Department of Informatics - University of Fribourg - Switzerland

A generic Architecture for local Computation

Marc Pouly
marc.pouly@unifr.ch

Master Thesis in Computer Science

Supervisors: Prof J. Kohlas, Dr. N. Lehmann

August 30, 2004

Département d'Informatique
Université de Fribourg
Rue Faucigny 2, CH-1700 Fribourg

Tel: +41 26 300 83 22
Fax: +41 26 300 97 26
http://www.unifr.ch/informatics

Contents

1 Valuation Algebras **1**
- 1.1 A Formal Definition . . . 1
- 1.2 Probability Potentials . . . 5
- 1.3 A Medical Example . . . 7

2 Local Computation **9**
- 2.1 Fusion Algorithm . . . 10
- 2.2 Graphical Representation of the Fusion Algorithm . . . 11
- 2.3 Collect Algorithm . . . 15
- 2.4 Computing Multiple Marginals . . . 17
- 2.5 Scaled Shenoy-Shafer Architecture . . . 19

3 Implementation Strategies **21**
- 3.1 Join Tree Construction . . . 21
- 3.2 Answering queries . . . 29

4 A Generic Computation Architecture in JAVA **30**
- 4.1 Generic Representation of Valuation Algebras . . . 30
- 4.2 Framework for Local Computation . . . 33
- 4.3 Realizing the Shenoy-Shafer Architecture . . . 38
- 4.4 Join Tree Viewer . . . 42

5 A first Instantiation **43**
- 5.1 Implementing Probability Potentials . . . 43
- 5.2 Experimenting with the Medical Example . . . 46

6 Outlook & Conclusions **50**

A Program Structure **52**

B The Big Picture **54**

C Disc Content **55**

List of Figures

1.1 Causal network of a fictitious medical example. 8

2.1 Building a graph by applying the fusion algorithm. 13
2.2 Continuation of the build process. 14
2.3 Distributing the initial valuations on the join tree. 15
2.4 Changing the root of a join tree. 18
2.5 A schematic representation of mailboxes. 18
2.6 A possible run of the message passing process. 19

3.1 Union and projection node. 23
3.2 Join trees created with different elimination sequences. 24
3.3 A hypergraph with the corresponding cost table. 24
3.4 Illustration of a Variable-Valuation-Link-List (VVLL). 26
3.5 Updated VVLL after eliminating variable a. 26
3.6 Heap data structure. 27
3.7 Removing a heap's root element. 28
3.8 Updating a heap when an element's key has changed. 28

4.1 The interface hierarchy to represent valuation algebras. 34
4.2 Illustration of the computing overhead with n-ary join trees. . . 38
4.3 No computing overhead for binary join trees. 39
4.4 Transforming an n-ary node to a binary tree. 40
4.5 A binary node with attached mailboxes. 41

5.1 Query answers of the medical example. 49
5.2 Output of the "TreeViewer" tool for the medical example. 49

B.1 A class diagram of the whole architecture. 54

Abstract

Valuation algebras are known to present a common roof over various applications out of different domains of research. In combination with the power of local computation it offers a possibility to solve inference problems in an efficient way. Implementing concretely these concepts by instantiating a valuation algebra means loosing the high level of abstraction. This paper presents a generic realization of local computation based on an abstract representation of valuation algebras in practice. The developing process itself results in a piece of software implementing local computation independently from any instantiation of valuation algebras. A further level of abstraction according to the architecture of local computation allows to integrate different architecture types each one taking advantage of additional properties of the valuation algebra instance. For illustration purposes, the realization of a famous medical example will guide through the chapters.

Chapter 1

Valuation Algebras

The basic elements of a valuation algebra are so-called *valuations*. Intuitively, a valuation can be regarded as a representation of knowledge about the possible values of a set of variables. Thus it can be said that each valuation ϕ refers to a finite set of variables $d(\phi)$, called its *domain*. For an arbitrary set s of variables, Φ_s denotes the set of valuations ϕ with $d(\phi) = s$. With this notation, the set of all possible valuations corresponding to a finite set of variables r can be defined as

$$\Phi = \bigcup_{s \subseteq r} \Phi_s. \tag{1.1}$$

Furthermore, let D be the lattice of subsets (the powerset) of r. Given a single variable X, Ω_X denotes the set of all its possible values. We call Ω_X the *frame* of the variable X. In an analogous way, we define the frame of a non-empty variable set s by the Cartesian product of frames Ω_X of each variable $X \in s$,

$$\Omega_s = \prod_{X \in s} \Omega_X. \tag{1.2}$$

The elements of Ω_s are called *configurations* of s. The frame of the empty variable set is defined by convention as $\Omega_\emptyset = \{\diamond\}$. This summarizes the most important notations and allows to define formally a *valuation algebra* by a set of operations and axioms.

1.1 A Formal Definition

Let Φ be a set of valuations with their domains in D. We assume the following operations defined on Φ and 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:* $\Phi \times D \to \Phi$; $(\phi, x) \mapsto \phi^{\downarrow x}$.

These are the three basic operations of a valuation algebra. If we interpret valuations as pieces of knowledge, the labeling operation tells us the questions

to which this knowledge corresponds. Combination on the other hand can be understood as aggregation of knowledge and marginalization as the focussing of some knowledge onto a smaller set of questions.

We impose now the following set of axioms on Φ and D:

1. *Semigroup:* Φ is associative and commutative under combination. For all $s \in D$ there is an element e_s with $d(e_s) = s$ such that for all $\phi \in \Phi$ with $d(\phi) = s$, $e_s \otimes \phi = \phi \otimes e_s = \phi$.

2. *Labeling:* For $\phi, \psi \in \Phi$,

$$d(\phi \otimes \psi) = d(\phi) \cup d(\psi). \tag{1.3}$$

3. *Marginalization:* For $\phi \in \Phi$, $x \in D$, $x \subseteq d(\phi)$,

$$d(\phi^{\downarrow x}) = x. \tag{1.4}$$

4. *Transitivity:* For $\phi \in \Phi$ and $x \subseteq y \subseteq d(\phi)$,

$$(\phi^{\downarrow y})^{\downarrow x} = \phi^{\downarrow x}. \tag{1.5}$$

5. *Combination:* For $\phi, \psi \in \Phi$ with $d(\phi) = x$, $d(\psi) = y$,

$$(\phi \otimes \psi)^{\downarrow x} = \phi \otimes \psi^{\downarrow x \cap y}. \tag{1.6}$$

6. *Neutrality:* For $x, y \in D$,

$$e_x \otimes e_y = e_{x \cup y}. \tag{1.7}$$

Axiom (1) indicates that Φ is a commutative semigroup under combination and that a unique neutral element exists for every sub-semigroup Φ_s. The labeling axiom tells us that the domain of a combination is the union of both valuations' domains. The marginalization axiom says that the resulting valuation of a marginalization on a domain x is itself a valuation on this domain x. Transitivity means that marginalization can be performed in two or more steps. The combination axiom says that instead of combining two valuations and marginalizing the result on the first valuation's domain, we can marginalize first the second valuation on the intersection of both valuations' domains and perform the combination in a second step. At last, the neutrality axiom assures that the combination of two neutral elements results in the neutral element of the union domain.

Definition 1 *A system (Φ, D) together with the operations of labeling, marginalization and combination satisfying these axioms is called a (labeled) valuation algebra.*

The following lemma describes a few elementary properties derived directly from this set of axioms:

Lemma 2 *1. If ϕ, $\psi \in \Phi$ with $d(\phi) = x$ and $d(\psi) = y$, then*

$$(\phi \otimes \psi)^{\downarrow x \cap y} = \phi^{\downarrow x \cap y} \otimes \psi^{\downarrow x \cap y}. \tag{1.8}$$

2. If ϕ, $\psi \in \Phi$ with $d(\phi) = x$, $d(\psi) = y$ and $z \subseteq x$, then

$$(\phi \otimes \psi)^{\downarrow z} = (\phi \otimes \psi^{\downarrow x \cap y})^{\downarrow z}. \tag{1.9}$$

3. If ϕ, $\psi \in \Phi$ with $d(\phi) = x$, $d(\psi) = y$ and $x \subseteq z \subseteq x \cup z$, then

$$(\phi \otimes \psi)^{\downarrow z} = (\phi \otimes \psi^{\downarrow y \cap z}). \tag{1.10}$$

Proof. Proofs can be found in [Koh03].

As we will see, these properties together with the combination axiom enable local computation; the topic of chapter 2.

The operation of marginalization can be replaced by an equivalent operation called *variable elimination*. This replacement changes our previously defined operation set. For the same system (Φ, D) and the set of variables r we define:

1. *Labeling:* $\Phi \to D$; $\phi \mapsto d(\phi)$,
2. *Combination:* $\Phi \times \Phi \to \Phi$; $(\phi, \psi) \mapsto \phi \otimes \psi$,
3. *Variable elimination:* $\Phi \times r \to \Phi$; $(\phi, X) \mapsto \phi^{-X}$ for $X \in d(\phi)$.

The equivalence between marginalization and variable elimination is shown by the following equation:

$$\phi^{-X} = \phi^{\downarrow d(\phi) - \{X\}} \tag{1.11}$$

Hence, the axioms of a valuation algebra can now be rewritten with the operation of marginalization replaced by variable elimination.

1. *Semigroup:* Φ is associative and commutative under combination. For all $s \in D$ there is an element e_s with $d(e_s) = s$ such that for all $\phi \in \Phi$ with $d(\phi) = s$, $e_s \otimes \phi = \phi \otimes e_s = \phi$.
2. *Labeling:* For ϕ, $\psi \in \Phi$,

$$d(\phi \otimes \psi) = d(\phi) \cup d(\psi). \tag{1.12}$$

3. *Variable elimination:* For $\phi \in \Phi$, $X \in d(\phi)$,

$$d(\phi^{-X}) = d(\phi) - \{X\}. \tag{1.13}$$

4. *Commutativity of variable elimination:* For $\phi \in \Phi$ and X, $Y \in d(\phi)$,

$$(\phi^{-X})^{-Y} = (\phi^{-Y})^{-X}. \tag{1.14}$$

5. *Combination:* For ϕ, $\psi \in \Phi$ with $d(\phi) = x$, $d(\psi) = y$ and $Y \notin x$, $Y \in y$,
$$(\phi \otimes \psi)^{-Y} = \phi \otimes \psi^{-Y}. \tag{1.15}$$

6. *Neutrality:* For x, $y \in D$,
$$e_x \otimes e_y = e_{x \cup y}. \tag{1.16}$$

The valuation algebra corresponding to the operations of labeling, variable elimination and combination satisfying the axioms above is denoted by (Φ, r).

Until now, the definition of variable elimination is restricted to the elimination of single variables. But the commutativity of variable elimination allows to perform their elimination ambiguously and independent of any ordering. We extend the notation on sets of variables as follows: For a sequence of variables $X_1, \ldots, X_n \in d(\phi)$ the elimination of a variable set is defined by
$$\phi^{-\{X_1,\ldots,X_n\}} = (\cdots(\phi^{-X_1}))^{-X_n}. \tag{1.17}$$

This extension implies some elementary properties:

Lemma 3 *1. If $x \subseteq d(\phi)$ for some $\phi \in \Phi$, then*
$$d(\phi^{-x}) = d(\phi) - x. \tag{1.18}$$

2. If x and y are two disjoint subsets of $d(\phi)$ for some $\phi \in \Phi$, then
$$(\phi^{-x})^{-y} = \phi^{-(x \cup y)}. \tag{1.19}$$

3. If ϕ, $\psi \in \Phi$ and y a subset of $d(\psi)$, disjoint to $d(\phi)$, then
$$(\phi \otimes \psi)^{-y} = \phi \otimes \psi^{-y}. \tag{1.20}$$

To complete the relationship between marginalization and variable elimination, we express vice versa the operation of marginalization by variable elimination,
$$\phi^{\downarrow x} = \phi^{-(d(\phi)-x)}. \tag{1.21}$$

To sum it up, we have seen two different ways of defining valuation algebras (Φ, D) and (Φ, r). The following theorem states that both definitions are equal.

Theorem 4 *If (Φ, D) is a valuation algebra and variable elimination is defined by (1.11), then the system of axioms above is satisfied. If (Φ, r) satisfies the axioms above and marginalization is defined as (1.21), then (Φ, D) is a valuation algebra.*

Proof. A detailed proof of the former properties and theorem 4 can be found in [Koh03].

To close this introduction, we mention without going into further details that examples of valuation algebras without neutral elements exist. Although we will continue our studies with the former defined set of axioms, this fact will have an influence on the later implementation process.
To internalize this formal introduction of valuation algebras, we may study *probability potentials* as a first concrete instance in the following section.

1.2 Probability Potentials

Probability potentials are perhaps the most popular example of a valuation algebra. They are closely related to discrete probability theory and they are well-suited to illustrate the concept introduced in the previous section.

