NENOK - A SOFTWARE ARCHITECTURE FOR GENERIC INFERENCE

MARC POULY∗
Cork Constraint Computation Center
University College Cork, Ireland
marc.pouly@unifr.ch

Received (20 August 2008)
Revised (27 July 2009)
Accepted (05 November 2009)

Computing inference from a given knowledgebase is one of the key competences of computer science. Therefore, numerous formalisms and specialized inference routines have been introduced and implemented for this task. Typical examples are Bayesian networks, constraint systems or different kinds of logic. It is known today that these formalisms can be unified under a common algebraic roof called valuation algebra. Based on this system, generic inference algorithms for the processing of arbitrary valuation algebras can be defined. Researchers benefit from this high level of abstraction to address open problems independently of the underlying formalism. It is therefore all the more astonishing that this theory did not find its way into concrete software projects. Indeed, all modern programming languages for example provide generic sorting procedures, but generic inference algorithms are still mythical creatures. NENOK breaks a new ground and offers an extensive library of generic inference tools based on the valuation algebra framework. All methods are implemented as distributed algorithms that process local and remote knowledgebases in a transparent manner. Besides its main purpose as software library, NENOK also provides a sophisticated graphical user interface to inspect the inference process and the involved graphical structures. This can be used for educational purposes but also as a fast prototyping architecture for inference formalisms.

Keywords: Generic Inference; Distributed Inference; Valuation Algebra; Local Computation; Semiring Valuation Algebra; NENOK.

∗Research supported by the Swiss National Foundation, grant PBFRP2-124190
1. Introduction

Writing generic software became a supreme discipline in modern computer science. Consider for example the plain task of writing a program to sort a list of items. It is clearly beyond all questions to reinvent such a sorting procedure for lists containing numbers, strings or other objects. In addition, we do not need fundamentally different algorithms to bring the list in either ascending or descending order. All these needs can in fact be fulfilled by a single generic algorithm, on condition that the underlying mathematical requirements (i.e. the total ordering between list elements) are understood. However, what appears naturally for a sorting procedure is often ignored in case of more challenging applications. It is known today that a single, generic message-passing algorithm hides behind such sophisticated applications as for example the fast Fourier transform, the evaluation of Bayesian networks, satisfiability of constraint or logical systems, linear decoding, query answering in relational databases or sparse matrix computation. More concretely, it is the mathematical structure of a valuation algebra that permits the application of this algorithm, such as the total order allows the application of a generic sort procedure in the introductory example. Nevertheless, software designers are still considering these tasks as isolated fields of computer science and hence ignore the untapped potential to realize generic software libraries. This article strikes a new path and presents a software architecture called NENOK\textsuperscript{a} that bears on an abstract representation of the valuation algebra upon which a library of generic inference procedures including the above message-passing algorithm and many of its variations are implemented. This means that once the abstract valuation algebra framework has been instantiated for a concrete formalism, the above applications simplify to a single method call. Today, a large number of formalisms are known which satisfy the structure of a valuation algebra and therefore qualify for an instantiation of the NENOK framework. Among them are discrete and continuous probability distributions, Dempster-Shafer theory, various constraint systems, possibilistic formalisms, systems of equations and inequalities over fields and semirings, relations or different types of logics.

The main contribution of the NENOK framework is to provide for the first time a library of generic inference procedures based on the valuation algebra framework. Users must deliver a clearly specified implementation of a formalism that satisfies the structure of a valuation algebra and then gain access to efficient and up-to-date inference procedures for its processing. In addition to this aim as a service library, NENOK also contains an elaborate user interface to inspect the inference process and the involved graphical structures. This is well suited as an experimental platform or fast prototyping architecture for researchers, but has also proved its worth as an educational tool. Altogether, this draws the picture of a plug-in architecture

\textsuperscript{a}The name refers to the world of abstract being in Persian mythology.
for automated inference as shown in Figure 1. NENOK\(^b\) (this article covers version 1.5) is implemented as a Java framework under GPL license and is the only generic inference library based on the theory of valuation algebras that is known to the author. Since inference procedures are naturally described as distributed algorithms, NENOK further includes a communication infrastructure based on a Jini service federation to perform generic inference with distributed data.

![Fig. 1. NENOK as plug-in architecture for automated inference.](http://diuf.unifr.ch/tcs/nenok.htm)

This article starts with a short introduction to valuation algebras. Based on this algebraic language, we then express the inference problem and outline one generic algorithm for its solution, exemplary for all further inference algorithms that are implemented in NENOK. This will be sufficient to make a first call on NENOK and to understand its core functionalities. Afterwards, we study a particular family of valuation algebras called semiring valuations and learn how they are supported by the NENOK framework. Semiring valuations are in fact representative for a particular technique called generic construction which serves to identify new valuation algebra instances and also simplify the user’s task of implementing formalisms. Finally, we are going to explore the service infrastructure that allows NENOK to perform distributed inference on processor networks. Here, efficient communication is naturally an important topic and we will see how this can be reached mathematically and on an implementation level.

2. Valuation Algebras

We pointed out in the introduction of this article that a deep understanding of the underlying mathematical requirements is indispensable to benefit from a generic implementation. In the case at hand, these requirements are bundled in the valuation algebra framework.\(^1,2,3,4\) The basic elements of a valuation algebra are so-called valuations. Intuitively, a valuation can be regarded as a representation of knowledge

\(^b\)http://diuf.unifr.ch/tcs/nenok.htm
about the possible values of a set of variables. It can be said that each valuation \( \phi \) refers to a finite set of variables \( d(\phi) \) called its domain. For an arbitrary set \( s \) of variables, \( \Phi_s \) denotes the set of all valuations \( \phi \) with \( d(\phi) = s \). With this notation, the set of all possible valuations for a finite variable universe \( r \) can be defined as

\[
\Phi = \bigcup_{s \subseteq r} \Phi_s.
\]

Let \( D \) be the power set of \( r \) and \( \Phi \) a set of valuations with their domains in \( D \). We assume the following operations defined in \((\Phi, D)\):

1. **Labeling:** \( \Phi \to D; \phi \mapsto d(\phi) \).
2. **Combination:** \( \Phi \times \Phi \to \Phi; (\phi, \psi) \mapsto \phi \otimes \psi \).
3. **Marginalization:** \( \Phi \times D \to \Phi; (\phi, x) \mapsto \phi \downarrow x \) for \( x \subseteq d(\phi) \).

These are the three basic operations of a valuation algebra: If we interpret valuations \( \phi, \psi \in \Phi \) as pieces of knowledge or information, the labeling operation \( d(\phi) \) tells us to which questions such a piece refers. Combination \( \phi \otimes \psi \) can be understood as aggregation of information and marginalization \( \phi \downarrow x \) as focusing or extraction of the part we are interested in. We now impose the following set of axioms on \( \Phi \) and \( D \):

1. **Commutative Semigroup:** \( \Phi \) is associative and commutative under \( \otimes \).
2. **Labeling:** For \( \phi, \psi \in \Phi \),
   \[
d(\phi \otimes \psi) = d(\phi) \cup d(\psi).
\]
3. **Marginalization:** For \( \phi \in \Phi \), \( x \in D \) and \( x \subseteq d(\phi) \),
   \[
d(\phi \downarrow x) = x.
\]
4. **Transitivity:** For \( \phi \in \Phi \) and \( x \subseteq y \subseteq d(\phi) \),
   \[
   (\phi \downarrow y) \downarrow x = \phi \downarrow x.
   \]
5. **Combination:** For \( \phi \in \Phi_x \), \( \psi \in \Phi_y \) and \( z \in D \) such that \( x \subseteq z \subseteq x \cup y \),
   \[
   (\phi \otimes \psi) \downarrow z = \phi \otimes \psi \downarrow z \cap y.
   \]
6. **Domain:** For \( \phi \in \Phi \) with \( d(\phi) = x \),
   \[
   \phi \downarrow x = \phi.
   \]

These axioms require natural properties of a valuation algebra regarding information or knowledge modeling. The first axiom indicates that \( \Phi \) is a commutative semigroup under combination. If information comes in pieces, the sequence of their aggregation does not influence the overall result. The labeling axiom tells us that the combination of valuations yields knowledge about the union of the involved domains. Neither do variables vanish, nor do new ones appear. The marginalization axiom expresses the natural functioning of focusing. Transitivity says that marginalization can be performed in several steps. In order to explain the naturalness of the combination axiom, let us assume that we have some knowledge about a domain
in order to answer a certain question. Then, the combination axiom states how the answer is affected if a new information piece arrives. We can either combine the new piece to the given information and focus afterwards to the specified domain, or first remove the uninteresting parts of the new information and combine it afterwards. Both approaches lead to the same result. Finally, the domain axiom expresses stability with respect to trivial marginalization.

Definition 2.1. A system \((\Phi, D)\) with the operations of labeling, marginalization and combination satisfying these axioms is called a valuation algebra.

