between the abstract classes

The second class `Abstract_Function` represent a mathematical function. This class requires instantiating a method `functionImplementation()` instead of the inherited method from `Abstract_Atom`, `calculate()`. This method represents the core of the function.

The three classes `Abstract_Atom`, `Abstract_Operation` and `Abstract_Function`, are here to handle the calculations in the library. The three overs classes that we can see in Figure 16 are used to represent the required framework of the GDL (*i.e.*, a commutative semiring). A commutative semiring is composed of two commutative monoids and one set. Therefore, the implementation in the library reproduces this organization. The class `Abstract_Monoid` represents a commutative monoid and requires instantiating three methods: `calculate()`, `calculateOnSet()` and `operation()`. These three methods are to handle the three possible cases:

- The method `operation()` is to calculate the result of a binary operation (*e.g.*, for a "+" monoid, this method should handle the addition).
- The second method is `calculate()` and handles the operation for subclasses of `Abstract_Atom`. In the library most calculation is not actually performed until the end of the algorithm. Instead of computing results during the message-passing algorithm described in Section 3.3, the library constructs objects representing unevaluated mathematical expressions. This will be discussed at greater length in Section 4.2.

- The method calculateOnSet() handles the sum over sets for expressions of type Abstract_Atom.

All the classes presented in this section represent the core of the library Cedar.Gdl. They all need to be instantiated corresponding to your own problem in order to perform computation with the library.

4.2 Classes implementing the GDL

This section presents some of the classes that are used in addition to the abstract classes presented in Section 4.1. These are used to construct a junction tree and compute the message-passing algorithm using the implemented abstract core.

Once a GDL problem is instantiated using the classes LocalDomain and LocalKernel, the library Cedar.Gdl will start the algorithm described in Section 3.2. All the steps described are implemented by the method of the class GraphManipulation. First, the library will construct the local domain graph using the classes Vertex, and Edge. These two classes represent respectively a vertex and an edge. This representation allow to spare calculations when the edges have to be deleted or stored in memory as for the Kruskal's Algorithm [17]. Then, if the library have to compute the moral graph like in Figure 8, a specific class MoralGraphCell is used.

We use a specific class for the moral graph construction. The triangulation of the moral graph is performed by the method triangulateMoralGraph that implements Tarjan's Elimination Algorithm (see Algorithm 1). At this point the library Cedar.Gdl possesses the desired junction tree. The last step before the beginning of the message-passing algorithm is to modify the intern representation of the junction tree. So, a class JunctionTreeCell is used. This class helps implementing the message-passing algorithm. Much information is required during the message-passing phase and, more importantly, the children and parents of a cell have to be easily accessible. This helps the message-passing algorithm to achieve good performances.

Now that a junction tree is created, the message-passing algorithm can start. A class messagePassing contains all the methods related to the message-passing algorithm. To send a message from a vertex i to a vertex j the algorithm described in Algorithm 5 is used.

When this method is used several times and messages exchanged, mathematical expressions are constructed as shown in Figure 17. Creating such expressions, which are Abstract_Atom objects, is the objective of the GDL. On such an expression, one may invoke the toString() method to obtain a simplified syntactic form, or the calculate() method to evaluate it.

This section was a brief overview of Cedar.Gdl library's archtecture. For detailed information about how use it, please refer to the Cedar.Gdl user manual [31] and its Javadoc documentation.[11] In the next section, we present original uses of the GDL and how to realize them with the Cedar.Gdl library.

[11] http://cedar.liris.cnrs.fr/documents/Cedar.Gdl-javadoc.html

Data: A vertex i and a vertex j with their respective lists of received messages m_i and m_j
Create a list L_i of the variables presents in i and not in j;
Create a result object, r initialized with the local kernel of i;
while *All the messages in m_i haven't been picked* **do**
  pick a message m in m_i and do the multiplication of m by r and store the result in r;
end
if *L_i is not empty* **then**
  Use the method `doAdditionOnSet()` of the $+$ monoid on r and L_i;
end
r is now the message that must be sent from i to j;

Algorithm 5: The message-passing algorithm implementation

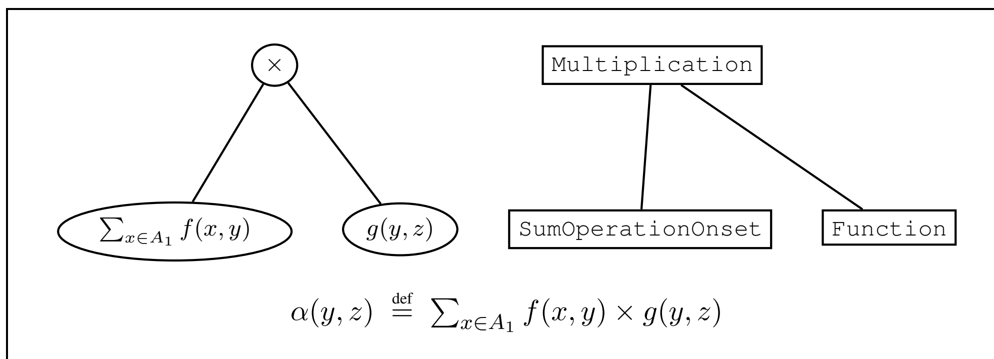


Figure 17: An expression evaluation and decomposition and a possible representation

5 Original Contribution

5.1 Constraint satisfaction with the GDL

This section explains how to use our GDL library for solving Constraint-Satisfaction Problems (CSP). The idea is to use functions to represent the constraints and then use the `Cedar.Gdl` to solve the problem so created.