Let $s = \{X_1, \ldots, X_n\}$ be a set of variables with finite frames $\Omega_{X_1}, \ldots, \Omega_{X_n}$. In the context of probability potentials, valuations are discrete functions $p : \Omega_s \to \mathbb{R}^+$. Any such discrete function can be represented as n-dimensional table of size $|\Omega_s| = |\Omega_{X_1}| \times \ldots \times |\Omega_{X_n}|$. If

$$\sum_{\mathbf{x} \in \Omega_s} p(\mathbf{x}) = 1, \tag{1.22}$$

the potential p is said to be *normalized* or *scaled*. In this case, the potential corresponds to a discrete probability distribution. Note that the bold-letter typeface $\mathbf{x}$ stands for configurations of the corresponding variable set.
To link probability potentials with valuation algebras, the basic operations of combination and marginalization have to be introduced. Combination is defined as multiplication: If p_1 is a potential over s and p_2 a potential over t, then for $\mathbf{x} \in \Omega_{s \cup t}$,

$$p_1 \otimes p_2(\mathbf{x}) = p_1(\mathbf{x}^{\downarrow s}) p_2(\mathbf{x}^{\downarrow t}). \tag{1.23}$$

The neutral element for s is $e(\mathbf{x}) = 1$ for all $\mathbf{x} \in \Omega_s$. Marginalization consists of summing up all variables to be eliminated. If p is a potential over s and $t \subseteq s$, then each configuration in Ω_s is decomposed according to the subsets t and $s - t$. Now, if $\mathbf{x} \in \Omega_t$,

$$p^{\downarrow t}(\mathbf{x}) = \sum_{\mathbf{y} \in \Omega_{s-t}} p(\mathbf{x}, \mathbf{y}). \tag{1.24}$$

These are the definitions of combination and marginalization in the context of probability potentials. The next step consists in verifying the axioms of a valuation algebra. Most of these axioms are easy to proof because the definition of combination and marginalization leads them back to simple calculus on real numbers. Nevertheless, the detailed steps of most of the proofs can be found in [Koh03].

Theorem 5 *Probability potentials together with the above definitions of combination, marginalization and neutral elements form a valuation algebra.*

At last, let us have a closer look on the influence of normalized potentials. As already mentioned, the motivation of using normalized potentials is to have a one-to-one relationship with a discrete probability distribution. However, it is easy to see that the combination of normalized potentials doesn't result in a normalized potential anymore. Therefore, an alternative combination must be defined to guarantee a normalized result at any time. Let p_1 be a normalized potentials over s, p_2 a normalized potential over t and $\mathbf{x} \in \Omega_{s \cup t}$. The alternative combination is defined as:

$$p_1 \otimes p_2(\mathbf{x}) = \frac{1}{K} p_1(\mathbf{x}^{\downarrow s}) p_2(\mathbf{x}^{\downarrow t}) \tag{1.25}$$

with

$$K = \sum_{\mathbf{x} \in \Omega_{s \cup t}} p_1(\mathbf{x}^{\downarrow s}) p_2(\mathbf{x}^{\downarrow t}). \tag{1.26}$$

If $K = 0$, $p_1 \otimes p_2(\mathbf{x}) = 0$. It can be shown that with this definition, the result of a combination is itself normalized if $K \neq 0$. Of course, the definition of the neutral element must itself be adapted in the case of normalization. The neutral element over s is now $e_s(\mathbf{x}) = 1/|\Omega_s|$ and it can be verified that the axioms of a valuation algebra stay satisfied for normalized potentials.

The following example may help to have a more intuitive understanding of probability potentials and their basic operations of combination and marginalization. To avoid rounding problems, the unnormalized definition of the combination is used here.

Example: Let $r = \{X, Y, Z\}$ be a set of binary variables, i.e. $\Omega_X = \Omega_Y = \Omega_Z = \{0, 1\}$. Consider the following two potentials $p_1 : \Omega_{\{X,Y\}} \to \mathbb{R}^+$ and $p_2 : \Omega_{\{Y,Z\}} \to \mathbb{R}^+$ given by the two tables:

$p_1(\mathbf{x}) =$

X	Y	
0	0	0.1
0	1	0.3
1	0	0.4
1	1	0.2

$p_2(\mathbf{x}) =$

Y	Z	
0	0	0.3
0	1	0.2
1	0	0.1
1	1	0.4

Labeling: $d(p_1) = \{X, Y\}$ and $d(p_2) = \{Y, Z\}$

Combination and Marginalization: The combination of p_1 and p_2, named p_3, results in the table below. This example illustrates that the combination of two normalized potential doesn't lead to a normalized potential anymore. Additionally, the second table shows the resulting potential after marginalizing p_3 onto $t = \{X, Y\}$.

$p_3(\mathbf{x}) = p_1 \otimes p_2(\mathbf{x}) =$

X	Y	Z	
0	0	0	0.03
0	0	1	0.02
0	1	0	0.03
0	1	1	0.12
1	0	0	0.12
1	0	1	0.08
1	1	0	0.02
1	1	1	0.08

$p_3^{\downarrow t}(\mathbf{x})$

X	Y	
0	0	0.05
0	1	0.15
1	0	0.20
1	1	0.10

Although p_3 is not normalized anymore, we can simply transform it back to a equivalent but normalized version. This process is called *scaling* and its result is denoted by $p_3^{\downarrow}$. To guarantee that the probabilities sum up to one, we will divide each probability entry by the total sum of all probabilities in this

potential. Coming back to the example above, we see that the total sum in p_3 is 0.5. The following table shows the normalized version of p_3 and the reader can easily verify that this corresponds again to a discrete probability distribution.

$p_3^{\downarrow} =$

X	Y	Z	
0	0	0	$0.03/0.5 = 0.06$
0	0	1	$0.02/0.5 = 0.04$
0	1	0	$0.03/0.5 = 0.06$
0	1	1	$0.12/0.5 = 0.24$
1	0	0	$0.12/0.5 = 0.24$
1	0	1	$0.08/0.5 = 0.16$
1	1	0	$0.02/0.5 = 0.04$
1	1	1	$0.08/0.5 = 0.16$

1.3 A Medical Example

Although probability potentials present a handsome example of a valuation algebra, their practical profit is as ever vague. In this section we will therefore consider a concrete application out of medical diagnostic. This example has been proposed by [LS88] and gives a first impression of the theory's practical application. The starting point is the following piece of fictitious medical knowledge:

> *Shortness-of-breath (dyspnoea) may be due to tuberculosis, lung cancer or bronchitis, or none of them, or more than one of them. A recent visit to Asia increases the chances of tuberculosis, while smoking is known to be a risk factor for both lung cancer and bronchitis. The results of a single chest X-ray do not discriminate between lung cancer and tuberculosis, as neither does the presence or absence of dyspnoea.*

This medical background needs to be applied to the following hypothetical situation: A patient presents himself at a chest clinic with dyspnoea and has recently visited Asia. Smoking history and chest X-ray are not yet available. The doctor would like to know the chance that each of the diseases is present. In this example we can distinguish two kinds of information, the general medical knowledge on one hand and information about a specific patient such as a recent visit to Asia on the other hand. We will refer to the latter kind of information as observations.

Clearly, the medical report above represents a classical knowledge-base and can therefore be studied in the context of a valuation algebra: In this example we find eight variables: A (visit to Asia), S (Smoking), T (Tuberculosis), L (Lung cancer), B (Bronchitis), E (Either tuberculosis or lung cancer), X (positive X-ray) and D (Dyspnoea). Each of these variables has a binary frame representing the possible values *true* or *false* modeled by 1 and 0.

The next step consists in finding the valuations that represent the facts in the medical report. There are eight valuations: α for $\{A\}$, σ for $\{S\}$, τ for $\{A, T\}$, λ for $\{L, S\}$, β for $\{S, B\}$, ϵ for $\{T, L, E\}$, ξ for $\{E, X\}$ and δ for $\{E, B, D\}$. The

observations about a specific patient are modeled by two additional valuations: o_A for $\{A\}$ and o_D for $\{D\}$. The structure of the knowledge-base is represented by the directed graph in Figure 1.1. It is worth to mention that we did not describe these valuations concretely, this will be done in section 5.2.

What is the interest of such an example? The doctor wants to know the chance

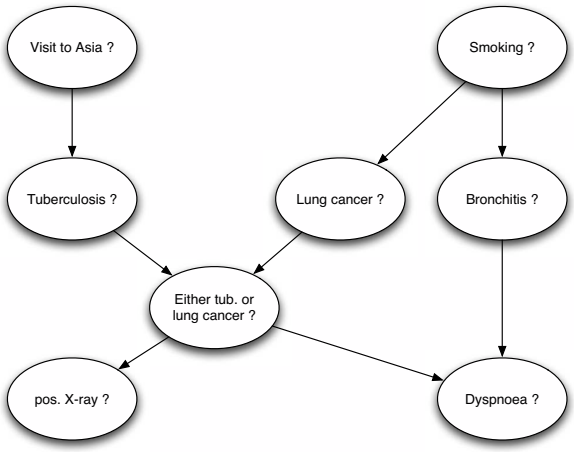


Figure 1.1: Causal network of a fictitious medical example.

that each of the diseases is present. To answer these questions, all information has to be aggregated, in order to focus afterwards the result on the question of interest. In this case, the questions of interest are either the presence of tuberculosis T, lung cancer L of bronchitis B. That is exactly the meaning of the following computation problem formulated for the case of tuberculosis:

$$(\alpha \otimes \sigma \otimes \tau \otimes \lambda \otimes \beta \otimes \epsilon \otimes \xi \otimes \delta \otimes o_A \otimes o_D)^{\downarrow\{T\}}. \tag{1.27}$$

It is well-known that the complexity of computation and of memory space for the operations of combination and marginalization tends to increase exponentially with the size of the valuations' domains. This is a crucial problem for the concrete computation of (1.27). The labeling axiom says that whenever the combination of two valuations is computed, the domain of the resulting valuation grows to the union of both valuation's domains. Therefore, a step-by-step computation of the combinations in (1.27) is not acceptable anymore. A similar problem exists for the final marginalization which would be performed on a maximum domain. In the following chapter we try to find an alternative computation scheme to reduce the efforts of such computations. This will be done by reorganizing the computations in such a way that each operation takes place on a domain which is not essentially larger than the domains of the original valuations. Such computation schemes are called *local computations*. The medical example itself will accompany our further studies and it will be treated whenever it is useful to have a concrete illustration of the introduced processes.

Chapter 2

Local Computation

We have seen that a valuation can be interpreted as some kind of knowledge container. Hence, a knowledge-base is represented by a set of such valuations $\{\phi_1, \phi_2, \ldots, \phi_m\}$. Our basic point of interest consists in using this knowledge-base to answer one or more *queries*. A straightforward way is to compute first the global aggregation of the knowledge-base and to focus afterwards the result on the domain of interest given by the query. This idea is put in a more concrete form: Let $\{\phi_1, \phi_2, \ldots, \phi_m\}$ be a set of valuations with $s_i = d(\phi_i)$. We consider now the joint valuation ϕ given by:

$$\phi = \phi_1 \otimes \phi_2 \otimes \ldots \otimes \phi_m$$

with $d(\phi) = s_1 \cup s_2 \cup \ldots \cup s_m$ and its marginalization onto a given query $x \subseteq d(\phi)$

$$\phi^{\downarrow x} = (\phi_1 \otimes \phi_2 \otimes \ldots \otimes \phi_m)^{\downarrow x}.$$

Although this first sight computation scheme would lead to the answer of the query, its efficiency has to be scrutinized. It is a well-known phenomenon that the complexity of combination and marginalization tends to increase exponentially with the size of the domains. By first calculating the global combination step-by-step, the domain of the operations will grow with each step and at last, the final marginalization has to be performed on the maximum domain $d(\phi)$. This would clearly be unbearable for most applications. We are therefore interested in methods where the operations can be limited to smaller domains. An obviously good procedure is to reorganize the computations in a way that each operation takes place in just one of the factors' domains. Such schemes of computations are called local computations. Local computation works with variable elimination instead of marginalization and this leads in a very straightforward way to a graphical structure called *join trees*. The computations on join trees themselves are organized as a message passing algorithm and there exists different architecture types taking advantage of additional properties of the underlying valuation algebra.
The following chapters will introduce an architecture called *Shenoy-Shafer architecture*, which has first been introduced in the work of [SS90]. The Shenoy-Shafer architecture is the most general one because it demands just a valuation algebra without further restrictions. Other important architecture types to mention are HUGIN [JLO90] and Lauritzen-Spiegelhalter [LS88].