2.1. The Inference Problem

Inference is the central computational problem of any knowledge representation system. Hence, we are next going to express this task in the abstract language of valuation algebras. Consider a set of valuations \(\{\phi_1, \ldots, \phi_n\} \subseteq \Phi\) called knowledgebase that models the user’s available knowledge. Inference then consists in combining all knowledge pieces and to marginalize the result afterwards onto some queries of interest. This is the statement of the following definition:

Definition 2.2. The task of computing

\[ \phi_{\downarrow x_i} = (\phi_1 \otimes \cdots \otimes \phi_n)_{\downarrow x_i} \]

for queries \(x_i \in \{x_1, \ldots, x_s\}\) where \(x_i \subseteq d(\phi_1 \otimes \cdots \otimes \phi_n)\) is called inference problem.

Inference problems with \(s = 1\) are usually called single-query inference problems, and we speak about multi-query inference problems if \(s > 1\). Further, we refer to the combination \(\phi = \phi_1 \otimes \cdots \otimes \phi_n\) of all knowledgebase factors as the objective function. It follows from the labeling and marginalization axiom that the domains of valuations grow under combination and shrink under marginalization. Moreover, since valuations often have exponential size with respect to their domain, it is in general impossible to compute \(\phi\) explicitly. Some examples that support this observation will be given below. Better methods for the computation of inference problems are therefore needed which confine in some way the domain size of all intermediate results, which can be achieved by alternating between combination and marginalization. This technique is called local computation and applied in all generic inference algorithms implemented in NENOK. Before we actually introduce an example of such an algorithm, we first look at some concrete examples of valuation algebras and expose the semantics of their induced inference problems.

2.2. Examples & Applications I

Formalisms that adopt the structure of a valuation algebra are very large in number and widely spread in computer science. Here, we bring up a couple of them without going into technical details. Instead, our ambition is to convince the reader from the generality of the valuation algebra framework and from the advantages of having
generic inference algorithms to process arbitrary valuation algebras. We thus limit ourselves to the definition of the three operations of labeling, combination and marginalization for each formalism and shortly discuss the semantics of the induced inference problems. Proofs that these formalisms indeed satisfy the valuation algebra axioms can be found in (Ref. 3) and (Ref. 4).

2.2.1. Probability Potentials & Bayesian Networks

A Bayesian network is an efficient representation of a joint probability mass function over a set \( r \) of variables. It is a directed, acyclic graph that represents the direct influences among the variables identified with the nodes of the network. Each node holds a conditional probability table (CPT) to express these influences quantitatively, i.e. node \( X \in r \) stores \( p(X|\text{pa}(X)) \) where \( \text{pa}(X) \) refers to the parents of node \( X \) in the network. These CPTs are the valuations in the valuation algebra of probability potentials where the domain of a CPT \( p(X|\text{pa}(X)) \) is defined as \( d(p) = \{X\} \cup \text{pa}(X) \).

To define the operations of combination and marginalization, we first introduce the notion of configuration. Since all variables are finite in a Bayesian network, we denote by \( \Omega_X \) the set of possible values for variable \( X \in r \) called its frame and write \( \Omega_s \) for the set of all configurations over variables in \( s \subseteq r \),

\[
\Omega_s = \prod_{X \in s} \Omega_X. \tag{1}
\]

The combination of two potentials \( p_1 \) and \( p_2 \) with \( d(p_1) = s \) and \( d(p_2) = t \) corresponds to point-wise multiplication: For each configuration \( x \in \Omega_{s\cup t} \) we define

\[
p_1 \otimes p_2(x) = p_1(x^{\downarrow s}) \cdot p_2(x^{\downarrow t}),
\]

where \( x^{\downarrow s} \) and \( x^{\downarrow t} \) denote the restriction of the configuration \( x \) to the variables in \( s \) and \( t \). Finally, the marginalization of a potential \( p \) with domain \( d(p) = s \) to some subset \( t \subseteq s \) is obtained by summing up all equal configurations with respect to the retained variables in \( t \),

\[
p^{\downarrow t}(x) = \sum_{y \in \Omega_{s-t}} p(x, y).
\]

We thus obtain the (unnormalized) joint probability mass function \( p \) of a Bayesian network by combining all probability potentials \( \{p_1, \ldots, p_n\} \) assigned to its nodes,

\[
p = p_1 \otimes \ldots \otimes p_n.
\]

This corresponds to the objective function of Definition 2.2. Evaluating a Bayesian network then requires to compute \( p(X|e) \) for some variable \( X \in r \) and evidence \( e \subseteq r - \{X\} \). We have

\[
p(X|e) = \frac{p(X, e)}{p(e)} = \frac{p^{\downarrow e}(X)}{p^{\downarrow e}}
\]

which poses an inference problem with two queries \( e \cup \{X\} \) and \( e \).
2.2.2. Relational Algebra & Query Answering in Relational Databases

In the relational algebra, relations correspond to valuations with their attribute set as domain. Combination is represented by natural join \( r_1 \otimes r_2 = r_1 \bowtie r_2 \), and marginalization to some attribute set \( x \subseteq d(r) \) corresponds to the usual projection \( r^{ix} = \pi_x(r) \). The inference problem equals query answering in relational databases:

\[
(r_1 \otimes \cdots \otimes r_n)^{ix} = \pi_x(r_1 \bowtie \cdots \bowtie r_n).
\]

2.2.3. Propositional Logic & Satisfiability

Propositional formulae form a valuation algebra with conjunction as combination and existential quantification as marginalization. Marginalizing a formula to the empty set corresponds to the elimination of all propositions and either leads to the tautology or the contradiction. The inference problem therefore corresponds to the computation of satisfiability. However, there are some technical difficulties related to this simple conception. If we define the domain of a valuation by the set of propositions that occur in a formula, we may express the same propositional information by different valuations. We therefore consider valuations as equivalence classes of formulae in the valuation algebra of propositional logic. Also, we may define a valuation algebra of sets of models instead of formulae. The resulting system is isomorphic to the relational algebra with natural join and projection above.

2.2.4. Linear Equation Systems & Sparse Matrix Techniques

Sets of linear equations with values from a field form another valuation algebra with set union as combination and Gaussian elimination as projection. Here, we again have similar difficulties with respect to the definition of the domain of valuations. The actual valuations are therefore the affine spaces of sets of equations, but the computations still take place on the equation sets as representatives of their affine spaces. Applying the generic inference algorithms of the following section to this valuation algebra leads to standard sparse matrix techniques.

There are a great many of other formalisms that adopt the structure of a valuation algebra. Some of them have already been mentioned in the introduction of this article, others will be worked out in Section 5.1. Here, we next focus on the central question how inference problems can be solved efficiently and independent of the underlying valuation algebra.

3. Local Computation

Local computation algorithms operate on graphical structures called join trees. For its formal definition, we first remind that an undirected graph \((V,E)\) with vertices \(V\) and edges \(E\) is called a tree if it contains no cycles. Further, we consider labeled trees where every node \(v \in V\) contains a domain \(\lambda(v) \in D\) called node label.
**Definition 3.1.** A labeled tree \((V, E, \lambda)\) is called join tree if for two nodes \(i, j \in V\) and \(X \in \lambda(i) \cap \lambda(j)\) we have \(X \in \lambda(k)\) for all nodes \(k \in V\) on the unique path between \(i\) and \(j\).

![Diagram of trees](image)

**Figure 2.** The left-hand tree is clearly not a join tree since variable \(C\) is missing in the node with label \(\{D\}\). Adding this variable leads to the join tree shown on the right-hand side.

For initialization purposes, it is important that we can always adjoin a unique *identity element* \(e\) with \(d(e) = \emptyset\) to every valuation algebra \((\Phi, D)\). For arbitrary \(\phi \in \Phi\), we define \(\phi \otimes e = e \otimes \phi = \phi\) and \(e^0 = e\). Then, the system \((\Phi \cup \{e\}, D)\) still satisfies the valuation algebra axioms. Without loss of generality, we may therefore assume that every valuation algebra contains such an identity element.

In order to solve an inference problem, the join tree must cover all queries and knowledgebase factors. A node \(v \in V\) is said to cover a query \(x \in D\), if \(x \subseteq \lambda(v)\). Likewise, a node \(v \in V\) covers a valuation \(\phi \in \Phi\), if \(d(\phi) \subseteq \lambda(v)\). We therefore say that a join tree covers an inference problem, if a covering node can be found for each knowledgebase factor and query. This allows us to distribute knowledgebase factors over join tree nodes: At the beginning, all join tree nodes are initialized with the identity element \(e\) (because \(d(e) = \emptyset\), \(e\) is always covered). Then, every knowledgebase factor is combined to the content of a single covering node. Consequently, every node contains either the identity element, a single knowledgebase factor, or a combination of multiple knowledgebase factors. These prerequisites are common to all local computation schemes.