Bistarelli, Rossi and Montanari [7] discuss the use of semirings as a framework for Constraints-Satisfaction Problems. A commutative semiring, with the logic operation "**and**" as operator for the multiplicative monoid, "**or**" as additive monoid and the set of boolean as Set K, can support a Constraint-Satisfaction Problem. Because of its genericity the library can handle such commutative semiring.

Constraint-Satisfaction Problems are mathematical problems, defined as a set of objects that must satisfy a number of constraints. This kind of representation can be use to represent real-life cases as well as fully abstract problems. More formally, a CSP is defined by a set of variables $x_1, \ldots, x_n$ join to a set of constraints $c_1, \ldots, c_m$. Each

variable x_i have possible values on a set S_i, and is constrained by a set of constraints X_i. A parallel with the input of the GDL, with local domains and local kernels, can be made. The first step is to express the constraints as functions. A constraint, and its equivalent function must return identical output for a same input. Then, for each function a local domain must be created with the variables needed to execute the function. The local kernel will be the function itself. Once the problem have been transformed, an additional pair $\langle$*local-domain*; *local-kernel*$\rangle$ must be created. The set of all the variables must be used as local domain and the multiplication identity as local kernel.

By merging all these facts, it is possible to transform a Constraint-Satisfaction Problem into a GDL instantiation that can evaluate an or/and expression, thereby solving the constraint. The proof of this is in Appendix Section A.

An example of a CSP that can be solved using the `Cedar.Gdl` library is the "N-*Queens problem*."

DEFINITION 12 (N-QUEENS PROBLEM) *The N-Queens problem consists in positioning N chess queen in a $N \times N$ chessboard such that not two queens can take each-over. For those not familiar with chess, this is equivalent to not having two pieces in the same row, column or diagonal.*

Let us takes the example of the 4-Queens problem. The first step is the conversion of the constraints into equivalent functions. The constraints of the N-Queens problem lies with the queens' positioning. However, this global constraint can be split into several simpler constraints. Let us assume that there exists a function that takes two queens's positions and returns whether or not the queens capture one another as a Boolean. If we use this function for all the different combinations of two queens, then the conjunction of these functions is equivalent to checking whether a combination of queens' positions is valid or not. Assume also that we can find a way to transform the queens' positions into variables and enumerate all the possible combinations of two variables. Then, it is possible to transform the N-Queens problem into a GDL problem and solve it with the library.

A queen's position on a chessboard is defined by its coordinate (x,y). In that case the origin must be define, for us the origin will be the top-left corner of the chessboard. Therefore for any queen Q_i $i \in N$, we can define x_i and y_i, respectively the x-coordinate and y-coordinate of Q_i. Then, we can create a function $f(x_a, x_b, y_a, y_b)$. The function f would check whether: $x_a \neq x_b$, $y_a \neq y_b$ and $|x_a - x_b| \neq |y_a - y_b|$. If all these conditions are met, then the two queens can coexist. The only thing left is to find a set of combinations of 4 variables from $2 \times N$, which ensures that no queens can take each-over. Such a mathematical function exists; the combination. Therefore, all the criteria are met, and it is possible to transform the N-Queens problem into a GDL's problem. However, it is possible to reduce greatly the calculations. The goal of the N-Queens problem is to find all the different set of queens. Having Q_1 in position p_1 and Q_2 in p_2 is equivalent to Q_1 in p_2 and Q_2 in p_1. It is then possible to associate a fixed y_i (*i.e.*, y-coordinate) to every x_i (*i.e.*, x-coordinate) variable. Doing so, the number or variables is reduced by half.

For the 4-Queens problem, four variables x_1, x_2, x_3 and x_4 are associates with the

x-coordinate 1, 2, 3 and 4, respectively. The associated value of a variable is denoted as x_{A_i}, x_{B_i}. Therefore, the function that checks whether two queens are in a correct position must be adapted. Now, the function call $f(x_a, x_b)$ must verify the following two properties:

$$x_a \neq x_b, x_{B_a} \neq x_{B_b} \text{ and } |x_a - x_b| \neq |x_{B_a} - x_{B_b}|.$$

The problem described in Table 2 can then be instantiated.

Local Domain	Local Kernel
$\{x_1, x_2\}$	$f(x_1, x_2)$
$\{x_1, x_3\}$	$f(x_1, x_3)$
$\{x_1, x_4\}$	$f(x_1, x_4)$
$\{x_2, x_3\}$	$f(x_2, x_3)$
$\{x_2, x_4\}$	$f(x_2, x_4)$
$\{x_3, x_4\}$	$f(x_3, x_4)$
$\{x_1, x_2, x_3, x_4\}$	$true$

Table 2: GDL formulation for the 4-Queens problem

In conclusion, the GDL algorithm can be applied for solving CSPs. However, the transformation of the constraints into functions is not always a straightforward exercise.

5.2 A modification of the GDL algorithm

In most cases, the algorithm proposed in the GDL works fine. However, during the implementation of some Belief Propagation networks, we realized that there are cases that do not quite fit the GDL algorithm as described. This section discusses such cases and solutions we propose to make the GDL capable of handling them.

As discussed earlier, the basic idea for the creation of the junction tree is to separate two cases: the case where the maximal weight spanning tree is a junction tree, from the case where the triangulation of the moral graph is needed. This is in the second case that the problem resides. After the creation of the moral graph, the next step is to triangulate it and form a junction tree from its cliques. Once this is done, all the local domains must be associated with a newly created clique and each clique must be merged with one local domain. However, in some cases not all the cliques can be merged because some of them cannot be associated with any local domains. In such cases, the resulting junction tree contains more vertices than the initial local domain, which makes the message-passing algorithm impossible—see Figure 18 for example.