2.1 Fusion Algorithm

As already mentioned, the starting point is to introduce the problem in the context of variable elimination. Let (Φ, D) be a valuation algebra. We consider a factorization of some $\phi \in \Phi$,

$$\phi = \phi_1 \otimes \phi_2 \otimes \ldots \otimes \phi_m, \tag{2.1}$$

with $d(\phi_i) = s_i$ and $d(\phi) = s = s_1 \cup s_2 \cup \ldots \cup s_m$. Furthermore, let $X_1, X_2, \ldots, X_n$ be an arbitrary arrangement of all variables $X_i \in s$ with $|s| = n$. It is known that, instead of computing $\phi^{\downarrow X_n}$, all other variables in the sequence can be eliminated. The question is: Does this stepwise elimination of single variables solve the complexity problem described in the introduction of this section? The answer is given by the following lemma:

Lemma 6 *In a valuation algebra, if $\phi \in \Phi$ is factored as (2.1), then*

$$\phi^{-X_1} = \psi^{-X_1} \otimes \left(\bigotimes_{i: X_1 \notin s_i} \phi_i \right), \tag{2.2}$$

where

$$\psi = \left(\bigotimes_{i: X_1 \in s_i} \phi_i \right).$$

Proof: A proof of this lemma can be found in [Koh03].

The statement of this lemma is very powerful. Instead of eliminating the variable X_1 over the whole domain $d(\phi)$, it is sufficient to eliminate it only in the often much smaller domain

$$\bigcup_{i: X_1 \in s_i} s_i.$$

We apply the same procedure successively to the remaining variable $X_2, \ldots, X_{n-1}$. Actually, this lemma is a simple conclusion of the combination axiom of variable elimination.
Shenoy named this algorithm of successive variable elimination *fusion algorithm*. Comparing the initial factorization of ϕ (2.1) with the factorization of ϕ^{-X_1} in (2.2), we notice that the valuations whose domain does not contain X_1 remain unchanged. On the other hand, all other valuations are first combined and then X_1 is eliminated from the resulting valuation. This operation is called *fusion*. We continue with a more formal definition of this process.

Let $Fus_Y(\{\phi_1, \phi_2, \ldots, \phi_m\})$ denote the set of valuations after fusing the valuations $\{\phi_1, \phi_2, \ldots, \phi_m\}$ with respect to the variable Y,

$$Fus_Y(\{\phi_1, \phi_2, \ldots, \phi_m\}) = \{\psi^{-Y}\} \cup \{\phi_i : Y \notin d(\phi_i)\}, \tag{2.3}$$

where

$$\psi = \left(\bigotimes_{i: Y \in d(\phi_i)} \phi_i \right).$$

Consequently, the result of lemma 6 can be reformulated as

$$\phi^{-X_1} = \bigotimes Fus_{X_1}(\{\phi_1, \phi_2, \ldots, \phi_m\}).$$

By repeated application of lemma 6 together with the commutativity axiom of variable elimination we get

$$\phi^{\downarrow X_n} = \bigotimes Fus_{X_{n-1}}(\ldots(Fus_{X_2}(Fus_{X_1}(\{\phi_1, \phi_2, \ldots, \phi_m\})))\ldots). \tag{2.4}$$

2.2 Graphical Representation of the Fusion Algorithm

The fusion algorithm can be illustrated constructively by an undirected graph. For this purposes, we restrict our attention on the domains of the valuations. At the beginning, the initial factorization (2.1) gives a list $l = \{s_1, s_2, \ldots, s_m\}$ of domains. Additionally, we choose an arbitrary elimination sequence $X_1, \ldots, X_n$ of the variables in s. For each variable X_i in this elimination sequence, we repeat the following algorithm:

Precondition: At iteration i, variables $X_1, \ldots, X_{i-1}$ have been eliminated. Clearly, the domains in l do not contain the variables $X_1, \ldots, X_{i-1}$ anymore. Therefore, we have the following situation:

$$\phi^{-\{X_1,\ldots,X_{i-1}\}} = \phi'_1 \otimes \ldots \otimes \phi'_{m'} \tag{2.5}$$

with $d(\phi'_i) = s'_i$. Let G be the resulting graph from the previous steps.

Step I: Build a new node s', containing the union of all domains in l, which contain the variable X_i,

$$s' = \bigcup_{j:X_i \in s'_j} s'_j.$$

Add this new node to the graph G.

Step II: All links found in the domains $\{s'_j : X_i \in s'_j\}$ are transformed into edges leading to the new node s'.

Step III: Modify the list l to the new list

$$l \cup \{s' - \{X_i\}\} - \{s'_j : X_i \in s'_j\}, \tag{2.6}$$

and annotate the new domain $s' - \{X_i\}$ with a link to the new node s' in the graph G.

This procedure corresponds exactly to the fusion process. All domains being absorbed by s' are removed and a new domain $s' - \{X_i\}$ is added at each step. After step n, the list l is empty. The creation of a new node goes simultaneous with the elimination of a variable and we can label each node with the number of the variable eliminated at this step. Thus, if i and j are two nodes

in the graph G and $i \leq j$, then node j has been introduced after node i. X_i is eliminated in node i and this leads to the property that X_i appears in no node introduced later than i. On the other hand, when X_i belongs to some node introduced earlier that node i, then X_i belongs to all nodes on the path from this node to node i.
At this point it is worth to go back to the medical example presented in section 1.3. Figures 2.1 and 2.2 show the stepwise construction of its graph when applying the fusion algorithm. These figures are taken from [Koh03]. Let us have a look at some very important properties of the graph G constructed by the fusion algorithm.

Lemma 7 *At the end of the fusion algorithm the graph G is a tree.*

This is actually not really a surprise if we look at the way how the graph has been constructed. Nevertheless, it is the first step in a proof that shows that the graph is a *join tree* and this is perhaps the most important property. A formal definition of a join tree is given by the so-called *Markov-Property*.

Definition 8 *A tree, whose nodes are domains s, is called a join tree, if for any pair of nodes s' and s'', if $X \in s' \cap s''$, then $X \in s$ for all nodes on the path between s' and s''.*

Lemma 9 *At the end of the fusion algorithm the graph G is a join tree.*

Proofs of this marvelous lemma and of the following two properties can be found in [Koh03].

Before we focus on two further properties of the graph G, we will introduce some new notations. For a given node s, we call the set of neighbor nodes s' which have been introduced before s its *parents*, and the unique neighbor that has been introduced later its *child*. For a node i, we denote these sets by $pa(i)$, $ch(i)$ respectively. If the set of parents is empty then the current node is called a *leaf*. Clearly, the latest node being introduced has no child and is therefore called the *root* of the join tree. At last we denote by t_i the domain of node i.

Lemma 10 *At the end of the fusion algorithm, the following two properties hold in the graph G:*

1. *For all i, we have $t_i \cap t_{ch(i)} = t_i - \{X_i\}$, and for all $j \in pa(i)$, we have $t_j \cap t_i = t_j - \{X_j\}$.*
2. *For all i we have*

$$t_i = \left(\bigcup_{j \in pa(i)} t_j - \{X_j\} \right) \cup \left(\bigcup_{j: X_1, \ldots, X_{i-1} \notin s_j, X_i \in s_j} s_j \right). \tag{2.7}$$

We should not forget that the graph G is only a graphical representation of the fusion process working on the domains of valuations. No real computing has been done until now. Thus, the next step consists in introducing these left out

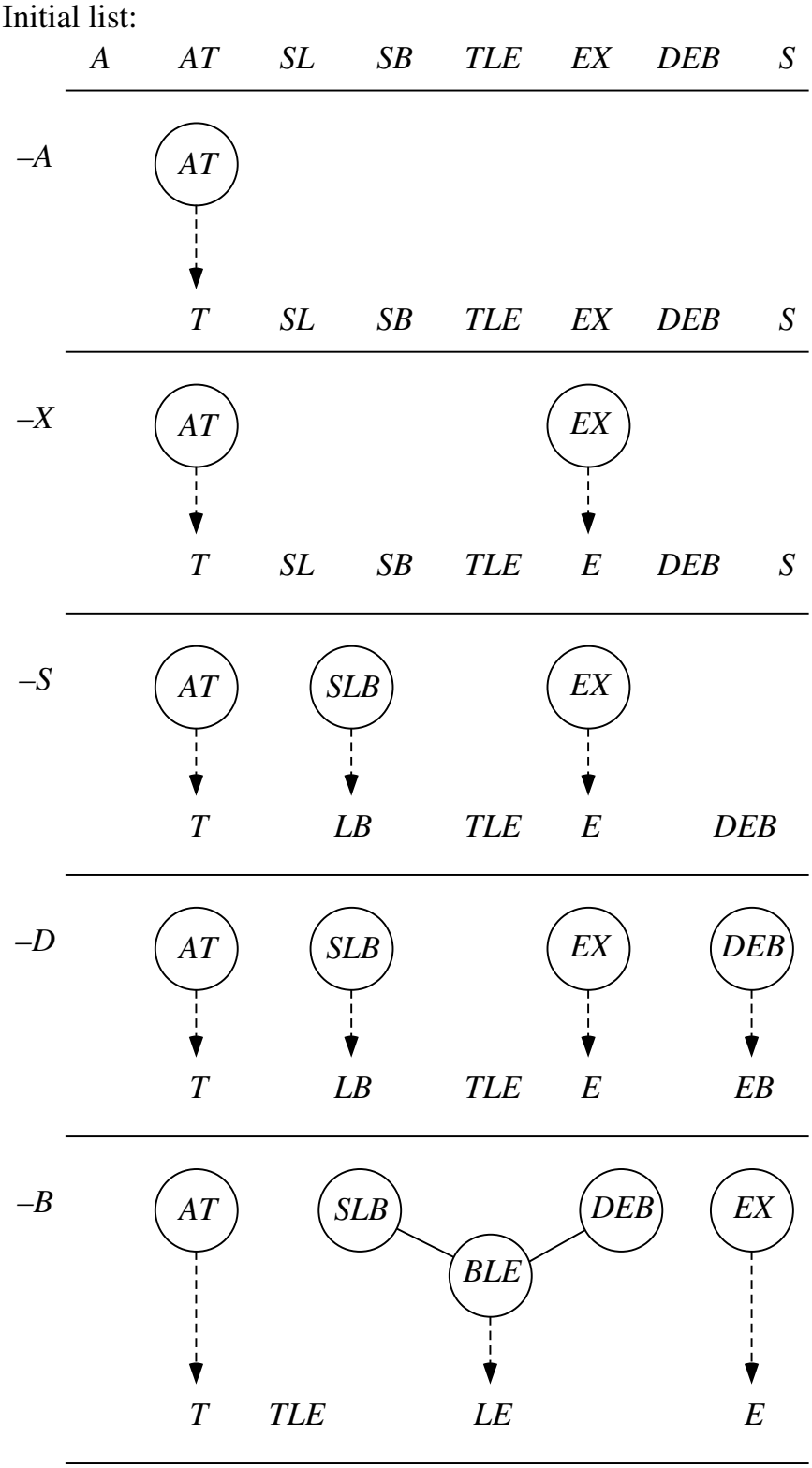


Figure 2.1: Building a graph by applying the fusion algorithm.

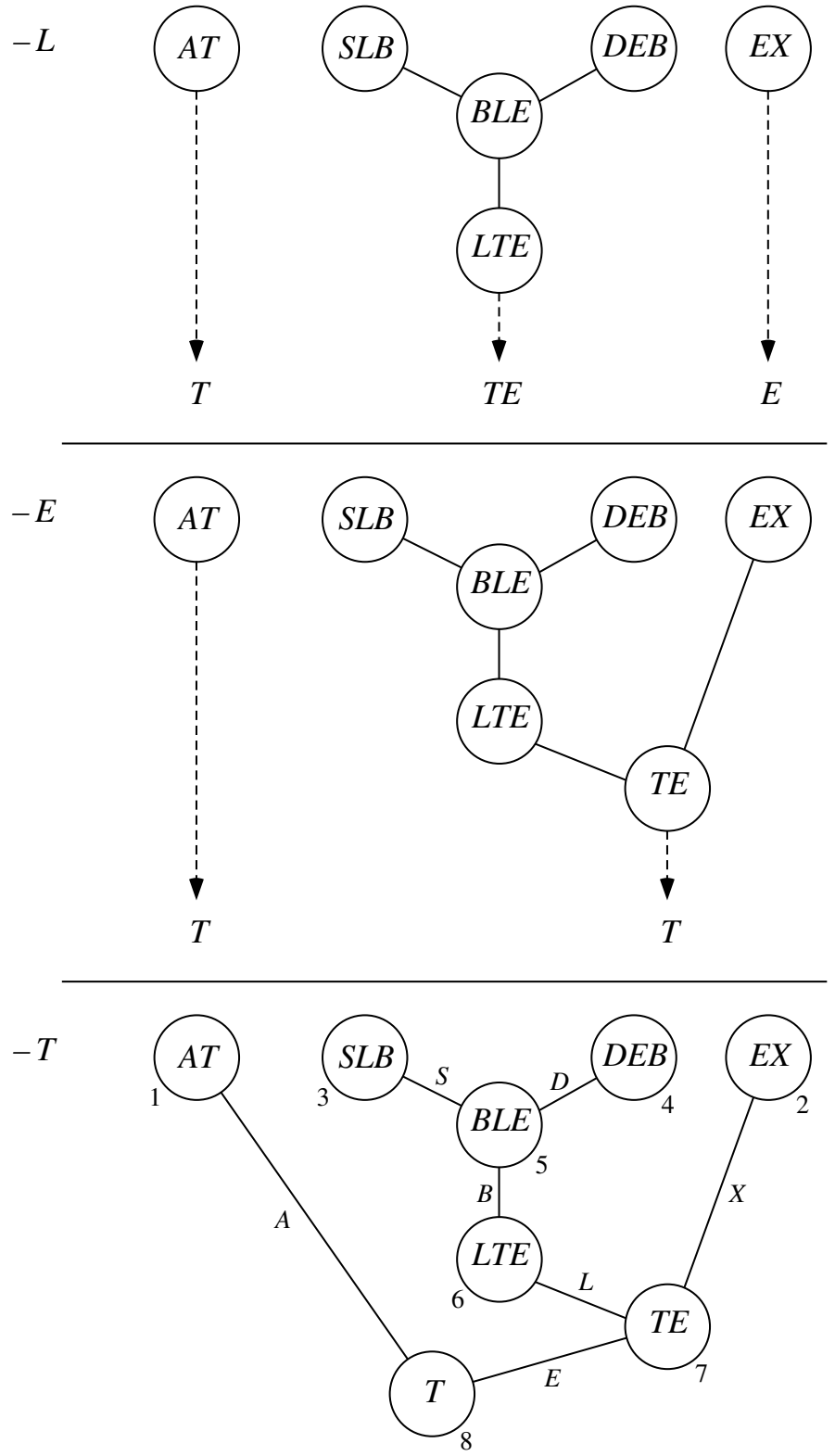


Figure 2.2: Continuation of the build process.