Exemplarily, we introduce the most general local computation architecture for the solution of multi-query inference problems called *Shenoy-Shafer architecture*\(^1\) that can be applied to all valuation algebras without any restriction. Besides the node content that results from the above factor distribution, we further assume that every node contains a storage to memorize incoming messages (valuations) during the local computation process. The Shenoy-Shafer architecture can then be specified by the following rules:

**R1:** A node \(i\) sends a message to its neighbor \(j \in ne(i)\) as soon as it has received all messages from its other neighbors.
R2: When node $i$ is ready to send a message to neighbor $j \in ne(i)$, it combines its initial node content $\psi_i$ with all messages from all other neighbors. The message is computed by marginalizing this result to the intersection of the result’s domain and the receiving neighbor’s node label:

$$
\mu_{i \rightarrow j} = \left( \psi_i \otimes \bigotimes_{k \in ne(i), j \neq k} \mu_{k \rightarrow i} \right)^{\downarrow \omega_{i \rightarrow j} \cap \lambda(j)} \quad (2)
$$

where

$$
\omega_{i \rightarrow j} = d(\psi_i) \cup \bigcup_{k \in ne(i), j \neq k} d(\mu_{k \rightarrow i}). \quad (3)
$$

The first rule implies that leaves can send their messages right away since they have only one neighbor. Further, the algorithm stops when every node has received all messages from its neighbors. If finally all messages are exchanged, the queries of the inference problem can be answered using the following theorem. 20

**Theorem 3.1.** At the end of the message-passing in the Shenoy-Shafer architecture, node $i$ can compute

$$
\phi^{\downarrow \lambda(i)} = \psi_i \otimes \bigotimes_{j \in ne(i)} \mu_{j \rightarrow i}. \quad (4)
$$

To sum it up, we answer the queries of an inference problem by the following procedure: We look for a covering join tree of the given inference problem and execute the message-passing of the Shenoy-Shafer architecture. For each query $x_i \in D$, we then choose a covering node $v \in V$ and compute $\phi^{\downarrow \lambda(v)}$ using Theorem 3.1. Then, as a consequence of the transitivity axiom, we obtain the query answer from one last marginalization

$$
\phi^{\downarrow x_i} = \left( \phi^{\downarrow \lambda(v)} \right)^{\downarrow x_i}.
$$

It is important to remark that the message-passing can always be organized in a particular scheduling: The leaves start sending their messages and this procedure continues from bottom-up until the root node has received all messages. We refer to this first half of the message-passing as *inward propagation* or *collect phase*. The second phase starts with the root node sending its messages to all neighbors and proceeds top-down until all leaves have received a message. This is called *outward propagation* or *distribute phase* and completes the message-passing. Figure 3 illustrates a complete run of the Shenoy-Shafer architecture.

During the run of the Shenoy-Shafer architecture, all computations take place within the nodes of the join tree that act as virtual processors and communicate through the exchange of messages. This is a common property of all local computation schemes and means that the domains of all intermediate factors are bounded by the largest node label in the join tree. This measure therefore determines the
\[\psi_4 = e\]

![Fig. 3. A join tree with node factors \(\psi_1\) to \(\psi_5\). The arrows indicate all messages that are sent during the Shenoy-Shafer architecture and the numbering refers to a possible scheduling. The node factors are obtained by assigning knowledgebase factors to join tree nodes. We assume that only \(\psi_4\) corresponds to the identity element and all other node factors are either single or combinations of multiple knowledgebase factors. Figure 4 shows the exchanged messages.](image)

<table>
<thead>
<tr>
<th>Message content:</th>
<th>Node domain:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. (\mu_1 \rightarrow 3) (\psi_1)</td>
<td>(w_{1 \rightarrow 3} = d(\psi_1))</td>
</tr>
<tr>
<td>2. (\mu_2 \rightarrow 3) (\psi_2)</td>
<td>(w_{2 \rightarrow 3} = d(\psi_2))</td>
</tr>
<tr>
<td>3. (\mu_3 \rightarrow 5) ((\psi_3 \otimes \mu_1 \rightarrow 3 \otimes \mu_2 \rightarrow 3))</td>
<td>(w_{3 \rightarrow 5} = d(\psi_3) \cup d(\mu_1 \rightarrow 3) \cup d(\mu_2 \rightarrow 3))</td>
</tr>
<tr>
<td>4. (\mu_4 \rightarrow 5) (\epsilon)</td>
<td>(w_{4 \rightarrow 5} = d(e) = \emptyset)</td>
</tr>
<tr>
<td>5. (\mu_5 \rightarrow 4) ((\psi_5 \otimes \mu_3 \rightarrow 5))</td>
<td>(w_{5 \rightarrow 4} = d(\psi_5) \cup d(\mu_3 \rightarrow 5))</td>
</tr>
<tr>
<td>6. (\mu_5 \rightarrow 3) ((\psi_5 \otimes \mu_4 \rightarrow 5))</td>
<td>(w_{5 \rightarrow 3} = d(\psi_5) \cup d(\mu_4 \rightarrow 5))</td>
</tr>
<tr>
<td>7. (\mu_3 \rightarrow 1) ((\psi_3 \otimes \mu_5 \rightarrow 3 \otimes \mu_2 \rightarrow 3))</td>
<td>(w_{3 \rightarrow 1} = d(\psi_3) \cup d(\mu_5 \rightarrow 3) \cup d(\mu_2 \rightarrow 3))</td>
</tr>
<tr>
<td>8. (\mu_3 \rightarrow 2) ((\psi_3 \otimes \mu_5 \rightarrow 3 \otimes \mu_1 \rightarrow 3))</td>
<td>(w_{3 \rightarrow 2} = d(\psi_3) \cup d(\mu_5 \rightarrow 3) \cup d(\mu_1 \rightarrow 3))</td>
</tr>
</tbody>
</table>

Fig. 4. A complete run of the Shenoy-Shafer architecture.

Alternatively to the Shenoy-Shafer architecture, other architectures for the solution of multi-query inference problems exist, that achieve better performance based on a more sophisticated caching policy for messages. These architectures require further mathematical properties that generally are not present in a valuation algebra. Although we omit a concrete discussion of these architectures, we nevertheless take a complexity of local computation. In other words, the smaller the node labels are, the more efficient is local computation. Regrettably, finding the join tree with smallest node labels is known to be NP-complete, but we have good heuristics that achieve reasonable execution time.
look at their mathematical requirements since this knowledge is important to access their implementation in the NENOK framework. Both, the Lauritzen-Spiegelhalter architecture and the Hugin architecture insist on valuation algebras with inverse elements, i.e. two valuations $\phi, \psi \in \Phi$ are called inverses, if

$$\phi \otimes \psi \otimes \phi = \phi \text{ and } \psi \otimes \phi \otimes \psi = \psi.$$  \hspace{1cm} (5)

If this additional property is present in a valuation algebra, then these architectures solve multi-query inference problems with a smaller number of messages than the above Shenoy-Shafer architecture. A further optimization is provided by the idempotent architecture which can only be applied in case of idempotent valuation algebras, i.e. if for every valuation $\phi \in \Phi$ and $t \subseteq d(\phi)$ we have

$$\phi \otimes \phi \downarrow t = \phi.$$  \hspace{1cm} (6)

By choosing $t = d(\phi)$, we conclude from the domain axiom that in an idempotent valuation algebra each element $\phi$ is the inverse of itself. This shows that idempotent valuation algebras are special cases of valuation algebras with inverse elements. Moreover, since each element is the inverse of itself, division becomes trivial and does not need to be executed during local computation. This effect is implemented in the idempotent architecture. There are also specialized local computation algorithms for the solution of single-query inference problems such as the fusion algorithm and the bucket-elimination scheme which accept arbitrary valuation algebras. Finally, specialized architectures for generic constructions (see Section 5) exist as for example the dynamic programming architectures for the identification of solution configurations in constraint systems or architectures for sparse matrix techniques. Generic implementations of all these methods are contained in the NENOK inference library, and the following section explains how they can be used for the solution of concrete inference problems.

4. Solving Inference Problems with NENOK

NENOK provides generic implementations of different local computation architectures for the solution of inference problems. To access these methods, the user must deliver an implementation of a formalism that satisfies the valuation algebra axioms and possibly some further properties as mentioned above. Typically for a Java framework, these algebraic conditions are mapped to an interface hierarchy that has to be instantiated. This part of NENOK is called its algebraic layer.

4.1. Implementing Valuation Algebras

Let us start out with the description of the smallest part of the algebraic layer whose instantiation is sufficient for the application of the Shenoy-Shafer architecture. This corresponds to the standard valuation algebra definition and consists of the components and relationships shown in the UML class diagram of Figure 5. The gray-framed interface does not belong to NENOK but is part of the standard Java
distribution. We first note that every component within this framework implements either directly or indirectly the `java.io.Serializable` marker interface which will later be important to perform distributed computations (Section 7.1).