This can be seen as a GDL problem by defining the local domains and local kernel (as described in Table 3). For this problem the creation of the local domain graph, and a maximal-weight spanning tree will lead to the conclusion that the maximal-weight spanning tree is not a junction tree. Therefore, the creation of the moral graph, and its

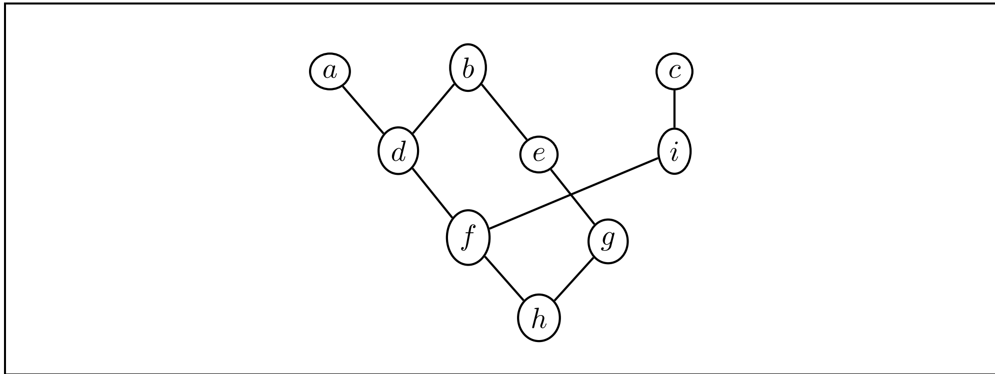


Figure 18: A belief-propagation network

Local Domain	Local Kernel
$\{a\}$	$p(a)$
$\{b\}$	$p(b)$
$\{c\}$	$p(c)$
$\{a, b, d\}$	$p(d \mid a, b)$
$\{b, e\}$	$p(e \mid b)$
$\{c, i\}$	$p(c \mid i)$
$\{d, f, i\}$	$p(f \mid d, i)$
$\{e, g\}$	$p(g \mid e)$
$\{f, g, h\}$	$p(h \mid f, g)$

Table 3: GDL formulation for the Bayesian network in Figure 18

triangulation is required. The method was described in Section 3. On our ezample, it yields the moral graph of Figure 19 and its triangulation shown in Figure 20.

It is easily verifiable that the graph is chordal. We can identify the following list of cliques: $(\{c, i\}, \{i, f, d\}, \{f, h, g\}, \{f, g, d\}, \{d, g, e\}, \{d, e, b\}, \{a, d, b\})$. In this example, we can see that no local domain can be merged with the clique $\{f, g, d\}$. Therefore, if a junction tree was to be created from this triangulated moral graph a node would not be associated with a local kernel. This would prevent the GDL algorithm to be performed. Later in this section, we present two algorithms to deal with such cases, and proofs of their correctness.

The first solution is trivial and consists in the creation of the junction tree associated to the triangulated moral graph with the additional vertices (for now, for convenience, we call such a tree an original tree). Then a vertex corresponding to a non associated clique must be merge with one of it's neighbor. The resulting cell will have a local domain equal to the union of the variables presents in both vertices. The local domain must be the one of the cell which is merged to the vertex of the non associated cliques. Most of the time, several solutions are available for the choice of the vertex to merge.