computations on the join tree. Once the join tree is constructed we are able to distribute the initial valuations $\{\phi_1, \phi_2, \ldots, \phi_m\}$ on the graph's nodes. In detail: for all domains s_i of the factors ϕ_i, there exists a node s on the join tree G such that $s_i \subseteq s$. The idea behind this fact is that if X_j is the first variable of s_i to be eliminated, then we have $s_i \subseteq t_j$. Next, we put every factor ϕ_i on the node t_j, if X_j is the first variable to be eliminated in s_i. If several valuations have to be put on one node, we combine them and put the resulting valuation on this node. Figure 2.3 shows the result of this valuation distribution process in the case of the medical example. This picture is taken from [Koh03].

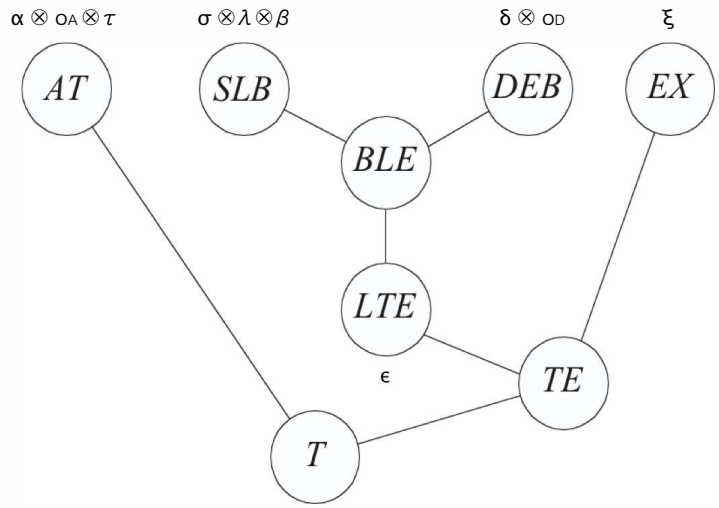


Figure 2.3: Distributing the initial valuations on the join tree.

2.3 Collect Algorithm

Until now, all join trees are constructed by a sequence of variable eliminations. Actually, we are not restricted on this but general join trees can be considered. To reach this more global level of abstraction, we have to define what is meant by saying: a valuation ϕ factors according to a join tree. This is exactly the statement of the following definition.

Definition 11 ***Property (J):*** *A valuation ϕ has property (J), if there is a factorization of ϕ,*

$$\phi = \psi_1 \otimes \psi_2 \otimes \ldots, \otimes \psi_m \tag{2.8}$$

such that the domains $d(\psi_i) = s_i$ are the nodes of a join tree. This is ever so often called a join tree factorization.

To recapitulate, the beginning of our investigation was simply an arbitrary factorization $\phi = \phi_1 \otimes \phi_2 \otimes \ldots, \phi_n$. The fusion algorithm supplies a join tree and the initial valuations can afterwards be distributed over the nodes of this

join tree. We assume that on every empty node of the join tree, we put a neutral valuation with the corresponding domain $\psi_j = e_{t_j}$. So, there is a valuation ψ_j on every node of the join tree. Consequently, we obtain a new factorization

$$\phi = \psi_1 \otimes \psi_2 \otimes \ldots \otimes \psi_m$$

since the domains of the added neutral valuations are smaller than $d(\phi)$. If we replace ψ_j on the nodes by $\psi_j \otimes e_{t_j}$, we obtain finally a factorization of ϕ according to the join tree G. Shortly spoken, a valuation ϕ with *property (J)*. So, the fusion algorithm provides a method for transforming an arbitrary factorization to a join tree factorization. The attentive reader remarks that we assume at this moment valuation algebras with neutral elements. A way out of this restriction will be presented later in this chapter.
With this knowledge, we can reformulate our initial problem. Instead of working with an arbitrary factorization of ϕ, we consider now a factorization like (2.8) according to a join tree G. This is not really a restriction because the fusion algorithm delivers such a factorization. To come back, we want to compute the marginal of ϕ relative to one of the factor's domain s_i. Let us assume s_m. We take the corresponding node s_m as root of the graph G and direct all edges towards this root. Now, the nodes are numbered in such a way, that if j is a node on the directed path from i to the root, then $j \leq i$. Remember at this point that, with the considered join tree factorization, each factor corresponds to a node in the tree and therefore we get also a numbering of the factors.
From now on, we will look at nodes as virtual processors with their own storage and describe a message passing algorithm with the goal that at the end of the message passing, the root contains the valuation $\phi^{\downarrow s_m}$. The messages themselves will be treated according to the introduced numbering. We denote by $\psi_j^{(i)}$ the content of the storage of node j before step i. The initialization at the beginning is given by $\psi_j^{(1)} = \psi_j$. The following steps describe the message passing algorithm in detail:

Step I: Each node waits to send its message to its unique child until it has received all messages from its parents. This implicates that leaves can send their messages right away.

Step II: As soon as a node is ready to send, it marginalizes its current content to the intersection of its domain and those of its child. Formally, the message is:

$$\mu_{i \to ch(i)} = \psi_i^{(i)\downarrow s_i \cap s_{ch(i)}}. \tag{2.9}$$

This message is sent to child $ch(i)$.

Step III: When a node receives a message, it replaces its current content by the combination of its former content and the message. The following equation describes this update procedure for the child $ch(i)$, when node i has sent its message:

$$\psi_{ch(i)}^{i+1} = \psi_{ch(i)}^{i} \otimes \mu_{i \to ch(i)}. \tag{2.10}$$

The storage of all other nodes does not change at step i, i.e. $\psi_j^{i+1} = \psi_j^i$, for all $j \neq ch(i)$.

We will refer to this procedure as *collect algorithm*. It is worth mentioning at this occasion, that the three steps above summarize also the computing part of the fusion algorithm. The difference is that instead of performing a general marginalization, the fusion algorithm eliminates in step II exactly one variable. Another point is that the collect algorithm could be executed partially in parallel.

Theorem 12 *At the end of the collect algorithm, node m contains the marginal of ϕ relative to s_m,*

$$\psi_m^{(m)} = \phi^{\downarrow s_m}. \tag{2.11}$$

Proof. A proof of this theorem can be found in [Koh03].

It is even possible to generalize this theorem on all nodes of the join tree. This is motivated by the following reflection: Each node of the tree is itself a root node of a corresponding sub-tree and this sub-tree is by itself a join tree. For a given node i, we refer to its subtree by G_i.

Lemma 13 *At the end of the collect algorithm, each node i contains the marginal relative to s_i of the factors associated to the nodes of the tree G_i,*

$$\psi_i^{(i)} = \left(\bigotimes_{j \in G_i} \psi_j \right)^{\downarrow s_i}. \tag{2.12}$$

It is clear that this marginal considers only the informations contained in the valuations of its sub-tree.

2.4 Computing Multiple Marginals

The collect algorithm provides a method to calculate a marginal of a valuation ϕ. Imagine the case that we want to compute multiple marginals of ϕ for different domains s_i. A straightforward idea is to repeat the collect algorithm for each domain s_i but this would end in a lot of redundant computations. Computing another marginal on the same factorization implies that we have to change the root node of the join tree. But this can easily be done by changing the direction of all edges between the old and the new root. Fig. 2.4 shows a simple illustration of this process.

Remember, the reason for directing edges in a join tree is to fix the direction of the message passing during the collect algorithm. Another possibility would be to address the messages i.e. to give each message a direction. For this purpose we introduce *mailboxes* on each edge between two neighboring nodes of the join tree. A schematic representation of such a mailbox is shown in Fig. 2.5. These mailboxes are used to store incoming messages. Leaving the idea of a directed graph, we can no longer identify parents and child among the neighbors of a

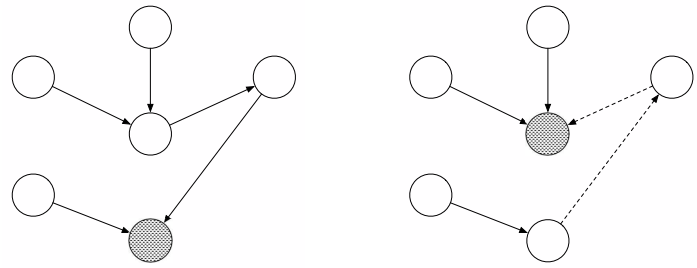


Figure 2.4: Changing the root of a join tree.

node. Therefore, we denote by $ne(i)$ the set of neighbors of node i in the join tree.

This leads to the following generalization of the message passing process: A

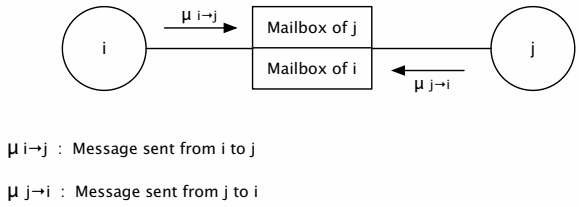


Figure 2.5: A schematic representation of mailboxes.

node i sends a message to its neighbor node j, as soon as it received all messages from the other neighbors. With the other neighbors' messages the new message can be computed with the following formula:

$$\mu_{i\to j} = \left(\psi_i \otimes \bigotimes_{k\in ne(i), k\neq j} \mu_{k\to i} \right)^{\downarrow s_i \cap s_j} . \tag{2.13}$$

It is clear that at the beginning, the leaves of the join tree can send their messages directly without any further computations. Figure 2.4 shows a possible run of the message passing within a join tree and is taken from [Koh03]. This computation scheme is called *Shenoy-Shafer architecture* and the following theorem confirms the expected result of this message passing process:

Theorem 14 *At the end of the message passing algorithm in the Shenoy-Shafer architecture, we obtain at node i*

$$\phi^{\downarrow s_i} = \psi_i \otimes \left(\bigotimes_{j\in ne(i)} \mu_{j\to i} \right) \tag{2.14}$$

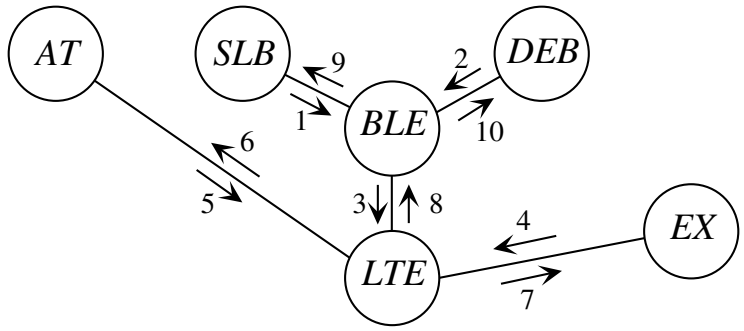


Figure 2.6: A possible run of the message passing process.

In other words: At the end of the message passing algorithm, the marginalization of ϕ on s_i can easily be computed by the formula above.
If we imagine a simulation of this process, we can distinguish two phases. In the first phase, all messages are sent towards the root. We refers to this phase as *inward propagation* or *collect phase* due to its nearness to the collect algorithm. In the second phase of the Shenoy-Shafer algorithm, the messages are sent from the root towards the leaves. This phase is named *outward propagation* or *distribute phase*. Another important gain of a message passing algorithm with mailboxes is that we don't need neutral elements to initialize the tree nodes anymore. This is really a fundamental generalization because not all valuation algebras have neutral elements.