Every user instance of the algebraic framework consists of two classes implementing the `nenok.va.Variable` and `nenok.va.Valuation` interface as foreshadowed in Figure 1. The `Variable` interface is a marker interface such that only a few work is related to its implementation. In short, the user should provide a suitable hash code method and string conversion by overwriting the corresponding methods from `java.lang.Object`. Based on these methods, NENOK offers in the `Domain` class a default realization of domains as hash sets of `Variable` objects. A more challenging task is the implementation of the `Valuation` interface which requires to specify four methods:

- **public Domain label()**: The labeling method returns the domain of a valuation object.
- **public Valuation combine(Valuation val)**: This method computes the combination of two valuation objects. It is essential that programmers keep the valuation algebra axioms at the back of their mind during the implementation process to assure properties like associativity or commutativity of combination.
- **public Valuation marginalize(Domain dom) throws VAException**: Marginalization of valuations is the third basic operation to implement. Again, the programmer must take attention to satisfy the mathematical properties. We remember for example that marginalization is only possible for variable sets that are subset of the current valuation’s domain. If this is not the case, the method signature allows to throw a `nenok.va.VAException`.
- **public int weight()**: The last method of this interface returns the current valuation’s weight as a
measure for the memory consumption of this object. This value will be used to measure and optimize the communication costs during distributed computing but has no effect on the correct computation of inference problems. More details will be given in Section 7.1.

Besides the signatures that are explicitly listed in the Valuation interface, we should also provide some methods to represent valuations on screen. NENOK offers an annotation class called nenok.va.Representor for this task which can be used to brand all output methods within the user implementation, on condition that the following requirements are satisfied: Output methods have empty parameter lists and return either a string object or any descendant of java.swing.JComponent for graphical output. NENOK then parses the user implementation and calls dynamically all annotated methods via the graphical user interface. This allows users to specify an arbitrary number of output methods with different return types for each valuation algebra instantiation.

Finally, the nenok.va.Identity class represents the identity element adjoined to the valuation algebra for initialization purposes during local computation. In general, NENOK users will not encounter this object except during the implementation of the combine method. Here, the user must ensure that the current valuation is returned whenever an identity element is given as argument. The two remaining components of Figure 5 named Separativity and Idempotency will be addressed subsequently.

To sum it up, a user definition of some formalism that satisfies the valuation algebra axioms requires to implement the two interfaces Variable and Valuation. If such an implementation is present, the user can define inference problems and call the various library methods for their solution.

4.2. Accessing the Local Computation Service Library

According to Definition 2.2, an inference problem consists of two components: a set of valuations called knowledgebase and a set of domains called query set. Once the user disposes of his proper valuation algebra implementation, a factory method contained in nenok.Knowledgebase enables the creation of knowledgebases:

- public static Knowledgebase create(Valuation[] vals, String name);
  Creates a knowledgebase from an array of valuations. To distinguish different knowledgebases, the second argument specifies its name.

Query sets on the other hand simply correspond to arrays of nenok.va.Domain object, i.e. the default implementation of a set of nenok.va.Variable objects. The corresponding constructor of the Domain class is:

- public Domain(Variable... vars);
  Creates a domain from an array of variables.
These two objects specify a multi-query inference problem which can be solved by one of the pre-implemented local computation architectures. It will later be shown that this process can be parameterized to a great extent. On the other hand, there is a default configuration of the local computation library to simplify the first contact with NENOK. This essentially executes the Shenoy-Shafer architecture presented in Section 3 which accepts all types of valuation algebras. The class nenok.LCFactory handles these configuration issues and serves as interface to the local computation library. Hence, the following code is sufficient to solve a multi-query inference problem using the Shenoy-Shafer architecture:

```java
LCFactory factory = new LCFactory();
JoinTree jt = factory.create(kb, queries);
jt.propagate();
```

The second code line builds a join tree that covers all factors of the knowledgebase `kb` and all elements from the query set `queries`. Its complete signature is:

- `public JoinTree create(Knowledgebase kb, Domain... queries) throws ConstrException;`

  Creates a join tree from a knowledgebase that covers the specified queries.

Here, NENOK throws a `nenok.constr.ConstrException` when queries are specified that contain variables which do not occur in the knowledgebase. Then, the third statement executes the local computation architecture that is currently specified, i.e. the Shenoy-Shafer architecture in the default configuration. Once the propagation method is terminated, we use the following method of the `nenok.lc.JoinTree` class to ask for the result of a query:

- `public Valuation answer(Domain query) throws LCException;`

  Returns the result of the inference problem for the given query.

In the case of the Shenoy-Shafer architecture, this method computes Theorem 3.1 and throws the indicated exception if we ask for the result of a query that cannot be computed, either because the propagation method has not yet been executed, or because the query is not covered by this join tree.

### 4.2.1. Inspecting Local Computation

We mentioned several times that NENOK has also been designed to serve as experimental platform. Indeed, a lot of measures are taken to inspect the computational processes behind the above method calls. To give an idea of the kind of information NENOK provides, we consider the following selection of methods contained in the `JoinTree` class:

- `public double getPropagationTime();`

  Returns the execution time in milliseconds of the join tree propagation.

- `public int countNodes();`

  Returns the number of join tree nodes.
• **public Domain getLargestDomain();**
  Returns the largest join tree node label as decisive complexity factor.

• **public Domain getAdapter();**
  Returns the adapter of this join tree.

Every join tree holds an `nenok.adapt.Adapter` object that is part of the `adapter` design pattern\(^9\) and used to abstract between local and remote computing (Section 7.1). In possession of this object, we can ask for information about the executed operations during the propagation such as the number of executed combinations for example:

• **public int getCombinations();**
  Returns the number of combinations executed during the propagation phase.

### 4.2.2. Changing the Local Computation Architecture

Solving an inference problem with the default configuration of the `LCFactory` object always executes the Shenoy-Shafer architecture. But we also mentioned the existence of other architecture that exploit additional valuation algebra properties for a more efficient message scheduling. Changing the default architecture to another local computation scheme contained in the NENOK inference library is simple, as shown in the following code snippet that runs the Lauritzen-Spiegelhalter architecture instead of the Shenoy-Shafer architecture.

```java
LCFactory factory = new LCFactory();
factory.setArchitecture(Architecture.Lauritzen_Spiegelhalter);
JoinTree jt = factory.create(kb, queries);
jt.propagate();
```

NENOK offers generic implementations of all well-established local computation architectures and makes them available through a Java enum type called `nenok.Architecture`. This includes among others the Shenoy-Shafer architecture and its improvement\(^10\) for binary join trees, the Lauritzen-Spiegelhalter and Hugin architectures, the idempotent architecture and many other more specialized architectures for specific families of valuation algebras (generic constructions). However, although changing the local computation architecture is straightforward, the user nevertheless has to ensure that its valuation algebra implementation satisfies the mathematical restrictions imposed by the chosen architecture. This leads over to the two yet undiscussed components of Figure 5. For the Lauritzen-Spiegelhalter and Hugin architecture we need valuation algebras with inverse elements. A corresponding method that returns the inverse of a given valuation is contained in the `nenok.va.Separativity`\(^3\) interface that extends `Valuation`:

• **public Separativity inverse();**
  Returns the inverse of the current valuation.

\(^{3}\)Separativity\(^3\) is the weakest algebraic condition to guarantee the existence of inverse elements.
Implementing Separativity is therefore sufficient to benefit from the Lauritzen-Spiegelhalter and Hugin architectures. More restrictive is the idempotent architecture that requires idempotent valuation algebras. As we have seen, each valuation is the inverse of itself in an idempotent valuation algebra which explains why \texttt{nennok.va.Idempotency} extends \texttt{Separativity} in the class diagram of Figure 5. But according to Equation (6) idempotency only refers to the behavior of combination. It is therefore a marker interface and the inherited method to compute inverses is trivially implemented by returning the current object.

4.2.3. Changing the Join Tree Construction Algorithm

We concluded in Section 3 that the complexity of local computation mainly depends on the quality of the underlying join tree, which in turn is determined by the size of the largest node label. Since building optimum join trees is generally impossible, we are naturally interested in using and comparing different join tree construction algorithms. NENOK provides this possibility based on the \texttt{command} pattern. The abstract class \texttt{nennok.constr.Algorithm} defines the signature of construction algorithms and allows users to contribute their own implementations. Then, the local computation factory can be persuaded to run a certain construction algorithm by the following method:

- \texttt{public void setConstructionAlgorithm(Algorithm algo);}

As a matter of course, NENOK already contains an assortment of ready-to-use construction algorithms. This includes for example the \texttt{One-Step-Look-Ahead Smallest Clique} (OSLA-SC) algorithm which is also part of the default configuration, or the \texttt{Static Sequence} algorithm that allows users to define the variable elimination sequence for the join tree construction process. Further, the \texttt{Algorithm} class also contains inspection methods such as:

- \texttt{public double getConstructionTime();}

With the explanations given in this section, the user can implement his own valuation algebra instance, specify inference problems and benefit from the existing local computation architectures for their solution. Moreover, it has been shown how to change the default architecture and join tree construction algorithm. This is suggestive of the various facilities of the NENOK framework. Before we present further theoretical concepts and their realization, we shortly introduce the graphical user interface and how it may provide assistance for the above tasks.