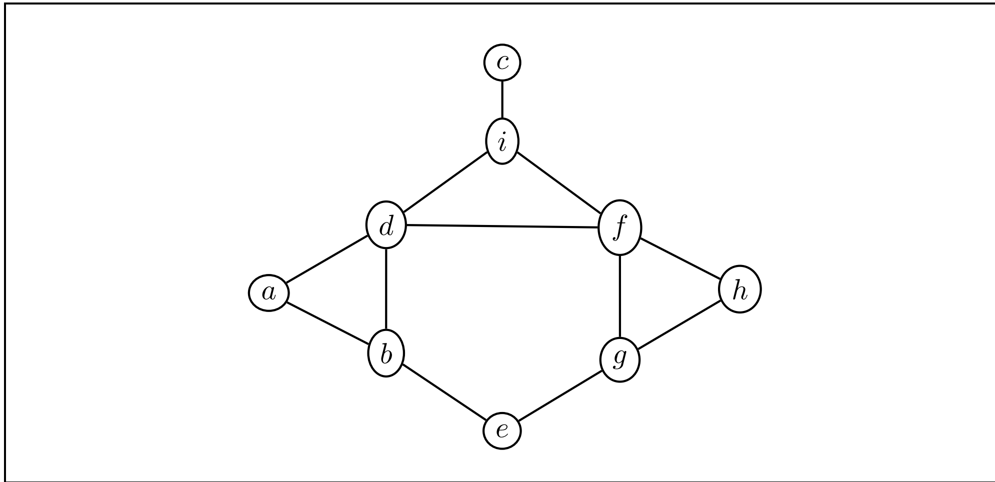


Figure 19: The moral graph for the belief network in Figure 3

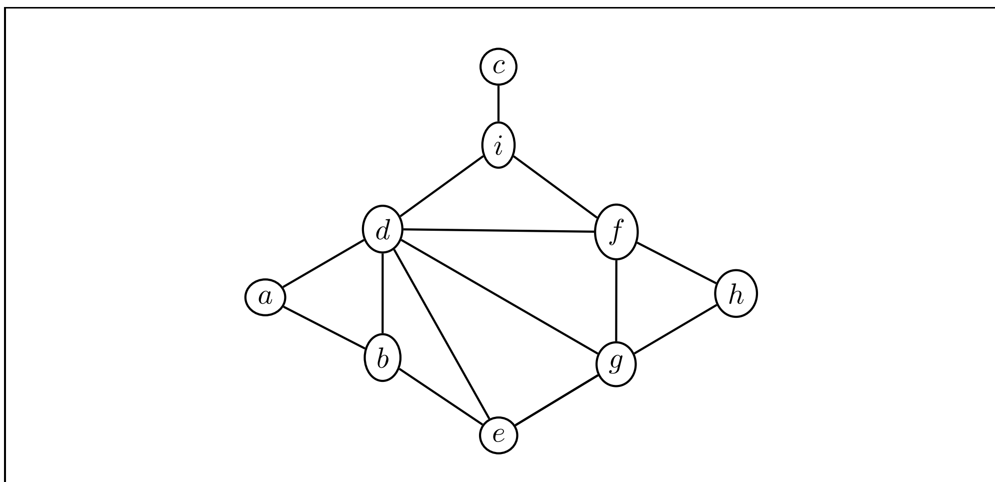


Figure 20: A triangulation of the moral graph of Figure 3

The Algorithm 6 display a heuristic that behave efficiently on most cases.

The merge procedure must be repeated until there is no non-associated vertex left in the tree. The resulting tree is called a merged tree. This method enlarges local domains, which is why the choice regarding which cells merge must be done carefully. But, doing so allows the GDL algorithm to be performed on the newly created junction tree.[12]

The second solution is to create a dummy function equal to the identity of the multiplication monoid from the current commutative semiring, and define it as the local kernel of the non associated clique. Then, the creation of a vertex for the junction tree is possible by defining the set of variable as local domain, and using the created local kernel. The proof is described in the appendix, Section B.2.

[12] See the proof in Appendix Section B.1.

if *there are neighbors with a local kernel associated* **then**
  if *there is several neighbors* **then**
    | Select the one for which the created clique will be the smallest
  else
    | Pick the only one
  end
  ;
else
  Merge the two vertices. However in this case the created vertex will also have to be associated;
end

Algorithm 6: Vertex-selection algorithm

5.3 Modeling Allen's interval algebra with the GDL

In this section, we discuss how one may use the generic algorithm of the GDL to perform qualitative temporal reasoning using Allen's Interval Algebra.[13].[14]. We also discuss some ideas for future developments.

In 1983, Allen created thirteen basic relations between time intervals (Figure 21. Theses thirteen relations are able to describe any pair of time intervals.[15] This representation of time can be used for temporal reasoning, and has been in so used AI. It has since been extended in various ways [3, 22, 4, 33]. As it is possible to formalize the Allen's Interval Algebra operations into a commutative semiring (See Section 5.3 for a detailed explanation), it is then possible to use the GDL algorithm to solve this kind of problems.

Allen's Interval Algebra represents all the possible relations between two time's intervals. It also provides tools to compute operations between them, and reason about them. An important point about this approach is that it is completely qualitative (*i.e.*, no time span are used for interval's description).

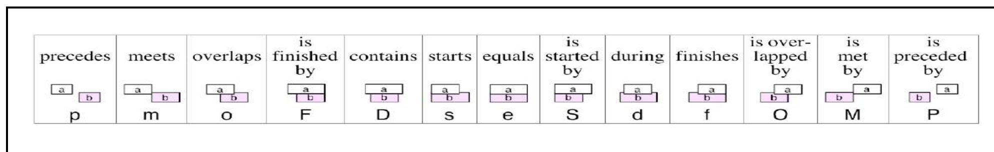


Figure 21: Allen's thirteen basic relations

Allen's Time-Interval Algebra has no relation with the Generalized Distributive Law. However, as said previously, the GDL is a parametric solving method for any application in which a ring structure can be exposed. Therefore, if such organisation could

[13] http://www.ics.uci.edu/~alspaugh/cls/shr/allen.html
[14] This work was not completed in [32]
[15] For more details about these relations, please refer to [29].

be found it would be possible to use the `Cedar.Gdl` to solve problem related to Allen's-Interval Algebra.

An example of a problem could be: At a moment A two events can occurs a_1 and a_2. Then, at a moment B the even b_1 can occur if a_1 appended in A, and b_1 and b_2 can occur if a_2 appended. Now let's transform this problem into a GDL instance.
The first step is to create variables to compose the local domains. These variables would represent the different possibles events at a given moment (*e.g.*, a variable x_A would have for set $\{a_1, a_2\}$). It is possible to use this to define local domains. Each of them must represent a given time and its relation with past events. Therefore, a local domain must be composed of the variable representing the events, and the variables of the related previous events (*e.g.*, for A a local domain $\{x_A\}$ must be created, and for B $\{x_A, x_B\}$). The next step is to instantiate the local kernels. A local kernel must be function of its local domain. Therefore, each local domain d_i must be associated with a local kernel k_i defined by a function f_i. A function must return the Allen's relation between the events described by the variables of the associated local domain.

In [4], a semiring instance that could handle Allen's Interval Algebra can be made explicit. This semiring has the logical "**or**" as $+$ operator and Allen's composition as $\times$ operator. This semiring allow to compose Allen's relations and thus, to solve Allen' Interval problems. By using the previously defined representation for an Allen's problem, it is possible to transform it into a GDL instance.

However, the function are as numerous as the number of local domain, and Allen's expression are between two intervals. Consequently, we have an additional function. This function is the one associated to the first moment of the sequence (*i.e.*, in our example A). Consequently, we introduce a neutral element for Allen's Algebra Id that must be returned by this function (*i.e.*, $f_A(x_A, x_b)$ must return Id). This representation is then capable of finding the result of the composition of several Allen's interval. However, the library `Cedar.Gdl` cannot detect incorrect expressions and therefore, cannot determine whether a expression is actually satisfied or not.