2.5 Scaled Shenoy-Shafer Architecture

Having the example of probability potentials in the back of our mind motivates us to think about extending the Shenoy-Shafer architecture to take scaling into consideration. Normalized probability potentials correspond to discrete probability distributions and as we have seen, the combination of scaled probability potentials doesn't lead to a normalized result of necessity. On the other hand, section 1.2 presented a procedure to lead them back to their normalized versions. This procedure is called scaling. Now, the question arises if scaling could be done directly during local computation. Again, we restrict our investigation on the Shenoy-Shafer architecture. Combining scaling with the Shenoy-Shafer architecture, we will meet the notions of *division* and *inverse elements* in the context of valuation algebras. Giving a formal introduction to these concepts would go beyond the scope of this work, so we will rather impart a more intuitive understanding of these concepts.

To get into, we define the *normalization* or *scaled version* of a valuation ϕ by

$$\phi^{\downarrow} = \phi \otimes (\phi^{\downarrow\emptyset})^{-1}. \tag{2.15}$$

In this formula, ϕ is marginalized on the empty set and we remember that by convention, the frame of the empty set has only one single configuration, $\Omega_\emptyset = \{\diamond\}$. However, from this marginalization we compute its inverse element and combine it with the initial valuation ϕ to get its scaled version. Combining with inverse elements is often called division and demands an underlying valuation algebra with division. Generally, we distinguish two kinds of valuation algebras allowing division. These are: *separative valuation algebras* and *regular valuation algebras*. In the separative case, the result of a division will fall out of the corresponding valuation algebra, in the regular case on the other hand, the result is still part of the algebra. Probability potentials are typically an example of a regular valuation algebra. We have already illustrated in section 1.2 how scaling in the case of probability potentials works. Therefore, we go back to the initial situation of local computation and consider a valuation $\phi \in \Phi$ given by a join tree factorization

$$\phi = \psi_1 \otimes \psi_2 \otimes \ldots \otimes \psi_m, \tag{2.16}$$

with domains $s_i = d(\psi_i)$. Our interest concerns not only the marginal $\phi^{\downarrow s_i}$ but its scaled version $(\phi^\downarrow)^{\downarrow s_i}$. Clearly, in this case we assume a regular or separative valuation algebra. Some algebraic transformation together with the Shenoy-Shafer architecture show:

$$\begin{aligned}(\phi^\downarrow)^{\downarrow s_i} &= (\phi^{\downarrow s_i})^\downarrow = \phi^{\downarrow s_i} \otimes (\phi^{\downarrow\emptyset})^{-1} \\ &= \psi_i \otimes \left(\bigotimes_{j \in ne(i)} \mu_{j\to i}\right) \otimes (\phi^{\downarrow\emptyset})^{-1}\end{aligned}$$

This formula is not yet satisfying because it demands a division by $\phi^{\downarrow\emptyset}$ in each node. But this can be avoided by replacing the valuation ψ_r in the root of the join tree by

$$\psi_r \otimes (\phi^{\downarrow\emptyset})^{-1}.$$

This leads to the scaled version of the Shenoy-Shafer architecture which is given by the following theorem.

Theorem 15 *At the end of the computations in the scaled Shenoy-Shafer architecture, we obtain in node i*

$$(\phi^\downarrow)^{\downarrow s_i} = \psi_i \otimes \left(\bigotimes_{j \in ne(i)} \mu'_{j\to i}\right). \tag{2.17}$$

$\mu'_{j\to i}$ are the modified messages due to the scaled computations.

Proof. A detailed proof by induction can be found in [Koh03].

To sum it up, the scaled version of the Shenoy-Shafer architecture consists of the following steps: First, the collect algorithm is performed without any changes. At the end of the collect algorithm, the root node contains $\phi^{\downarrow r}$ which allows to compute $\phi^{\downarrow\emptyset}$. All outgoing messages from the root node (starting the distribute phase) use this new valuation. The division itself is computed one single time.

Chapter 3

Implementation Strategies

After having studied the mathematical concepts of local computation, the time has come to look at its realization. As already explained, we restrict our efforts on the most general architecture of local computation - the Shenoy-Shafer architecture. In section 2.2 we got join trees from a graphical illustration of the fusion process based on the valuations' domains. The real computations of the fusion process has been cut out during this process. Once a join tree exists, we can apply the collect algorithm in a second step to complete the missing computation parts. This allows us to concentrate our endeavors first on the efficient construction of join trees.
In this section we will present first a general but inefficient way to build up join trees based on the fusion algorithm. Then, this algorithm will be optimized and refined step-by-step by introducing some helping data structures. The descriptions of the algorithm and data structures are highly independent to any programming language. The following chapter will be about their realization in JAVA and their integration in a generic computation architecture.

3.1 Join Tree Construction

Before presenting an algorithm to build join trees in practice, we want to introduce some slight modification of their appearance. A join tree is defined by the so-called Markov property (see definition 8). This means that modifications will not offend the join tree definition, as long as the join tree respects the Markov property. The following list summarizes the addressed modifications and the corresponding motivation to introduce them:

- For each of the initial valuations $\phi_i \in \Phi$, a new leaf node with the corresponding domain will be attached to the join tree. This results in the new property that for each initial valuation a node with the same domain can be found without additional effort and that each leaf node contains one and only one of the initial valuations. This simplifies incredibly the distribution of the valuation at the beginning of the collect algorithm because we don't have to search the corresponding place in the join tree anymore. The valuations are assigned directly at node creation time.

- We will separate completely the operations of combination from marginalization during the message passing algorithms by introducing some new nodes in the join tree. Each iteration of the fusion algorithm will create a union node and a projection node, instead of one single node. The domain of the new nodes will be chosen in such a way that the Markov property stays untouched. Actually, the only justification of this step is to simplify the implementation of the message passing algorithm. We will come back to this point later in the explication of the join tree construction algorithm.
- Thanks to the distribute algorithm, it is not necessary anymore to choose the root node in dependence to the later queries. As another simplification we will therefore choose the empty domain as root of each join tree.

Perhaps the major disadvantage of these modification is that they all lead to the introduction of new nodes. So one could think that this results in a loss of efficiency. But as we will see, the only computation on the additional nodes is copying references and the resulting overhead can be neglected. The following piece of pseudo-code presents a general join tree construction algorithm based on the fusion process and the modification list above.

```
[01]  function construct(\{\phi_1, \ldots, \phi_n\})
[02]        \mathcal{N} = \emptyset;
[03]        \mathcal{E} = \emptyset;
[04]        \mathcal{R} = \{\mathbf{new\ node}(\phi_i), i = 1, \ldots, n\};
[05]        \mathcal{D} = \cup\{d(\phi_i, i = 1, \ldots, n\};
[06]        loop until |\mathcal{R}| \leq 1
[07]              select x \in \mathcal{D};
[08]              \Phi_x = \{N \in \mathcal{R}, x \in d(N)\};
[09]              N^u = new node (d(\Phi_x));
[10]              \mathcal{N} = \mathcal{N} \cup \Phi_x \cup \{N^u\};
[11]              \mathcal{E} = \mathcal{E} \cup \{\{N, N^u\}, N \in \Phi_x\};
[12]              \mathcal{R} = \mathcal{R} - \Phi_x;
[13]              if |\mathcal{R}| > 0
[14]                    N^p = new node(d(\Phi_x) - \{x\});
[15]                    \mathcal{R} = \mathcal{R} \cup \{N^p\};
[16]                    \mathcal{E} = \mathcal{E} \cup \{N^u, N^p\};
[17]              \mathcal{D} = \mathcal{D} - \{x\};
[18]        \mathcal{N} = \mathcal{N} \cup \mathcal{R};
```

Let us study this code in more detail. Row [04] performs the first modification we introduced. For each initial valuation a new node is constructed by this statement. The second modification is hidden within the loop. Statement [09] builds the union node, being the child of all nodes containing the current variable x. Then, row [14] builds the projection node and links it with the union node in statement [15]. The other statements of this code correspond to

the fusion algorithm. Figure 3.1 shows a schematic representation of the union and projection node.

Another interesting point steps out of statement [07]. The entry point of the

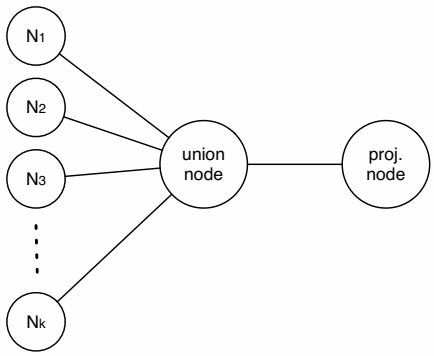


Figure 3.1: Union and projection node.

fusion algorithm was an arbitrary elimination sequence. Although each possible sequence would lead to a valid (but different) join tree, there are important differences in their optimality. Generally, we are interested in join trees, whose domains are as small as possible. This follows from the motivation of introducing local computation. But the problem of finding an optimal join tree turned out to be NP-hard [ACP87] and the only way out of this dilemma is to use heuristics. Before we have a closer look at such a heuristic working with a *hyper-graph*, we may illustrate the generation of different join trees by different elimination sequences with a simple example.

Example: Consider a set of valuations $\Phi = \{\phi_1, \phi_2, \phi_3\}$, with $d(\phi_1) = \{a, b\}$, $d(\phi_2) = \{b, c\}$ and $d(\phi_3) = \{c, d\}$. Let's construct join trees for two different elimination sequences as for example $E_1 = (a, b, c, d)$ and $E_2 = (b, c, a, d)$. Figure 3.2 shows the two resulting join trees. Although the first elimination sequence creates a join tree with more nodes, this first join tree is preferable because its domains are smaller.

To find a heuristic to determine a good elimination sequence, it is inalienable to analyse first the occurrences of each variable in the initial valuation set. A neat solution was proposed by [Leh01] and is based on a structure called hyper-graph. The hyper-graph is constructed by drawing a hyper-edge for each valuation and this hyper-edge surrounds all variables occurring in the corresponding valuation. The left part of Figure 3.3 shows the hyper-graph for the valuation set in the example above. Note that a variable is called a *leaf* of the hyper-graph, if it is surrounded by only one hyper-edge.

As well as the hyper-graph is constructed, a costs-notion for each variable can be computed standing for the costs of its elimination. If the current variable is a leaf, its costs are 0. Else, its costs corresponds to the number of variables

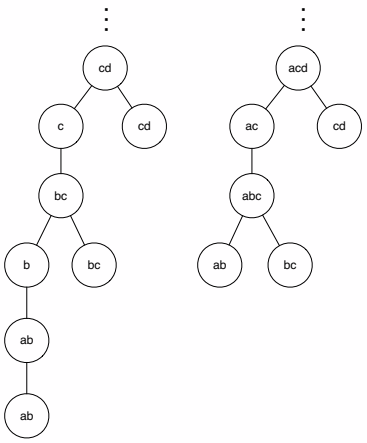
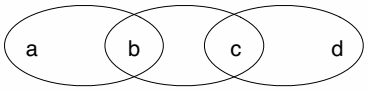


Figure 3.2: Join trees created with different elimination sequences.

being surrounded by a common hyper-edge. These variables are members of the variable's *clique*. Eliminating a variable in the join tree construction algorithm leads to the introduction of a new union node. The domain of this node is then the union of all valuations' domains containing the current variable. Eliminating the variable with the lowest costs leads to union nodes with small domains. This summarizes the basic idea of this heuristic. The right part of Fig. 3.3 shows the cost table of the example above. For each variable the table contains its clique, the cardinality of its clique and the corresponding elimination costs. Based on the hyper-graph and its cost table, we formulate the following



x	$leaf?$	$cl(x,\mathcal{H})$	$\lvert cl(x,\mathcal{H})\rvert$	$cost(x)$
a	yes	$\{a,b\}$	2	0
b	no	$\{a,b,c\}$	3	3
c	no	$\{b,c,d\}$	3	3
d	yes	$\{c,d\}$	2	0

Figure 3.3: A hypergraph with the corresponding cost table.

heuristic:

> Eliminate always the variable with the lowest costs. If there are more than one, choose a variable by random.

The problem of using this heuristic is that such a hyper-graph represents only a temporary situation of a valuation set. Whenever a variable is eliminated, certain domains can be dropped and another domain steps on stage. Hence, a hyper-graph is a dynamic structure that has to adapt itself to the current

set of domains. This problem will be handled later in this chapter. For the moment, it is enough to remark that the hyper-graph may not be constructed explicitly to compute a variable's cost. Instead, it is sufficient to count just the members of the cliques of each variable and the costs can be computed easily. This heuristic will be used instead of the selection in statement [07].