4.3. The NENOK Graphical User Interface

The class \texttt{gui.Viewer} contains a multiplicity of static methods to start the graphical user interface under different configurations. Let us choose the following exponent:
public static void display(Knowledgebase... kbs)

Displays the graphical user interface with a selection of knowledgebases.

This method displays the main window and enumerates the passed knowledgebases on the top of the control panel. For the screenshot of Figure 6, we assume a user implementation of probability potentials to build two knowledgebases named Dog\(^4\) and Studfarm\(^3\) that both model well-known examples of Bayesian networks. Opening the main window causes NENOK to directly analyse the given objects using its inspection instruments. We can gather from the control panel that the marked Studfarm knowledgebase contains 13 valuations over 12 variables. To grasp knowledgebases graphically, the Network button opens a dialog showing the valuation network\(^5\) that illustrates the internal structure of the marked knowledgebase by relating variables (circles) with valuations (squares).

A join tree can be constructed from the currently marked knowledgebase using the corresponding button in the control panel together with the drop-down list to specify the local computation architecture. In Figure 7, we constructed a join tree from the Dog knowledgebase running the Shenoy-Shafer architecture which is then added to the join tree list in the middle of the control panel. There, we also see that the join tree has 9 vertices and that its largest node contains 3 variables. It is important to note that this course of action produces a join tree that only covers the knowledgebase factors. Additional queries could be specified through the dialog that opens from the Queries button. Displaying join trees graphically is simply achieved using the Show button and the drop-down list aside to choose.
an appropriate display mode. Finally, we can run local computation by pressing the two buttons *Collect* and *Distribute* in the undermost area, where we also get the runtime information and the number of executed valuation algebra operations. It is important to understand that this user interface is itself a generic component that displays, inspects and processes knowledgebases of arbitrary content. Also, join tree construction and local computation are naturally independent of the concrete knowledgebase content, whereas the results of the computations are displayed using the output methods in the user instantiation (see Section 4.1). This gives users the possibility to work simultaneously with different types of valuation algebras and different local computation architectures within the graphical user interface.

The NENOK user interface offers many further functionalities to explore local computation and the involved graphical structures. But since this article focuses on the generic inference library, we now continue the theoretical considerations regarding inference by addressing an important question. Local computation methods solve inference tasks modeled by a formalism that satisfies the valuation algebra axioms. The practical impact of such a framework can naturally be increased by extending its application field which is synonymical to the identification of further valuation algebras. Clearly, the obvious approach for this undertaking is to verify the axioms for a given formalism. But far more promising are so-called *generic constructions*\(^4\) that identify whole families of connatural valuation algebras. The advantages are apparent: On the one hand, we cover a large number of formalisms with a single verification proof of the axiomatic system and also derive their algebraic properties.

![Fig. 7. Graphical representation of join trees in NENOK.](image-url)
in a single step. On the other hand, we detect yet unknown formalisms that adopt the valuation algebra structure. Although different generic constructions exist and are implemented in NENOK, we confine ourselves to semiring valuation algebras.\cite{4,5}

\begin{tabular}{|c|c|c|}
\hline
A & + & \times \\
\hline
\mathbb{R}_{\geq 0}, \mathbb{N} \cup \{0\} & + & \times \\
\mathbb{R} \cup \{-\infty, \infty\} & \text{max} & \text{min} \\
\{0,1\} & \lor & \land \\
\mathbb{N} \cup \{0, \infty\} & \text{min} & + \\
\mathbb{R} \cup \{-\infty\} & \text{max} & + \\
\mathcal{P}\{s_1, \ldots, s_n\} & \cup & \cap \\
\mathbb{N} & \text{lcm} & \text{gcd} \\
[0,1] & \text{min} & \text{max} \\
\hline
\end{tabular}

\text{Arithmetic Semiring} \\
\text{Bottleneck Semiring} \\
\text{Boolean Semiring} \\
\text{Tropical Semiring} \\
\text{Arctic Semiring} \\
\text{Distributive Lattice} \\
\text{T-Norm Semiring}

Fig. 8. A selection of commutative semiring examples. We remind that a \textit{t-norm} is a binary operation on the unit interval that is associative, commutative and non-decreasing in both arguments. \mathcal{P}(s) refers to the powerset of a finite set \textit{s}.

5. Semiring Valuation Algebras

Let us first recall that a tuple \((A, +, \times)\) with binary operations + and \times is called \textit{commutative semiring} if + and \times are associative, commutative and if \times distributes over +, i.e. if for \(a, b, c \in A\) we have

\[ a \times (b + c) = (a \times b) + (a \times c). \]

A selection of typical examples is contained in Figure 8. It will next be shown that every commutative semiring induces a valuation algebra by means of a simple construction that maps configurations to semiring values. We consider thereto a set of variables \(s \subseteq \mathcal{r}\) with finite frames and the corresponding set of configurations \(\Omega_s\) defined in Equation (1). A semiring valuation \(\phi\) with domain \(d(\phi) = s\) is then defined by the mapping

\[ \phi : \Omega_s \rightarrow A. \] (7)

Combination of semiring valuations reduces to semiring multiplication: If \(\phi\) and \(\psi\) are semiring valuations with domain \(d(\phi) = s\) and \(d(\psi) = t\), we compute their combination for \(x \in \Omega_s \cup t\) by

\[ (\phi \otimes \psi)(x) = \phi(x^s) \times \psi(x^t). \] (8)

Marginalization on the other hand uses semiring addition. For a semiring valuation \(\phi\) with domain \(s, t \subseteq s\) and \(x \in \Omega_t\), we define

\[ \phi^{lt}(x) = \sum_{y \in \Omega_{s-t}} \phi(x,y). \] (9)
This definition is well-defined due to the associativity and commutativity of semiring addition, and the following theorem states what we promised beforehand:

**Theorem 5.1.** A system of semiring valuations with labeling, combination and marginalization as defined above, satisfies the axioms of a valuation algebra.

We therefore obtain a new valuation algebra for every commutative semiring, and may apply the usual local computation architectures to solve corresponding inference problems. In this context, we should also analyse the requirements for a semiring to guarantee specific properties in the induced valuation algebra, since these properties are important for the application of other multi-query local computation algorithms than the Shenoy-Shafer architecture. In particular, we brought up in Section 3 that the existence of inverse valuations makes the application of more efficient local computation architectures possible. It is easy to see that the definition $\psi(x) = \phi(x)^{-1}$ satisfies Equation (5) which implies that inverse semiring elements are sufficient to obtain inverse valuations. Idempotency on the other hand requires even more structure from the underlying semiring. However, we pass on the definition of these properties for the sake of simplicity and turn towards the exemplification of semiring valuation algebras. Since commutative semirings are very large in number, they also produce a multiplicity of valuation algebras. To get a more concrete picture, we briefly delineate some valuation algebras that are obtained from the semirings in Figure 8.

### 5.1. Examples & Applications II

Obviously, the valuation algebra induced by the arithmetic semiring directly corresponds to the formalism of probability potentials introduced in Section 2.2.1 to express (conditional) probability mass functions in a Bayesian network. Alternatively, we may compute the value of most or least probable explanations by considering valuations induced by the t-norm semirings $\langle [0,1], \max, \times \rangle$ or $\langle [0,1], \min, \times \rangle$ respectively. If we take the Boolean semiring, we obtain the formalism of (crisp) constraints or indicator functions. In this case, the inference problem with empty query models the constraint satisfiability problem:

$$\phi^\land(\emptyset) = (\phi_1 \otimes \cdots \otimes \phi_n)^\land(\emptyset) = \max\{\phi(x) : x \in \Omega_d(\phi)\}. \quad (10)$$

Here, $\emptyset$ denotes by convention the single configuration for the empty variable set. Similarly, we obtain further constraint systems from other semirings as for example weighted constraints from the tropical semiring $\langle \mathbb{N}, \min, +_\mathbb{N} \rangle$, or set-based constraints from the powerset lattice $\langle \mathcal{P}\{s_1, \ldots, s_n\}, \cup, \cap \rangle$. Also, we hit upon the formalism of possibility potentials by considering valuations induced by different t-norm semirings. In addition to the rediscovery of established formalisms for knowledge representation, the semiring view also identifies yet unknown valuation algebras. Interestingly, we directly dispose of efficient algorithms to solve the corresponding inference problems, although the exact semantical meaning of the latter
is still unknown. This is for example the case if we consider the semiring valuation algebra obtained from the distributive lattice \((\mathbb{N}, \text{lcm}, \gcd)\).