This is initial work toward expressing Allen's temporal algebra in the context of the GDL. Although it needs to be completed and tested, we have shown that it should be possible to be cast into the framework of our GDL library.

6 Implemented Instances of the `Cedar.Gdl` Library

This section presents experiemental test results otained with some actual implemented cases of the GDL such as the Fast-Hadamard Transform (Section 6.1) and Judea Pearl's Belief Propagation (Section 6.2).

All the computations were performed using a laptop with an IntelCore i5-3210M and 8G of RAM. All the experiments listed below are reproducible as all these cases are implemented in the library `Cedar.Gdl`. All tests were performed five times and the average value of the execution time recorded.

6.1 The Fast-Hadamard Transform

We implemented two ways for doing the Fast-Hadamard Transform: one with a function of three variables as in the example of the Generalized Distributive law, and one with a function of ten variables. In both cases, the function returns the sum of its parameters. To compare the results, we use Apache's Commons Math Library[16] (`commons-math3-3.2.jar`). In Table 4, we can see that the `Apache` library

Try	1st	2nd	3rd	4th	5th	Average
`Cedar.Gdl`	39	40	38	38	39	**39**
`Apache`	9	9	9	9	9	**9**

Table 4: Execution time in **MS** for the FHT of a function of three variables.

takes only a fourth of the time of the `Cedar.Gdl`. This difference is due to the fact that the two methods are different, the `Apache` library use as entry a list of reel which correspond to the result of the function. In the `Cedar.Gdl` cases, the input in only the function and the library also compute the result of the function. Furthermore, the GDL algorithm performs better in the all-vertex problem because of all the stored intermediate results. In the all-vertex problem, after computation of the first vertex, most of the calculations are already done for the other vertices' of the graph. The second reason is that most of the time is not spent in the message passing but in the construction of the junction tree.

It comes as a result that the library is better suited for problems where the all-vertex method is needed.

6.2 Judea Pearl's belief propagation

As explained in Section 6.1, in order to test our implementation on a problem that requires the all-vertex GDL version, we developed Judea Pearl's Belief Propagation as a second application.

This section presents the results using the library on different Bayesian Networks. It makes little sense to compare these results with other existing solutions because of the variety of software and libraries that exist. However, we report the results we obtained so that anyone may compare the times we obtained with those obtained by one's own solution. Each table displays the time required to compute the probabilities of all the nodes in the tree (*i.e.*, for a node with 100 nodes, all 100 probabilities are computed).

The calculation over a simple Bayesian network as described in Figure 22 are extremely quick. However, the number of nodes drastically increase the computation time. If done in the most straightforward manner the calculation time should double for each new node added to the tree. However, the GDL allows to save a large part of the computation. This allow big Bayesian networks to be computed. We can see in

[16] `http://commons.apache.org/proper/commons-math/`

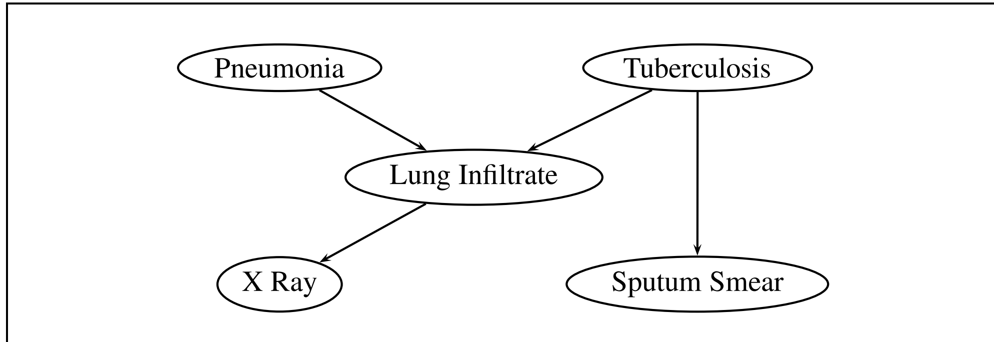


Figure 22: A basic belief-propagation network

Try	1st	2nd	3rd	4th	5th	Average
`Cedar.Gdl`	38	34	36	38	39	**37**

Table 5: Execution time in milliseconds for the example displayed in Figure 22

Figure 6 that it takes 15 seconds to compute the Belief Propagation over a tree with three-hundreds nodes.

Try	1st	2nd	3rd	4th	5th	Average
`Cedar.Gdl`	15.1	15.1	15.2	15.1	15.0	**15.1**

Table 6: Execution time in seconds for a Bayesian network with 300 nodes

However, we can see in Figure 7 that the required time augments very quickly when nodes are added. It is important to notice that the time also depends on the topology of the Bayesian Network. Indeed, the topology of the Bayesian Network impacts the junction tree and consequently the simplification.

From these experiments with, and tests of, our `Cedar.gdl` abstract librarywhen for the regeneration of Bayesian network applications, we could observe that it performs as predicted in the Aji-McEliece paper.

6.3 Constraint processing

We now describe how to cast constraint-processing as a GDL instance. Section 6.3.1 recalls the essence of Constraint-Satisfaction Problems. Then, Section 5.1 shows how, under certain conditions, it is possible to use the `Cedar.Gdl` library to generate a Constraint-Satisfaction Solver.