The construction of join trees using the algorithm above is not unproblematically because several components have to be managed simultaneous and this leads to severe efficiency concerns. These components are: the variables and their elimination costs, the valuation set, which variable occurs in which valuation and vice versa which valuation contains which variable. Let's analyse this problem a little bit more. First, we choose the next variable to eliminate by analyzing the cliques (search procedures) of each variable. Then, this variable has to be eliminated in all nodes containing it (search procedure). These nodes will disappear in favor of the union node. The domain of the union node has to be calculated (loop). Next, the costs of each variable must be updated (loop) because one variable has been eliminated. This brain-storming gives a good impression how a straightforward implementation would suffer from serious complexity problems. A brute force implementation of the construction algorithm is therefore not recommendable. A way out of this dilemma is the use of a so-called *VVLL* (*Variable Valuation Link List*).
A VVLL is a data structure for linking variables and valuations together. It allows via a pointer system to jump from a given variable to all valuations containing this variable respectively from a given valuation to all variables in its domain. Basically, a VVLL is set up by a *variable-container-list* and a *valuation-container-list*. The variable-container-list contains a *variable-container* for each existing variable. Each variable-container knows the costs to eliminate its variable and has a list of pointers to the valuation-containers containing this variable. The valuation-list on the other hand contains a *valuation-container* for each node that has not yet been linked with other nodes. The valuation-containers themselves are built up from the valuation contained in the appropriate node and from a list of pointer, pointing to the corresponding variable-container of the variables in the valuation's domain. May Figure 3.4 give a better idea of the internal structure of a VVLL. Its entries refer to the example above.

The advantages of using a VVLL are obvious. Whenever a variable is selected, the valuations containing this variable in their domain are known by the internal list of pointers. And later, after the elimination of this variable in a given valuation, we are able to update all variables' costs containing the current valuation by the other list of pointers. Figure 3.5 illustrates the changes in the VVLL caused by eliminating variable a.

The use of a VVLL solves the problem of correspondence between variables and valuations. Nevertheless, at each iteration, the variable with the lowest costs has to be found in the variable-container list. Another significant improvement that surrounds this additional search is to order the variable-containers in such a way, that the container with the lowest costs is always at first position in the list. But ordering the variable-containers is not as simple as one could think

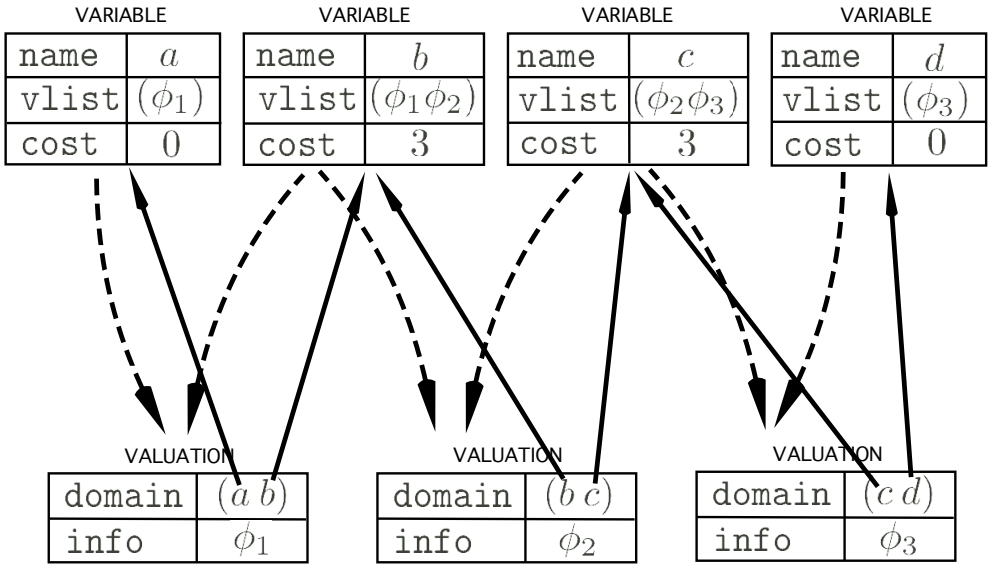


Figure 3.4: Illustration of a Variable-Valuation-Link-List (VVLL).

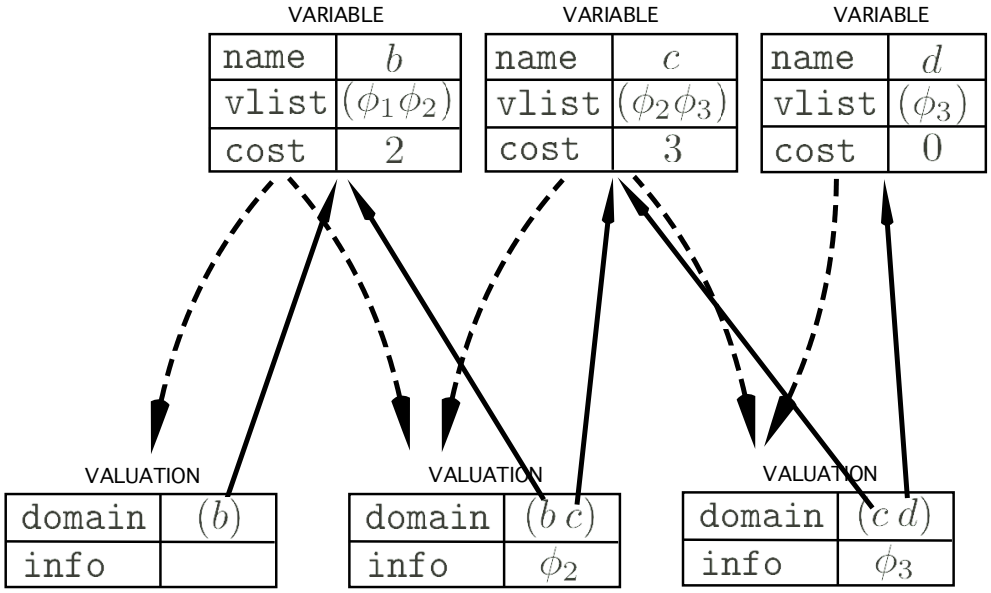


Figure 3.5: Updated VVLL after eliminating variable a.

because the variables' costs change periodically and the changes are distributed over the whole list. So, we would have to reorder the list at each time and this would even make is worse.
One more time, a solution to this problem is presented by the choice of a suitable data structure. In this case, we will use a *heap*. A heap is a binary tree where each node has a key such that the key of a parent is always lower or equal to the key of the children. Figure 3.6 shows a heap structure for some arbitrary chosen key values.

The power of a heap structure lies in its update complexity. A heap guaran-

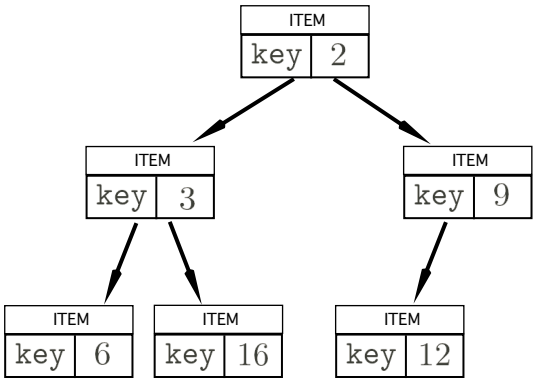


Figure 3.6: Heap data structure.

tees that the smallest element is always the root element, the same goal could be achieved by a simple list. But as soon as the ordering structure has been destroyed either by removing the root or by changing a variable's costs, a heap can reorganize itself with logarithmical complexity. It follows an overview of the most important heap procedures and their complexities:

Constructing the heap:	$O(n * log(n))$
Removing the root element:	$O(log(n))$
Updating an arbitrary element:	$O(log(n))$

For a better intuitive understanding of these complexities, we will describe the basic heap operations graphically. Removing the element with the lowest key value equals with removing the heap's root element. To realize it, replace the root element by the rightmost leaf of the heap and rattle it downwards the tree until the heap structure is satisfied again. Figure 3.8 illustrates this process. When a single element changes its key value, the process is inverse. The corresponding element is rattled upwards the tree until the heap structure is satisfied again. This process is shown by Figure 3.8. Both updating processes work on a single branch of the tree and are therefore logarithmic. In the VVLL

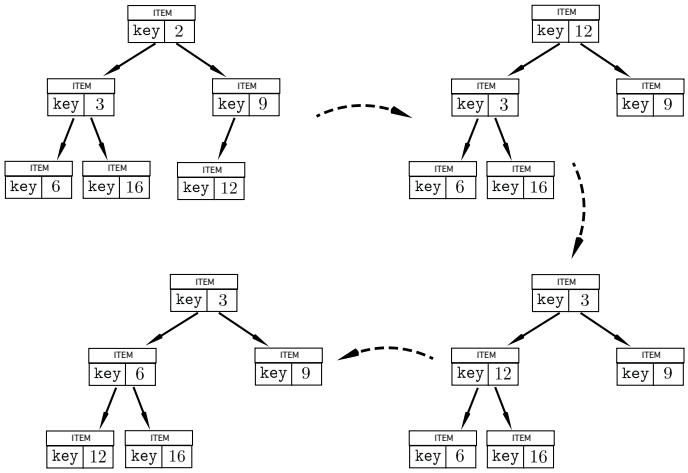


Figure 3.7: Removing a heap's root element.

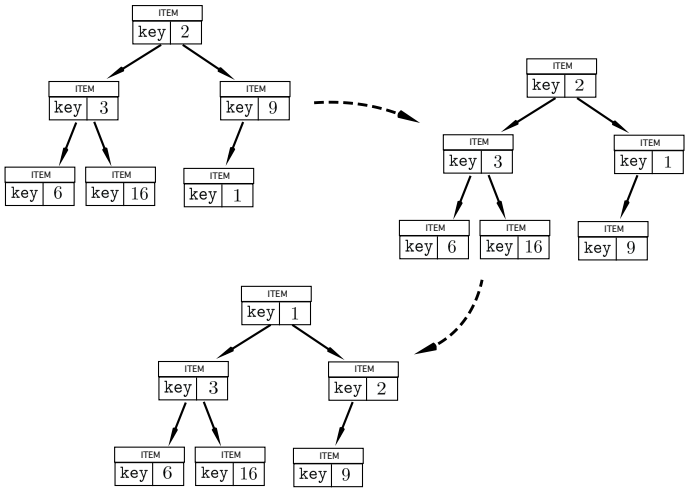


Figure 3.8: Updating a heap when an element's key has changed.

construction we will replace the list containing the variable-containers by a heap structure. More informations about heaps in general, its procedures and their implementation in JAVA can be found in [Bai98].

3.2 Answering queries

We already mentioned in the theory part that queries are represented by the domains onto which the global marginalization has to be computed. The situation at the end of the distribute algorithm is that each node contains the marginalization onto its proper domain. To answer a given query, we must therefore ensure that this query corresponds to the domain of a join tree node (it would be sufficient that the query is a subset of a node's domain but we will add this restriction to avoid marginalizations during query processing). Remember that the root domain of the join tree is always the empty domain. This implies that the construction itself is independent to the query of interest. Nevertheless, there must be a possibility to ensure that a given query can be answered by the join tree. A simple solution is presented by the use of neutral valuations. Building a neutral valuation with the appropriate domain and adding it to the knowledge-base ensures that this query can be answered at the end of the distribute algorithm. The problem of this procedure is that there exists valuation algebras without neutral element or valuation algebras with neutral elements that are not representable in practice such as relations with an infinite number of tuples for example. Fortunately, in practice a way exists to surround this problem although its application is not very proper in the mathematical sense. Every valuation algebra can artificially by extended with one neutral element per domain (we will refer to these special elements as *empty valuations* to respect the name given in the implementation). Combining a valuation ϕ with such an empty valuation returns ϕ unmodified and the combination of two empty valuations results in a new empty valuation whose domain is the union of both initial valuations' domains. These handicrafts work in our context of local computation but we will take care not to generalize it. Another special situation is present if a query contains variables that are not part of any valuation in the knowledge-base. Such queries are not allowed within this architecture.

Chapter 4

A Generic Computation Architecture in JAVA

The following subsequent chapters will guide through the development of a generic computation architecture in practice. From the developer's point of view it is advantageous to choose a programming language that offers itself facilities in the development of generic programs such as object oriented languages. For this end, we chose JAVA for its realization.
At this point, everyone has to become clear about the meaning of the word "generic". In the context of this architecture, its meaning is even two-fold. On one side a valuation algebra is a generic concept. We introduced it in section 1 by the three operations of labeling, combination and marginalization and by a set of axioms which have to be satisfied. From this point of view, a valuation algebra is a mathematical layer of abstraction that can be instantiated for various examples. A generic architecture must preserve this abstraction to allow then a concrete implementation for any instance of a valuation algebra. On the other side, the architecture of local computation is itself some kind of an abstract concept. We described in the theory part the Shenoy-Shafer architecture in detail but there are other architecture types that differ only in the implementations of their message passing algorithm and in their representation of join trees. So, an implementation of the local computation architecture must provide a solution to integrate different architecture types. We will take both asks for abstraction into account during the implementation process of this architecture.