Another advantage of the semiring perspective is the ability to identify inference problems directly without first studying the underlying valuation algebra. To understand this, we first rewrite the inference problem in the language of semiring valuation algebras:

\[
\phi^{1,s}(x) = (\phi_1 \otimes \cdots \otimes \phi_n)^{1,s}(x) = \sum_{d(\phi) = s} \left( \phi_1(x^{1,d(\phi_1)}) \times \cdots \times \phi_n(x^{1,d(\phi_n)}) \right). \tag{11}
\]

Now, consider for example the discrete Fourier transform\(^8\) for \(N \in \mathbb{N}\) and a function \(f : \mathbb{Z}_N \to \mathbb{C}\) given as:

\[
F(y) = \sum_{x=0}^{N-1} f(x)e^{-\frac{2\pi xy}{N}}.
\]

Let us next take \(N = 2^m\) and write \(x\) and \(y\) in binary representation:

\[
x = \sum_{k=0}^{m-1} x_k 2^k \\
y = \sum_{l=0}^{m-1} y_l 2^l
\]

with \(x_k, y_l \in \{0, 1\}\). This corresponds to an encoding of \(x\) into binary vectors \((x_0, \ldots, x_{m-1})\) and \(y\) into \((y_0, \ldots, y_{m-1})\). The product \(xy\) can then be written as

\[
xy = \sum_{0 \leq k, l < m} x_k y_l 2^{k+l},
\]

and we obtain for the discrete Fourier transform

\[
F(y_0, \ldots, y_{m-1}) = \sum_{x_0, \ldots, x_{m-1}} f(x_0, \ldots, x_{m-1}) \prod_{0 \leq k, l < m} e^{-\frac{i\pi x_k y_l}{2^{m-1-k-l}}}.
\]

The second equality holds because the exponential factors become unity if \(k+l \geq m\). We can easily convince ourselves by comparison with Equation (11) that this formula corresponds to an inference problem over the complex, arithmetic semiring \((\mathbb{C}, +_\mathbb{C}, \times_\mathbb{C})\). Moreover, if we apply local computation upon such an inference problem, we indeed reach the complexity of the fast Fourier transform. Similarly, we also detect inference problems behind other discrete transforms such as the discrete cosine transform or the Hadamard transform.\(^4,8\)

### 5.2. Defining Semiring Valuations in NENOK

The algebraic layer of the NENOK framework allows to define semiring valuations directly. Instead of delegating the (perhaps) challenging task of implementing combination and marginalization to the user, NENOK builds semiring valuations according to Equation (7) and provides generic implementations of both operations.
Thus, it only requires a user implementation of semiring addition and multiplication which is always a good deal simpler than implementing the combination and marginalization rule of valuation algebras. The core components of the algebraic layer have been introduced in the class diagram of Figure 5. This framework is extended for semiring valuation algebras in Figure 9.

![Diagram](image)

Fig. 9. Extension of the algebraic layer to define semiring valuation algebras.

Since variable frames are always finite when dealing with semiring valuations, NENOK proposes a default implementation of the `Variable` interface named `nenok.va.FiniteVariable` with the following constructor:

- `public FiniteVariable(String name, String[] frame);`

  Constructs a `Variable` instance from the variable's name and its possible values.

Hence, variables are simply identified by a string argument which also affects the internal realization of the equality check and output method. No user work concerning the implementation of variables is required here.

The second component needed to derive semiring valuations is the implementation of the semiring itself. For that purpose, NENOK defines the generic `nenok.semiring.Semiring<E>` interface which appoints the two standard semiring operations and a string conversion:

- `public E add(E val1, E val2);`
  
  Computes the addition of two semiring values.
- `public E multiply(E val1, E val2);`
  
  Computes the multiplication of two semiring value.
- `public String valueToString(E val);`
  
  Transforms a semiring value into a string.

Again, the user must pay attention to respect associativity and commutativity during the implementation of these operations. Here, the practical advantages of generic constructions become apparent: Instead of implementing combination and marginalization of probability potentials for example, it is in fact sufficient to deliver multiplication and addition of real numbers. The construction of the semiring
valuation is then accomplished by the abstract class `nenok.sva.SRValuation<E>` that implements `Valuation` with labeling, combination and marginalization for semiring valuations as defined in Section 5.

Remember, semiring valuations are simple mappings from configurations to semiring values. This is also reflected in the default constructor of `SRValuation<E>`:

- **public SRValuation(FiniteVariable[] vars, Semiring<E> semiring, List<E> values);**
  
  Builds a semiring valuation from variables and semiring values.

This constructor produces the standard enumeration of the configuration set from the given array of finite variables and assigns the semiring values in the given order to these configurations. For instance, the third value of the `values` list will be assigned to the third configuration in the standard enumeration. Doing so, the configuration set does not need to be produced explicitly, which naturally provides a measurable increase of performance. If the number of semiring values does not match with the computed size of the configuration set, an `IllegalArgumentException` will be thrown. In addition, this constructor requires a `Semiring<E>` object for the processing of the semiring values in the pre-implemented combination and marginalization rule.

The only abstract component of `SRValuation<E>` is the following factory method:

- **public abstract SRValuation create(FiniteVariable[] vars, Semiring<E> semiring, List<E> values);**

  The abstract factory method within `SRValuation<E>`.

Its importance and functionality can best be understood by reflecting on a user implementation of a certain semiring valuation algebra: The user starts writing a class that extends `SRValuation<E>` and that contains to the programmer’s pride a very sophisticated output method called `output()`. If then a combination of two such instances is executed, the result will be of type `SRValuation<E>` because the combination was delegated to the superclass `SRValuation<E>`. Consequently, it is no longer possible to apply `output()` on the resulting instance. This problem is addressed by the above factory method. Instead of directly calling a constructor, all pre-implemented methods within `SRValuation<E>` that return new instances, build them by executing the above factory method. Then, if the user delegates `create` to its own class constructor, the above problem ends in smoke.

The `Separativity` interface of the algebraic layer which is used to endue valuation algebras with inverse elements must also be implemented to obtain inverse semiring valuations. However, we have also seen that the existence of inverse semiring elements is required for inverse semiring valuations which then allows to give a generic pre-implementation of the `inverse` method from `Separativity`. This property is mirrored in Figure 9 by the interface `Dividability<E>` that extends `Semiring<E>` with a method to compute inverse semiring elements:
• public E inverse(E val);
   Returns the inverse of the given semiring value.

NENOK does not provide a specialized subclass of SRValuation<E> that takes
semiring elements of type Dividability<E> and contains a default implementation
of Separativity. Although reasonable at first glance, representing valuation algebra
properties in the shape of abstract classes makes the equipment of valuation algebras
with different properties impossible since Java does not allow multiple inheritance.
Instead, all properties are mirrored by corresponding interfaces which (wherever
applicable) contain inner delegator\(^9\) classes with generic pre-implementations. These
inner classes are uniformly named Implementor. Hence, it is sufficient for a descendant
of SRValuation<E> to implement Separativity and to delegate its inverse method to
the delegator class of Separativity. This corresponds to the following code:

```java
public Separativity inverse() {
    return Separativity.Implementor.getInstance().inverse(this);
}
```

To sum it up, inverse semiring valuations are obtained by implementing
Dividability<E> and adding the above code to the valuation algebra implementation
that extends SRValuation<E> and implements Separativity.

6. Existing NENOK Instantiations
The main mission of the NENOK software project is to provide a library of generic
inference tools based on the valuation algebra framework. It is therefore not its am-
bition to provide as many pre-implemented valuation algebra instances as possible.
Nevertheless, the current distribution contains showcase implementations of prob-
bility potentials, indicator functions and weighted constraints which are all based
on the semiring framework. In addition, there are several student projects that
contributed implementations of other valuation algebra instances as for example
distance functions, Gaussian potentials, assumption based reasoning, propositional
logic or the relational algebra. Links to these projects can be found on the NENOK
website. Finally, (Ref. 23) also implemented further generic local computation archi-
tectures including updating procedures that can be used via the plug-in mechanism
of the NENOK core system.

7. Remote Computing
The idea of a knowledgebase as basis for inference does not provide any evidence
about the (geographic) origin of its factors. But this is admittedly a very interest-
ing question since the interpretation of valuations as pieces of information suggests
that these factors may come from different sources. Additionally, local computation
algorithms are generally regarded as distributed algorithms where join tree nodes
act as virtual processors\(^1\) that store valuations within their private memory space
and communicate through the exchange of messages. Altogether, this motivated the elaboration and implementation of a communication infrastructure, upon which all local computation methods contained in the generic inference library are implemented as real distributed algorithms to compute inference from factors that reside on remote processors. To formalize this new situation, we consider the concept of a distributed knowledgebase:

Definition 7.1. A distributed knowledgebase is a knowledgebase \( \{\phi_1, \ldots, \phi_n\} \) together with an assignment mapping \( \chi : \{\phi_1, \ldots, \phi_n\} \rightarrow P \) determining the host processor of each valuation \( \phi_i \) with respect to a given processor set \( P \).