Try	1st	2nd	3rd	4th	5th	Average
Cedar.Gdl	1.44	1.44	1.44	1.43	1.44	**1.44**

Table 7: Time in milliseconds for a Bayesian network with 310 nodes

6.3.1 Constraint-satisfaction problems

Citing [7], it is claimed that:

> *"...Constraint-Satisfaction Problems (CSPs) [...] are a very expressive and natural formalism to specify many kinds of real life problems. In fact, problems ranging from map coloring, vision, robotics, job-shop scheduling, VLSI design, etc., can easily be cast as CSPs and solved using one of the many techniques that have been developed for such problems or subclasses of them."*

Formally, a Constraint-Satisfaction Problem is defined by a set of variables $x_1, \ldots, x_n$ occurring in a set of constraints $c_1, \ldots, c_m$. Each variable x_i have possible values on a set S_i, and is constrained by a set of constraints X_i.

Most of the algorithms for solving CSPs search systematically through the possible assignment of values to variables. Such algorithms are complete in that they guarantee finding a solution if one exist, or prove that none exists. In [8], Brailsford, Potts, and Smith, present a review of algorithms and applications related to CSP. For example, the well-known problem of the location of facilities is a type of OR problem which can be seen as a CSP. For every problem, very efficient algorithms have been developed in order to solve them in an extremely efficient manner. However, this requires to have several algorithms depending on the type of problem for optimal resolution.

In addition, more flexible types of CSP have been proposed, such as Probabilistic CSP, Fuzzy CSP, or Weighted CSPs.[17] With these more general types of CSPs, is it possible to model a larger number of problems as CSPs. However, different algorithms are required depending on the type of the problem. An abstract library using the algorithm of the GDL enables rendering different types of CSPs with a single algorithm (*viz.*, the GDL by mere instantiation of an appropriate commutative semiring. Then, optmizing a single abstract algorithm as opposed to several specific ones offers an interesting alternative. Supplementing such an abstract design with classes of problem-dependent heuristics factors out all common features, while staying flexible to exploit addditional structure specific to each instance.

How to use the algorithm of the GDL in order to instantiate a Constraint-Satisfaction Problem is discussed at greater length in Section 5.1.

[17] http://ktiml.mff.cuni.cz/~bartak/constraints/extend_csp.html

6.3.2 Constraint solving

Here, we present results obtained for some Constraint-Satisfaction Problems (CSP). The CSP of our experiemnt is the N-Queens Problem (see Definition 12). We did measurements with four and six queens. The timer stops only when all the solutions have been found.

Try	1st	2nd	3rd	4th	5th	Average
`4-Queens`	44		45	44	44	**44**
`6-Queens`	402	402	403	402	403	**402**

Table 8: Execution time in milliseconds for the N-Queens problem

As can be seen in Table 8, the 4-Queens process is executed quickly. However, the 6-Queens problem takes some time. This is because the `Cedar.Gdl` library works similarly to Prolog and its algorithm mimics a backtracking algorithm. The more possibilities there are the more time it will take to explore all of them.

7 Conclusion

7.1 Recapitulation

Our experimentation with the design, implementation, and use of `Cedar.Gdl` Java library has yielded some interesting results. For one, its genericity has been verified and tested to regenerate test examples. Further, its reusablity has also been verified as we used on a new use case (constraint solving) to generate a constraint solver for the N-Queens' problem. More applications in varied domains can be derived and generated in a similar manner. As long as the commutative-semiring structure required by the library is respected, it is possible to use the `Cedar.Gdl` Java library to do so.

7.2 Perspectives

The most intriguing future work would be to implement Allen's Interval Algebra fully. Since its basic commutative-semiring structure has been exposed, it is now possible to instantiate a GDL problem on Allen's Interval Algebra to generate an interval-logic reasoner's implementation. It would be as well interesting to investigate other novel instantiations of the GDL such as, for example, Fuzzy Set reasoning. Finally, the GDL algorithm may show potential for running on parallel architectures for Big Data such as Hadoop/MapReduce to optimize and solve large computational mathematical problems.

A Correctness of the construction of Section 5.1

PROOF A vertex calculates its state only when it has received messages from all it neighbors. Furthermore, a vertex only sendsf a message to another vertex when it has received messages from all its neighbors. Consequently, a vertex only calculates its state when it has received, or been forwarded, a message from every other vertex in the junction tree.

When a vertex v_i needs to send a message to a vertex v_j, the following rule is used:

$$\mu_{i,j}(x_{S_i} \cap x_{S_j}) = \sum_{x_{S_i} \setminus x_{S_j} \in A_{S_i} \setminus A_{S_j}} \alpha_i(x_{S_i}) \prod_{\substack{v_k \textbf{ adj } v_i \\ k \neq j}} \mu_{k,i}(x_{S_k} \cap x_{S_i}) \tag{8}$$

Furthermore, the formula used to calculate the state of a vertex is:

$$\sigma_i(x_{S_i}) = \alpha_i(x_{S_i}) \prod_{v_k \textbf{ adj } v_i} \mu_{k,i}(x_{S_k} \cap x_{S_i}). \tag{9}$$

From Equation (8), and Equation (9), we can conclude that the state of the vertex corresponds to the multiplication of all the local kernels of the tree. However, the multiplicative operation, for a constraint satisfaction problem, is the logic operation "**and**." Therefore, the vertex evaluates to the validity check of all the constraint functions. Furthermore, because the addition operation is the logic operation "**or**," and because of Equation (8), an "**or** operation is performed for all the values of the variables which are not included in the current local domain. Hence, the correctness of our constraint-solving instantiation is entailed by the algebraic correctness of the GDL algorithm for evaluating a commutative-semiring expression proven by Aji and McEliece in [2]. □

B Correctness of the GDL Modification of Section 5.2

B.1 First method

PROOF The GDL algorithm requires a junction tree. Hence, if the merged tree can be proven to be a junction tree, then the GDL algorithm can be used. Furthermore, the transformation process from the original junction tree to the merged tree can be seen as a succession of merges of two vertices. If a merge of two vertices in a junction tree creates a tree which is still a junction tree, then the method is correct. Let us show this.