4.1 Generic Representation of Valuation Algebras

Obviously, the entry point in the development of such an architecture is the generic representation of a valuation algebra. As we have seen in the first chapter, its fundamental components are variables, domains and valuations. Variables and valuations form the generic part, domains on the other hand can be represented statically because a domain is just a set of variables and the abstract nature of the variable implementation guarantees the necessary flexibility of the domain implementation.
A promising procedure in JAVA programming is to express generic parts as

interfaces or abstract classes. This leads to a method based description of variables and valuations independent of their internal appearance. It follows a detailed description of the corresponding JAVA API because every designer of a new instance would have to implement these interfaces.

Variables:

Variables are generic components although this is not evident at first sight. Mostly, variables are just simple strings and we are tempted to represent them statically as extension of the JAVA string type but the following example may illustrate the limitation of this solution: Suppose that we want to implement binary variables. The values of multiple binary variables can therefore be represented in a bit array. This bit array as far as it is concerned can itself be interpreted as a natural number. This is a very interesting representation because computation with natural numbers is more efficient that computation on strings. Nevertheless, in this case, variables are not strings anymore but indices in a bit array. This shows that variables must be generic components to ensure this liberty. The interface `Variable` contains the following method list:

- `public boolean equals(Object obj)`
- `public int hashCode()`
- `public String toString()`

The most important method `equals(Object obj)` performs an equality check and returns true, if two variables are equal by value. The hash-code method is used during the join tree creation. As already mentioned, different elimination sequences lead to different join trees. Finding an optimal join tree is known to be NP-hard, that is why we have to fall back on heuristics. The heuristic we use in this architecture requires a heap structure to order variables and the hash-code is used to perform the ordering operation. At last, the method `toString()` converts the current variable to a string representation. This method is used for output procedures.

Domains:

The generic implementation of variables allows to define domains in a non-generic way without loss of generality. A domain is a set of variables and therefore a collection of `Variable` objects. Clearly, there is no ordering of the domains' content and there are no double elements. The class `Domain` contains various methods that are typical for a set type. A complete description of these methods can be found in the attached Javadoc [Pou04].

Valuations:

The definition of the valuation interface content is straightforward because a valuation algebra is itself defined by the three basic operations of labeling, com-

bination and marginalization (see chapter 1). Here is the corresponding API of the `Valuation` interface:

- `public Domain label()`
- `public Valuation combine(Valuation val)`
- `public Valuation marginalize(Domain dom)`

This is a general representation of a valuation algebra based on its mathematical background. Often, instances of valuation algebras present additional properties and consequently we will have to refine the `Valuation` interface. The first additional property being fully implemented are scaled valuations. Probability potentials are shown to be an example of such a scaled valuation algebra.

Scaled Valuations:

To realize scaled valuations, the interface `ScaledValuation` is made available. This interface extends the `Valuation` interface with the following two methods:

- `public Object getScalingFactor()`
- `public void scale(Object scalingFactor)`

The first method computes the scaling factor of a given valuation. During local computation, this method will be called for the root node (see section 2.5). The second method scales a valuation with the formerly computed factor. The factor itself is represented as a JAVA top-level object to avoid restriction onto its content.

Printing Valuations:

Our interest in computations on join trees and query processing is based on the assumption that resulting valuations can be printed out. The corresponding output can either be in a text-based format or as a computer-generated image (bar charts for example). To take this into consideration, we have to extend our interface-hierarchy by a possibility to distinguish different output formats. For abstraction purposes, we extended the valuation interface first by an empty interface called `Representable`. All valuations that are somehow representable graphically implement this interface. To specify the kind of representation concretely, we add interfaces for each representation type. In the current state of this architecture, there exists only text-based valuations represented by the interface `TextRepresentable`. It contains the following method:

- `public String getText()`

This method transforms the current valuation to an ordinary string.
The number of interfaces used in this context may seem to be exaggerated but it will allow to add any combination of properties to any valuation algebra instance. Additionally, any instance can have different representation strategies

and the join tree viewer presented in section 4.4 will manage them without user interventions.
All interfaces presented in this section together with the class `Domain` form the content of the package `valuationAlgebra`. Figure 4.1 gives an overlook of these interfaces and their relations.
Implementing the interfaces above has some classical pitfalls. The most important are listed in the following table:

1. Both methods `combine` and `marginalize` in the interface `Valuation` have to create new `Valuation` objects. During the collect phase for example, valuations are combined, marginalized and afterwards sent to the child node. But the original valuations will be reused again in the distribute phase. Consequently, the original valuations have to stay untouched during the computations of combination and marginalization.
2. Combination is commutative:
   `val1.combine(val2) = val2.combine(val1)`
3. The `hashCode()` and `toString()` methods of the `Variable` interface are already implemented in `java.Object`. Thus, one can implement `Variable` without overwriting these method. But this can cause serious problems in the join tree generation - so, do not let it slip your mind to overwrite them. Unfortunately, there is no way in JAVA to solve this well-known problem.

4.2 Framework for Local Computation

Not only valuation algebras have an abstract, underlying concept but in a certain sense also local computation. Chapter 2 is about the Shenoy-Shafer architecture being the most general architecture of local computation. But there are other well-known architecture types as for example the *HUGIN* architecture or *Lauritzen-Spiegelhalter* and it must be possible to integrate these other architecture types in the integral whole of our generic architecture. It should not stay unmentioned that the main motivation in this case is not to create a system allowing each user to implement its own architecture type. This would not make sense because the total number of such architectures is overlookable. But it should be possible to replace the current architecture in an application without changing existing code. The different architectures themselves distinguish in the appearance of the corresponding join tree and in the implementation of

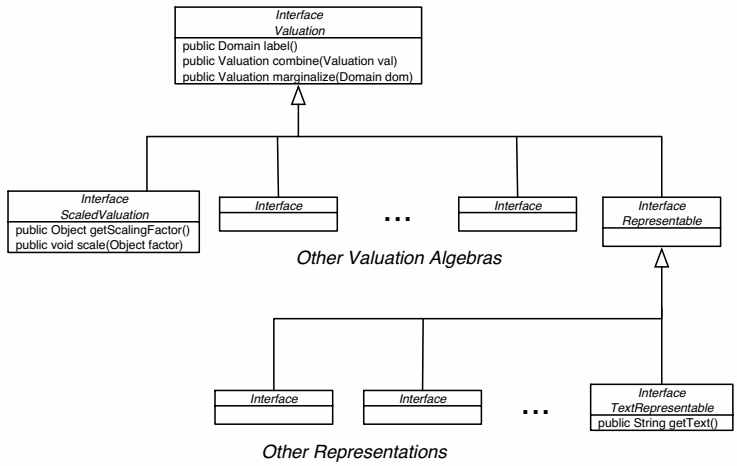


Figure 4.1: The interface hierarchy to represent valuation algebras.

their message passing algorithms. A possible strategy to receive the addressed goals is to define an overall join tree type which can later be adapted for each variant. This section describes our abstract join tree type based on an abstract variant of a join tree node. All these components are contained in the package `localComputation`.

Abstract join trees:

The abstract class `JoinTree` offers a general join tree type independent of its architecture. It is common to implement trees as linked nodes and this is exactly what has been done here. A `JoinTree` object has three major components:

- A pointer on its root node as usually for the implementation of a tree data structure.
- An array of pointers onto the leaves of the join tree. This useful extension was made because the collect algorithm starts by sending each leaf's valuation to its unique child and with this extension the leaves can be accessed directly.
- The elimination sequence as an array of type `Variable` having built this join tree. We introduced this field for educational reasons and its existence is not absolutely necessary.

These are the components defining an abstract join tree. For each of these components, getter and setter methods are pre-implemented and can be used directly without any extensions. Nevertheless, there are a few methods that have to be overwritten to define a valid join tree instance. These methods are:

- `public abstract String getArchitecture()`

- `public abstract void startPropagation()`
- `public static Node getNode()`

`getArchitecture()` returns a string describing the implemented architecture type. There are also educational reasons behind this method. We will come back to its use in the presentation of the graphical interface. The `startPropagation()` method starts the message passing algorithm. Its implementation is therefore unalienable. A very important position has the static method `getNode()`. Different architecture types use different node types, that is why the join tree creator method has to know the correspondence between node types and join tree architectures. For this purpose, each join tree returns an instance of the suitable node type by calling the `getNode()` method. Static methods can not be declared as being abstract. So, the `getNode()` method returns `null` by default and consequently, it is absolutely indispensable to overwrite this method. Another quite special method is

- `public void finalize()`

This method has an empty pre-implementation and can be overwritten if necessary. The `finalize()` method is the last method that will be called during the join tree generation. Overwriting this method allows the programmer to perform last actions on a finished join tree before returning the instance to the user. Similar *framework technics* are applied in the Java architecture itself as for example the `finalize()` method in the JAVA exception handling.
Implementing its own join tree class demands an important restriction. The constructor of this class must have an empty parameter list. The reason of this restriction can be found in the `JoinTreeFactory` class. As we will see, this class demands the architecture type and will then build join trees of this architecture automatically. To do this, it assumes an empty constructor of the appropriate join tree class. We will come back on this point when discussing the `JoinTreeFactory` class.

Abstract node type:

Each join tree contains a pointer onto its root node. So the skeleton of a join tree is basically a linked set of nodes. This is a very common tree implementation and for its use in the abstract local computation architecture we defined the `Node` interface. The following method API defines the structure of this node interface:

- `public List getParent()`
- `public void addParent(Node parent)`
- `public Node getChild()`
- `public void setChild(Node node)`

These are the most basic operations which allow to build a tree by a set of doubly linked nodes. In the context of local computation we use this tree type to realize join trees. Essentially, each node must contain a domain and a valuation storage. We add the following method set to guarantee its join tree suitability.

- `public Domain getDomain()`
- `public void setDomain(Domain domain)`
- `public Valuation getNodeValuation()`
- `public void setNodeValuation(Valuation val)`

At last, we are interested in a type allowing the implementation of local computation and logically, the node type must offer the following methods:

- `public void collect()`
- `public void distribute()`
- `public Valuation answerQuery(Domain domain)`

The `collect()` method runs the collect algorithm on the current node and applies itself recursively onto its child node. `distribute` runs the distribute algorithm onto the current node and applies itself recursively onto its parent nodes. At last, `answerQuery(Domain domain)` searches the highest occurrence of the given domain in the sub-tree of this node (the tree whose root is the current node) and returns the corresponding valuation.

N-ary nodes:

Regarding the former defined node method set, it is likely that most of these methods will be implemented independently of their later use. To take this into account, we defined an intermediate node type offering pre-implementations of the first method list. The corresponding abstract class is named `NaryNode` and represents an n-ary node with all the above methods pre-implemented, except the following:

- `public void collect()`
- `public void distribute()`

The watchful reader may scrutinize the right to exist of the `Node` interface - or in other words, why not taking the `NaryNode` as top level class of the node hierarchy. The answer to this question is of a more technical nature. Multiple inheritance is not allowed in JAVA, but implementing more than one interface is unproblematically. Defining a node interface, it is left up to the programmer to extend his own perhaps already existing and more sophisticated node class. If no other pre-implementation is present, the programmer can extend the abstract class `NaryNode` that offers already a fully implemented n-ary node type.

VVLL:

Constructing join trees starts with the realization of the VVLL and its use is widely independent of any architectures of local computation because we specify neither the kind of join tree nor the kind of node we use in this context. All components to build a VVLL are part of the package `localComputation.vvll`. These are essentially the following JAVA classes:

- `VariableValuationLinkList`
- `VariableContainer`
- `ValuationContainer`
- `Heap`

Further explications of these components are not necessary because they correspond exactly to the building parts of the VVLL described in section 3.1. For a detailled method API we refer to the Javadoc ([Pou04]).

JoinTree Factory:

The class `JoinTreeFactory` is the core class of the generic architecture and puts all these concepts together. Its main task is the construction of join trees. Calling the constructor

- `public JoinTreeFactory(String architecture)`

results in the creation of a factory to build join trees of the architecture type determined by the constructor's argument. This string argument is the class name of the join tree class corresponding to the demanded architecture type. Once the factory is created, we can build new join trees of this architecture by calling the method

- `public JoinTree create(Valuation[] vals, Domain[] queries)`

and the factory will return such a join tree object. Internally it calls the empty constructor of the appropriate join tree class. Of course, the initial set of valuations is an inalienable parameter. The second parameter allows to specify queries of interest and ensures that these queries can be answered (see section 3.2). The factory itself will construct a VVLL for this initial set of valuations and will build up the join tree by the algorithm we have studied in section 3.1. If the factory needs to create a new join tree node, it will call the `getNode()` method in the appropriate join tree class and gets a new node object suitable to its architecture. This guarantees right combinations of join tree and node types. As already mentioned, the last action before returning the join tree object is the call of method `JoinTree.finalize()` to perform last actions of the provided join tree object.