As shown in Figure 10, it is always possible for a given knowledgebase to find a covering join tree where every node contains either the identity element or a single knowledgebase factor. Here, combinations of multiple factors within a single node are explicitly undesired. Doing so, we may assign to every node that holds a knowledgebase factor the host processor of the latter and therefore obtain a partial processor distribution over join tree nodes. The join tree now assumes the role of an overlay network for the underlying physical processor infrastructure. This new perception of join trees is illustrated in Figure 11.

Efficient communication is of prime importance in a distributed system. To this end, we introduce a measure to express the costs of transmitting a message between remote processors. These costs naturally depend on the actual message content which brings us to the concept of a valuation’s weight. Further, we recall that the efficiency of local computation is justified by the fact that the domains of all intermediate factors are bounded by the join tree node labels. It is therefore reasonable to assume that a valuation’s weight shrinks under marginalization since otherwise the application of local computation would hardly be a runtime improvement.

Definition 7.2. Let \((\Phi, D)\) be a valuation algebra. A function \( \omega : \Phi \rightarrow \mathbb{N}_0 \) is called weight function if \( \omega(\phi) \geq \omega(\phi \upharpoonright x) \) for all \( \phi \in \Phi \) and \( x \subseteq d(\phi) \). Without loss of generality, we define \( \omega(\epsilon) = 0 \).
The communication costs $c_{\phi}(i, j)$ of sending some valuation $\phi \in \Phi \cup \{e\}$ from processor $i$ to processor $j$ are now defined by

$$c_{\phi}(i, j) = \omega(\phi) \cdot d_{i,j},$$

(12)

where $d_{i,j}$ denotes the distance between the processors $i, j \in P$. Note that we do not necessarily refer to the geographic distance between processors. Instead, we may consider the channel capacity, bandwidth, or another network related measure that satisfies the following properties: $c_{\phi}(i, j) \geq 0$, $c_{\phi}(i, j) = c_{\phi}(j, i)$ and $c_{\phi}(i, j) = 0$ if $i = j$. The third property states that communication costs are negligible if no network activities occur.

Above, we identified join tree nodes with the processors that store their factors. This ensures that no antecedent communication costs are incurred. From the property of a weight function, we further conclude that marginalizing a message decreases communication costs. It is therefore more efficient to transmit local computation messages instead of the original knowledgebase factors. However, before local computation can be performed on this remote computing environment, we need to determine the processors of the remaining join tree nodes that hold the identity element. Using the notation of Section 3 we define $\Psi = \{i \in V : \psi_i = e\}$ to be the set of join tree nodes that originally contain an identity element. Thus, we have $|P|^{|\Psi|}$ different possibilities to complete the partial processor distribution. We will see in Section 8 how a suitable choice with respect to efficient communication can be found. For the moment, we assume an arbitrary completion $\xi : V \rightarrow P$ of the processor assignment that satisfies $\xi(i) = \chi(\psi_i)$ if $i \notin \Psi$. 

![Diagram](image-url)
Using this model, we now compute the total communication costs of the message passing in the Shenoy-Shafer architecture with respect to a specific processor distribution:

\[
T_{\xi} = \sum_{i=1}^{m} \sum_{j \in ne(i)} c_{\mu_{i \to j}}(\xi(i), \xi(j)) = \sum_{i=1}^{m} \sum_{j \in ne(i)} \omega(\mu_{i \to j}) \cdot d_{\xi(i), \xi(j)}.
\]  

(13)

7.1. Remote Computing in NENOK

NENOK offers a complete service infrastructure for the processing of remote valuations which is realized in a highly transparent manner: If the user delivers a knowledgebase with only local valuations (using the Knowledgebase factory method at the beginning of Section 4.2), the local computation factory will build join trees that perform all computations locally without any network interaction. On the other hand, if we deliver the NENOK counterpart of a distributed knowledgebase, local computation will also be executed as a real distributed algorithm. However, performing remote computations requires at first to set up a communication infrastructure that offers the basic communication facilities. NENOK uses the Jini framework for that purpose and implements processors as independent Jini services. Thus, we first skim through the needed configuration work to start Jini services in general.

Foremost, every Jini environment contains a lookup service that acts as a broker between clients and services. At start time, Jini services contact the lookup service to announce their presence and functionality. Clients in turn locate services by communicating their needs to the lookup service which then gives access to an appropriate service. The second important component in a Jini environment is a webserver. Jini extracts its flexibility from dynamic class loading that allows to serialize and transfer objects to a remote virtual machine where the objects are brought back to life again. This however requires that also the object’s code is transmitted since the latter is not necessarily present on the remote client. Therefore, a webserver is contained in every Jini system that handles this code exchange. To sum it up, NENOK requires a running Jini system that consists of a webserver and a lookup service. The NENOK distribution contains start scripts for both services which only require some small adjustments. (Ref. 4) describes this preparation work in more detail and also addresses further configuration work for network and security issues. Assuming such a Jini system, we next focus on the services that are directly related to NENOK. On the one hand, processors exchange and process remote valuations. As subsequently explained, we can start an arbitrary number of processors on any network host to build up the processor networks of the foregoing section. This is in contrast to the second NENOK service called knowledgebase registry which will be used to define distributed knowledgebases. The knowledgebase registry is a central service in the NENOK environment and should therefore be started just after the lookup service using the available start script.
7.2. Setting up a Processor Network

The class nenok.Services is the main interface to the Jini communication infrastructure and provides static methods to administrate NENOK services. Primarily, this includes the setup of a processor network that took center stage in the foregoing section. The following method can be used to start an arbitrary number of processors either on a single host to simulate distributed computing or on different computers:

- public static Processor startProcessor();
  Starts a new processor on the current host machine.

Processors are identified by globally unique identifiers named net.jini.id.Uuid which users can request from nenok.net.processors.Processor instances:

- public Uuid getPID() throws RemoteException;
  Returns the identifier of the current processor.

It is pointed out that most methods related to distributed computing throw RemoteException to indicate network problems. Using the requested identifier, the Services class can locate processors running anywhere in the network by contacting the Jini lookup service.

- public static Processor findProcessor(Uuid pid, String... urls);
  Finds the processor with the given identifier either by unicast or, if the second argument is omitted, by multicast requests.

The last important administrative method concerns the termination of processors:

- public static boolean destroyProcessor(Uuid pid);
  Destroys the processor with the given identifier and communicates the successful termination with a boolean value.

7.3. Publishing Valuation Objects

Processors are equipped with a public memory space where users can store and retrieve Valuation objects using the corresponding methods of the Processor interface:

- public Locator store(Valuation val) throws RemoteException;
  Stores a valuation in the processor’s public memory space.

The returned nenok.net.Locator object figures as cloakroom ticket that allows to retrieve the valuation at a later date. Locators essentially contain two identifiers: one for the processor and a second for the valuation within the processor’s memory. Thus, an application can store a valuation on a certain processor, send the returned locator to another application which then retrieves the valuation using the following method of the Locator class:

- public Valuation retrieve() throws RemoteException;
  Retrieves the valuation identified by the current locator.
7.4. Distributed Knowledgebases

Following the above description of exchanging valuations via their locators, NENOK implements distributed knowledgebases as collections of Locator objects. However, a key requirement is that different users can contribute valuations to a shared knowledgebase. For that purpose, NENOK provides the central knowledgebase registry service that has been mentioned in the introduction of this section. Getting in touch with the previously started knowledgebase registry is again possible through the Services class:

- **public Registry findRegistry(String... urls) throws MalformedURLException;**

  Finds the knowledgebase registry either by unicast or, if the argument is omitted, by multicast requests.

The returned nenok.net.registry.Registry object can now be used for the administration of distributed knowledgebases. For example, the following method creates a new knowledgebase on the registry server:

- **public boolean createKnowledgebase(String name) throws RemoteException;**

  Creates a new knowledgebase with the given name as identifier.

Contributing valuations to a shared knowledgebase is possible by sending their locators to the registry service:

- **public boolean add(String name, Locator loc) throws RemoteException;**

  Adds a locator to the knowledgebase with the given name.

Evidently, there are also methods to delete knowledgebases or to remove single locators from a certain knowledgebase, but more important is the method that allows to download knowledgebases from the registry service:

- **public Knowledgebase getKnowledgebase(String name) throws RemoteException;**

  Downloads the knowledgebase to the current host.

As one can see, we are now in possession of a Knowledgebase object. In contrast to Section 4.2, this knowledgebase contains locators instead of valuations directly, but this information is not relevant for the application of local computation. In fact, all knowledgebases can be processed with the three lines of code shown in Section 4.2 thanks to the already mentioned Adapter class that abstracts between local and remote computing. More concretely, all local computation architectures execute valuation algebra operations through this object which either performs the computations itself or, if the computations involve locators pointing to remote valuations, forwards the task to the corresponding processor. Thus, it follows that the local computation library does not need to distinguish between local and distributed knowledgebases.
We introduced in Section 4.1 the signature of the Valuation interface to define the weight function of a valuation algebra. This method is successively called during local computation to measure the total communication costs according to Formula (13), and its value is returned by the following method of the JoinTree class:

- **public double getCommunicationCosts();**
  - The total communication costs caused by the message passing.