We denote the non-associated vertex as v_a and the vertex that must be merged as v_b. Using the same conventions as before, the associated set of variables present in each vertex is, respectively, denoted by S_a and S_b.

The original tree, which is a junction tree, can be seen as a maximal weight spanning tree. In the GDL, the authors define w^* and $w_{\max}$ as:

$$w^* = \sum_{i=1}^{M} |S_i| - n \tag{10}$$

and $w_{\max}$, as the weight of the maximal weight spanning tree.

For the original tree we have $w_{\max} = w^*$. But, if two vertices are merged in the tree w^* and $w_{\max}$ will be impacted. First, the edge between the two merged vertices will no longer exist, so $w_{\max}$ will decrease from the weight of the edge, $|(S_a \cap S_b)|$. Then, the vertex v_a will no longer exist so, w^* will decrease by $|S_a|$. However, because of the merge, v_b will be extended by $|S_a - (S_a \cap S_b)|$. The result is that, in total, w^* will increase by $|S_a - (S_a \cap S_b)|$. And so, $w^*_{new}\ w^*_{old} - |(S_a \cap S_b)|$. At this stage, we have $w_{\max} = w^*$ for the merged tree. However, one question remains. Do the variables added to v_b increase the weight of the edges between v_b, and its neighbors? In that case $w_{\max}$ would be different from w^*. In a junction tree, for any two vertices v_i and v_j, the intersection of the corresponding labels, (*viz.*, $v_i \cap v_j$), is a subset of the label on each vertex, on the unique path from v_i to v_j. Consequently, no neighbors of v_b can contain a variable which was not in $(S_a \cap S_b)$. Therefore the merged tree is a junction tree. □

B.2 Second method

PROOF Let v_{id} be the vertex that has the identity of the multiplicative monoid set as local kernel. Intuitively, v_{id} will act as a simple "message's forwarder" in the propagation phase of the GDL. When it receive messages, v_{id} applies the $+$ operation, over a set variable, on the product $\times$ of all the received messages.

In a junction tree, when a vertex v_i needs to send a message to a vertex v_j, the following rule is used by the GDL:

$$\mu_{i,j}(x_{S_i} \cap x_{S_j}) = \sum_{x_{S_i} \backslash x_{S_j} \in A_{S_i} \backslash A_{S_j}} \alpha_i(x_{S_i}) \prod_{\substack{v_k \ \mathbf{adj}\ v_i \\ k \neq j}} \mu_{k,i}(x_{S_k} \cap x_{S_i}) \tag{11}$$

When the local kernel is set to the identity of the multiplication monoid, the previous rule is simplifies to:

$$\mu_{i,j}(x_{S_i} \cap x_{S_j}) = \sum_{x_{S_i} \backslash x_{S_j} \in A_{S_i} \backslash A_{S_j}} \prod_{\substack{v_k \ \mathbf{adj}\ v_i \\ k \neq j}} \mu_{k,i}(x_{S_k} \cap x_{S_i}) \tag{12}$$

As seen in the first method, it is possible to merge the non-associated vertex with an other vertex, and this will allow the GDL algorithm to work. Using this observation, the extended junction tree appears as a merged junction tree with the difference that, instead of having merged cell, we keep the cells separate. We call such a pair of cells a cell system. We must verify that the messages, that passed through the cell system are the same as the ones that would transit through the merged cell.

The set of variables in the merged cell is equal to the union of the set of variables of the cell's system. Consequently, once the messages have pass through the cell system, the variables, on which the $+$ operation is performed, are the same in both methods. This is because of the message rule Figure 11. Furthermore, the multiplication's identity multiplied by a local kernel α is equal to α, which would be the local kernel of the merged cell.

This proves that this method is correct. □

References

[1] Srinivas M. Aji. *Graphical Models and Iterative Decoding*. PhD thesis, California Institute of Technology, Pasadena, CA, USA, May 2000. [Available online[18]].

[2] Srinivas M. Aji and Robert J. McEliece. The generalized distributive law. *IEEE Transactions on Information Theory*, 46(2):325–343, March 2000. [Available online[19]].

[3] James F. Allen and Patrick J. Hayes. Moments and points in an interval-based temporal logic. *Computational Intelligence*, 5(4):225–238, May 1990. [Available online[20]].

[4] Silvana Badaloni and Massimiliano Giacomin. A fuzzy extension of Allens interval algebra. In Evelina Lamma and Paolo Mello, editors, *AI*IA 99: Advances in Artificial Intelligence—Selected Papers of the 6th AI*AI Congress*, pages 155–165, (Bologna, Italy), September 1999. Italian Association for Artificial Intelligence, Springer-Verlag. Vol. 1792 [Available online[21]].

[5] Ann Becker and Dan Geiger. A sufficiently fast algorithm for finding close to optimal junction trees. In *Proceedings of the Twelfth International Conference on Uncertainty in Artificial Intelligence*, pages 81–89. Morgan Kaufmann Publishers Inc., 1996.

[6] P. George Benson, Shawn P. Curley, and Gerald F. Smith. Belief assessment: An underdeveloped phase of probability elicitation. *Management Science*, 41(10):1639–1653, October 1995. [Available online[22]].