Naturally, we may also treat distributed knowledgebases via the graphical user interface. The View menu of the main window adapts the topmost element of the control panel for remote computing as shown in Figure 12. Using the Search button we may scan the network for a knowledgebase registry service. Here, such a service has been found on host 134.21.73.91 that contains the distributed knowledgebase Earthquake. Join tree construction works as supplied before with the effect that communication costs are now displayed in the lowermost part of the control panel. We further point out that NENOK represents the assignment of processors to join tree nodes in the right-hand dialog with different colors.

![Fig. 12. Distributed computing via the graphical user interface.](image)
8. Optimizing Communication

Formula (13) measures the total communication costs of any local computation method with respect to a given completion $\xi$ of the original processor distribution. However, we have also seen in Section 7 that $|P|^9$ different completions exist which naturally imply different communication costs. Thus, we next address ourselves to the task of determining the completion with minimum communication costs. The principal problem here is that all messages $\mu_{i \rightarrow j}$ must be known in order to compute their weight $\omega(\mu_{i \rightarrow j})$. Therefore, this formula can only be used to compute communication costs after the execution of local computation which makes a predictions of communication costs to rate a certain completion impossible. This is the reason why NENOK contents itself with a simple greedy approach to determine the final completion in its most general setting. Nevertheless, communication can be optimized if we assume a further property of the valuation algebra called weight predictability:

Definition 8.1. Let $(\Phi, D)$ be a valuation algebra and $\omega$ its weight function. $\omega$ is called weight predictor if there exists a computable function $f : D \rightarrow N_0$ such that for all $\phi \in \Phi$ we have $\omega(\phi) = f(d(\phi))$.

Thus, a valuation algebra is said to be weight predictable if the weight of a valuation can be computed from its domain. To give a concrete example, we remark that all semiring valuations are weight predictable with the following weight predictor:

$$\omega(\phi) = \prod_{X \in d(\phi)} |\Omega_X|.$$

On the other hand, we do not have weight predictability if we measure the weight of a relation (see Section 2.2.2) by its number of tuples.

An important consequence of weight predictability is that if $d(\psi) \subseteq d(\phi)$ we have $\omega(\psi) \leq \omega(\phi)$. Using this property and the domain of a message given in Equation (2), we find an upper bound for the total communication costs:

$$T_\xi = \sum_{i=1}^{m} \sum_{j \in ne(i)} \omega(\mu_{i \rightarrow j}) \cdot d_{\xi(i), \xi(j)} = \sum_{i=1}^{m} \sum_{j \in ne(i)} f(\omega_{i \rightarrow j} \cap \lambda(j)) \cdot d_{\xi(i), \xi(j)} \leq \sum_{i=1}^{m} \sum_{j \in ne(i)} f(\lambda(i) \cap \lambda(j)) \cdot d_{\xi(i), \xi(j)} = 2 \sum_{i=1}^{m-1} f(\lambda(i) \cap \lambda(ch(i))) \cdot d_{\xi(i), \xi(ch(i))}.$$

This formula applies only to the join tree node labels and is independent of the actual node content which allows us to estimate communication costs without first executing local computation. To ensure minimum communication costs, we are therefore looking for the processor assignment $\xi$ that minimizes

$$\sum_{i=1}^{m-1} f(\lambda(i) \cap \lambda(ch(i))) \cdot d_{\xi(i), \xi(ch(i))}.$$
It has been shown that this problem corresponds to the so-called multiway cut problem which is NP-complete in general. Fortunately, there exists a low polynomial algorithm to solve multiway cut problems on trees which can easily be applied to the problem at hand. To sum it up, we may find the best processor assignment with respect to communication costs provided that the valuation algebra is weight predictable.

8.1. Optimized Communication in NENOK

Naturally, NENOK supplies an extension of the algebraic layer to support weight predictable valuation algebras and ensures minimum communication costs of local computation with distributed knowledgebases through the implementation of the polynomial multiway cut algorithm for the determination of the optimum processor distribution. Since all valuation algebra properties have been represented by appropriate Java types, this should also be the case for weight predictability. However, the concrete realization is perhaps more cumbersome than one imagines due to the requirement to compute a valuation’s weight even before this object actually exists. In Java terms, this demands for a static implementation. But every valuation algebra possesses its own weight predictor which in turn contradicts the design based on a static method. This reasoning shall give an impression of the task’s complexity. In order to meet the requirements for the most part, the delegator technique was again applied. Weight predictable valuation algebras are mirrored in NENOK by the nenok.va.Predictability interface that contains the following method signature:

- public Predictor predictor();  
  Returns the weight predictor of this valuation algebra.

The nenok.va.Predictor object returned by the above method is the actual weight predictor of the current valuation algebra. The corresponding interface contains naturally the method to compute a valuation’s weight from a given domain:

- public int predict(Domain dom);  
  Returns the weight of all valuations that have the given domain.

The design’s only drawback is that we must dispose of at least one valuation object in order to ask for the weight predictor. But then, we can compute the weight of an arbitrary non-existing valuation using the Predictor object. The application of the delegator design strategy may seem complicated, but a closer inspection proves its value. First, each valuation algebra can possess its own weight predictor. Second, we can demand the predictor of an arbitrary object in order to compute the weight of any other instance of the same valuation algebra. Finally, weight predictable valuation algebras are realized by a new type which harmonizes with the implementation of other properties.
Provided that a valuation algebra is weight predictable, we can give a pre-implementation of the generic weight method inherited from the Valuation interface. It should not be surprising that Predictability contains an implementor class which offers this functionality via the following delegation:

```java
public int weight() {
    return Predictability.Implementor.getInstance().weight(this);
}
```

To sum it up, NENOK analyses distributed knowledgebases for the property of weight predictability. If the underlying valuation algebra shows this property, cheap communication is ensured through the determination of the optimum processor distribution. Otherwise, NENOK simply contents itself with a greedy approach for the processor distribution.

9. The Parser Interface

The construction of a covering join tree for a given knowledgebase is the first essential step to compute inference. In the course of this article, we always started from a Knowledgebase object which can either be created directly with the factory method given in Section 4.2 or downloaded from the registry service. In both cases, the appropriate method of the local computation factory was called to obtain a covering join tree. Alternatively, join trees can also be constructed from a file’s content using the NENOK parser interface. This is especially useful to deal with knowledgebases that include a very large number of valuations or to reuse examples that have originally been created for other inference libraries. To do so, we first implement the parser interface called nenok.parser.KB_Parser with the following signature:

- **public ResultSet parse(File file) throws ParserException;**
  
  Parses the given file into a result set.

The returned nenok.parser.ResultSet object is just a wrapper class for the parsed knowledgebase and query set. The LCFactory class then contains the following method to build a covering join tree from a file’s content:

- **public JoinTree create(File file, KB_Parser parser) throws ParserException;**
  
  Parses the given file and creates a join tree from the file content.

NENOK already includes a partial implementations of KB_Parser for XML based files and also provides default implementations for different file formats related to Bayesian networks. In addition, the local computation factory also enables to process files that specify the shape of a join tree directly. This can for example be used to construct several equal join trees running different local computation architectures for comparison. Building them with the usual construction algorithms is generally not possible since the latter are based on heuristics that possibly include random components.
10. Conclusion

The valuation algebra framework has proved its value to serve as unifying theory for knowledge representation systems. This allows us to specify efficient inference algorithms for the processing of arbitrary valuation algebras and further permits to study problems independently of the underlying inference formalism. However, the obvious potential of this theory to write generic software libraries has so far been unrecognized which coerced programmers into rewriting the same algorithm for every different application. This indeed corresponds to the unreasonable image of rewriting a sorting procedure for every task that requires to sort a list with different content. NENOK puts things right and offers for the first time a purely generic inference library based on an abstract representation of the valuation algebra framework. Every user can instantiate this framework for the current formalism at hand and directly access the large library of sophisticated inference tools. This turns the often complicated task of computing inference into three simple method calls. It is in the nature of things that knowledge and information are distributed resources which also requires distributed algorithms for their processing. In addition, we have seen that inference algorithms are traditionally described as message-passing schemes. Both statements motivated the realization of NENOK upon a distributed computing environment with a special focus on transparency between local and remote computations. Finally, the NENOK project also comprises a graphical user interface to inspect the inference process and the involved graphical structures which may serve either as educational tool or as fast prototyping architecture. It is a central aspect that every component of NENOK’s configuration (local computation architecture, construction algorithm, etc) can be exchanged which also highlights its buildup as experimental plug-in system.

References