[7] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Constraint solving over semirings. In *IJCAI (1)*, pages 624–630. Citeseer, 1995.

[8] Sally C. Brailsford, Chris N. Potts, and Barbara M. Smith. Constraint satisfaction problems: Algorithms and applications. *European Journal of Operational Research*, 119(3):557–581, 1999.

[9] Rina Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113(1–2):41–85, September 1999.

[10] Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[11] Arthur P. Dempster, Nan M. Laird, Donald B. Rubin, and *et al.*. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal statistical Society*, 39(1):1–38, 1977.

[12] Yariv Ephraim and Neri Merhav. Hidden markov processes. *IEEE Transactions on Information Theory*, 48(6):1518–1569, 2002.

[13] Jensen V. Finn. *An introduction to Bayesian networks*, volume 210. UCL press London, 1996.

[14] John C. Gower and G.J.S Ross. Minimum spanning trees and single linkage cluster analysis. *Applied statistics*, pages 54–64, 1969.

[15] Anthony T.S. Ho, Jun Shen, and Soon H. Tan. Robust digital image-in-image watermarking algorithm using the fast hadamard transform. In *International Symposium on Optical Science and Technology*, pages 76–85. International Society for Optics and Photonics, 2003.

[18] `http://thesis.library.caltech.edu/1340/`
[19] `http://authors.library.caltech.edu/1541/1/AJIieeetit00.pdf`
[20] `https://urresearch.rochester.edu/.../&itemFileId=9760`
[21] `http://www.researchgate.net/.../file/79e4150b86d38c7b8b.pdf`
[22] `http://mansci.journal.informs.org/cgi/content/abstract/41/10/1639`

[16] Uffe Kjærulff and Anders Madsen. Probabilistic networks—an introduction to Bayesian networks and influence diagrams, May 2005. [Available online[23]].

[17] Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956.

[18] Frank R. Kschischang, Brendan J. Frey, and H.-A. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 47(2):498–519, 2001.

[19] Harri Lähdesmäki and Ilya Shmulevich. Learning the structure of dynamic Bayesian networks from time series and steady state measurements. *Machine Learing*, 71(2–3):185–217, 2008.

[20] David J.C. MacKay. *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, 2003.

[21] Todd K. Moon. The expectation-maximization algorithm. *IEEE Signal Processing Magazine*, 13(6):47–60, 1996.

[22] Kamel Mouhoub and Jia Liu. Probabilistic temporal network for numeric and symbolic time information. *IEEE Transactions on Systems, Man, and Cybernetics*, pages 3399–3404, October 12–15, 2008. [Available online[24]].

[23] Kevin Murphy. A brief introduction to graphical models and Bayesian networks, 1998. [Available online[25]].

[24] Radu Stefan Niculescu, Tom M. Mitchell, and R. Bharat Rao. Bayesian network learning with parameter constraints. *Journal of Machine Learning Research*, 7:1357–1383, July 2006. [Available online[26]].

[25] Angmin O, Jae Won Lee, Sung-Bae Parkl, and Byoung-Tak Zhangl. Stock trading by modelling price trend with dynamic Bayesian networks. In *Intelligent Data Engineering and Automated Learning (IDEAL 2004)*, pages 794–799. Springer, Exeter, UK, August 25–27 2004.

[26] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988. Revised 2nd printing.

[27] Judea Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 1988.

[28] Alok K. Porwal, E. John M. Carranza, and Martin Hale. Bayesian network classifiers for mineral potential mapping. *Computers & Geosciences*, 32(1):1–16, February 2006. [Available online[27]].

[29] Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational intelligence*, 9(3):268–299, 1993.

[30] Lawrence R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, February 1989.

[31] Kevin S. Sancho. Cedar.Gdl Java Library User Manual. *CEDAR* Technical Report Number 10, Université Claude Bernard Lyon 1, Villeurbanne, July 2014. [Available online[28]].

[23] http://www.cs.aau.dk/~uk/papers/pgm-book-I-05.pdf
[24] http://202.154.59.182/mfile/files/...Information.pdf
[25] http://www.cs.ubc.ca/~murphyk/Bayes/bnintro.html
[26] http://jmlr.csail.mit.edu/papers/volume7/niculescu06a/niculescu06a.pdf
[27] http://portal.acm.org/citation.cfm?id=1650536
[28] http://cedar.liris.cnrs.fr/interns/KevinSancho/KevinSancho/ctr10.pdf

[32] Kevin S. Sancho. The Cedar.Gdl java library for the generalized distributive law. Master's thesis, Université Claude Bernard Lyon 1, Computer Science Department, Villeurbanne, France, June 2014.

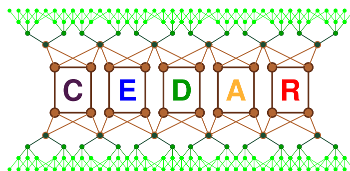
[33] Steven Schockaert, Martine De Cock, and Etienne E. Kerre. Fuzzifying allens temporal interval relations. *IEEE Transactions on Fuzzy Systems*, 16(2):517–533, April 2008. [Available online[29]].

[34] Andrew J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2):260–269, 1967.

[35] Geoffrey Zweig and Stuart Russell. Probabilistic modeling with Bayesian networks for automatic speech recognition. *Australian Journal of Intelligent Information Processing*, 1999. [Available online[30]].

[29] http://www.researchgate.net/publication/.../file/72e7e52126b9eeb7d8.pdf

[30] http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.32.9156



Technical Report Number 9

The `Cedar.Gdl` Java Library for the Generalized Distributive Law

Kevin Sancho and Hassan Aït-Kaci

July 2014