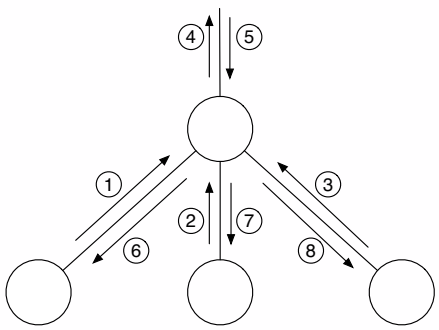
4.3 Realizing the Shenoy-Shafer Architecture

The first fully implemented architecture is the Shenoy-Shafer architecture and the goal of this chapter is to give an overlook of the implementation process. We will describe the concepts and their realization rather in general than giving a complete listing of the corresponding API. A detailed API description can be found in the attached Javadoc [Pou04]. All necessary classes are contained in the package `shenoy_shafer`.

Shenoy-Shafer join tree

The first step in defining a new architecture of local computation is the implementation of an appropriate node type. The fusion algorithm as described in section 3.1 creates an n-ary join tree and therefore we must also implement an n-ary node type to take this into account. But one thing is certain, using n-ary join trees during the collect and distribute phase results in an overhead of computation. Although this overhead can widely be dodged by memorizing temporary results, a neater solution is presented by the use of binary join trees. Figure 4.2 illustrates the additional overhead during the distribute phase for a join tree node with three parent nodes. The circled numbers stands for messages being sent from node to node. Messages 1 to 4 are sent during the collect phase, messages 5 to 8 during the distribute phase. The table on the right side of Fig. 4.2 lists the content of these messages. The overhead consists in repeated computations of combinations as for example the combination $\phi_1 \otimes \phi_2$ is computed first for message 4 and repeated in message 8. Of course the number of repetitions increases rapidly with a higher number of parent nodes.

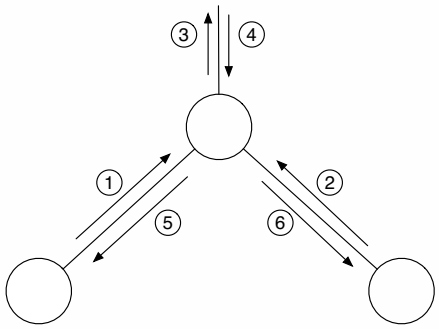
We consider now the same situation for binary join trees (join trees with bi-



Message:	Message content:
1	ϕ_1
2	ϕ_2
3	ϕ_3
4	$(\phi_1 \otimes \phi_2 \otimes \phi_3)^{\downarrow \star_1}$
5	ϕ_5
6	$(\phi_5 \otimes \phi_2 \otimes \phi_3)^{\downarrow \star_2}$
7	$(\phi_5 \otimes \phi_1 \otimes \phi_3)^{\downarrow \star_3}$
8	$(\phi_5 \otimes \phi_1 \otimes \phi_2)^{\downarrow \star_4}$

Figure 4.2: Illustration of the computing overhead with n-ary join trees.

nary nodes) illustrated in Figure 4.3. The table on the right of Fig. 4.3 shows quickly that in this case there is no repetitive computation anymore. More informations about the advantages of using binary join trees can be found in [Hae02]. To sum it up, we have the following situation. The fusion algorithm constructs n-ary join trees and therefore, we have to implement a corresponding



Message:	Message content:
1	ϕ_1
2	ϕ_2
3	$(\phi_1 \otimes \phi_2)^{\downarrow\star_1}$
4	ϕ_4
5	$(\phi_4 \otimes \phi_2)^{\downarrow\star_2}$
6	$(\phi_4 \otimes \phi_1)^{\downarrow\star_3}$

Figure 4.3: No computing overhead for binary join trees.

n-ary node type. But on the other hand, we know that computing with n-ary nodes results in a computing overhead and that the use of binary join trees would be a straightforward way out of this problem. The realization of this situation is easier as one could think. We will define an n-ary node type to construct the join tree. After that, just before running the collect algorithm, we ensure artificially that each node has two or less parent nodes. This can be achieved by introducing new nodes as we will see later. The consequence is that we get binary join trees. The n-ary nature of the nodes is hidden within the implementation and exists only as long as the join tree construction process has not finished yet.
The class `BSS_Node` puts this strategy into action. This class extends the abstract class `NaryNode` allowing the construction of n-ary nodes. To realize the transformation toward a binary sub-tree, the method

- `public void makeBinary()`

has been implemented. As already mentioned, the goal of this method is to ensure that each node has at most two parent nodes. This can be reached by introducing additional nodes and Figure 4.4 illustrates the transformation of a node with five parents to a binary tree. There are various possibilities to do such transformations in practice, but it is recommendable to create a more or less balanced tree. Here is the recursive procedure used in the `makeBinary()` method, we remember that each node has a list of pointers to its parent nodes and a pointer to its unique child node.

```
[01]  while (parents.size > 2) {
[02]        pairs = parents.size quotient 2;
[03]        for (i = 0; i < pairs; i++) {
[04]              node = new Node();
[05]              pa_1 = parents.remove(i);
[06]              pa_2 = parents.remove(++i);
[07]              node.addParent(pa_1);
[08]              node.addParent(pa_2);
[09]              node.setChild(this);
[10]              this.addParent(node, parents.size);
[11]              node.setDomain(pa_1.getDomain() ∪ pa_2.getDomain());
[12]        }
[13]  }
```

As long as the current node has more than two parents, this algorithm does the following steps: First it counts the number of pairs (line [02]). For each pair it creates a new node (line [04]), removes two neighboring pointer from the parent list (lines [05] and [06]) and adds them as parents to the new node. The new node itself is added at the last position to the parent list and gets the union domain of its parent nodes' domains to satisfy the Markov property (line [11]). This procedure will be executed for each node in the join tree from root to leaves.
Once the transformation is terminated, it is ensured that each node has less than three parents and they are accessible via the following methods:

- `public BSS_Node getFirstParent()`
- `public BSS_Node getSecondParent()`

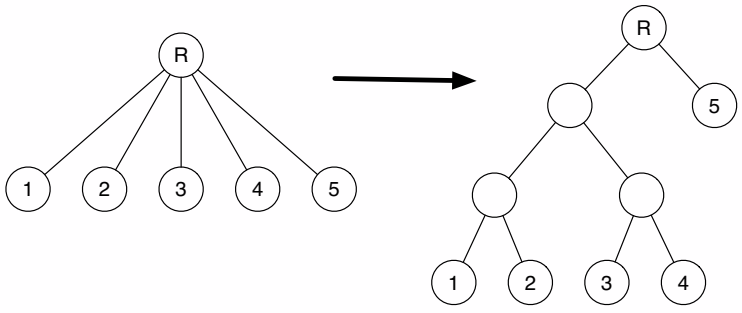
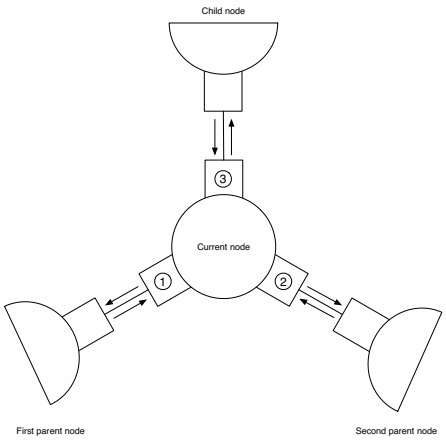


Figure 4.4: Transforming an n-ary node to a binary tree.

Having solved these problems, we are motivated to think about the interior structure of binary nodes. We have seen that the message passing algorithms demand the realization of mailboxes and the question arises, how the concept

of mailboxes can be implemented efficiently. A first possibility is to attach a mailbox object on the edge between two neighboring nodes in the join tree. Such an implementation is nice, because it realizes exactly the mathematical concept of message passing. On the other hand, it leads to an increase of objects without further functionalities because the only sense of a mailbox is to store messages. Another disadvantage of this solution is that it destroys the idea of creating join trees by linking homogenous nodes together. A quite better solution is to attach these storages directly on the nodes. Figure 4.5 shows a sketch of such a node. Thought in JAVA, we have to extend each node with three additional fields of type `Valuation` representing the mailboxes. The number "three" is determined by the use of binary nodes. To access the mailboxes of a given node,



1 First parent mailbox

2 Second parent mailbox

3 Child mailbox

Figure 4.5: A binary node with attached mailboxes.

we define the following two methods:

1. `public void receiveMessageFromParent(Valuation val, Node parent)`
2. `public void receiveMessageFromChild(Valuation val)`

These methods are called as soon as a node sends a message to one of its neighbors. Sending a message to a parent node, the receiving node must be able to identify the sender, this explains the second parameter of method (1). The message itself is represented by a `Valuation` object.
To complete the discussion about a suitable node type for the Shenoy-Shafer architecture, we must implement the message passing algorithm itself namely `collect()` and `distribute()`. The facts that we work on binary nodes and that the mailboxes are within the nodes themselves simplify this intention. Indeed, it is sufficient to translate formula (2.13) and theorem (14) in section 2.4 for the current node and to apply them recursively on all neighbor nodes to implement the two methods `collect()` and `distribute()`.

Having defined the class `BSS_Node` the implementation of a corresponding binary

Shenoy-Shafer join tree is straightforward and offered by the class `BSS_JoinTree`. This class extends the abstract class `JoinTree` as follows:

- The constructor of the class `BSS_JoinTree` is empty. This is a claim from the factory class.
- The method `getArchitecture()` returns the string "shenoy-shafer".
- We already described the role of the method `finalize()`. In this case, the last action before terminating the join tree construction is to apply the `makeBinary()` method on its root node.
- The static method `getNode()` returns a new `BSS_Node` object and defines therefore the suitable node type for this kind of join tree.
- The `startPropagation()` method causes all leaves to send their messages and calls the `collect` method on these nodes. As soon as the collect phase is terminated, it causes the root to send its message to all its parents and applies `distribute()` on them.

This closes the implementation of the Shenoy-Shafer architecture.

4.4 Join Tree Viewer

This section deals with an additional tool whose purpose is to display join trees graphically. Our abstract concept of local computation defines a join tree as a linked set of nodes and the nodes themselves by a small set of methods. Therefore, based on these node methods, this tool can display any kind of join trees extending the abstract join tree class above. The tool itself is called *TreeViewer* and its implementation is contained in the package `localComputation.graphics`. This section is restricted on the description of the package's main class `TreeViewer` and the use of the graphical user interface. Calling the constructor of the class `TreeViewer`

- `public TreeViewer(JoinTree joinTree)`

opens a new JAVA `Swing` window and displays the join tree given as argument to the constructor. Each join tree node is represented by a rectangle together with the corresponding node domain. Clicking the mouse onto one of these rectangles causes a pop-up window to display the current node's valuation. Of course, this behavior takes the implementation of a sub-interface of `Representable` for granted.
The "JoinTree" menu in the menu bar of the main window allows to get additional informations about the join tree currently displayed such as the elimination sequence that generated this join tree or its architecture type. The menu "File" offers to save the image of the currently displayed join tree as a *.png* file. Figure 5.2 shows an image that has been created this way.

Chapter 5

A first Instantiation

Instances of valuation algebras are known to be present in large numbers. Famous examples are Gaussian potentials, relations and many applications in the field of propositional logic. In the theory part we studied the example of probability potentials in detail and this will also be our first realized instance. Once we dispose of such an implementation, we will use it together with the Shenoy-Shafer architecture to compute explicitly the medical example introduced in section 1.3.

5.1 Implementing Probability Potentials

Probability potentials will serve as the first example of instantiating a valuation algebra. The mathematical background can be found in section 1.2 and the implementation itself is closely related to the definitions and representations in this section. This means that the reader finds a proper realization of the mathematical concept but on the other hand, no optimization has been carried out. We recommend therefore to consider it as an educational realization rather than as a piece of software to do expensive computations. This section will guide through the implementation of the two necessary interfaces namely `Variable` and `Valuation`.

Implementing the Variable interface:

For our purposes it is advisable to follow the most obvious way to represent a variable, that is by a simple string. We create a new class named `StringVariable` extending the `Variable` interface which has a string instance field representing the variable's name. Besides this, each variable has a frame with all its possible values. This can most easily be represented as a string array. To sum it up, we have the following constructor to create a new `StringVariable` instances:

- `public StringVariable(String name, String[] frame)`.

The remaining methods of the `Variable` interface are trivial to implement because it is sufficient to forward the corresponding methods of the JAVA `String`

class.

Implementing the Valuation interface:

In section 1.2 we represented probability potentials by simple tables. Each table entry corresponds to a possible configuration of the variable set equipped with a probability. That is exactly what the class `Potential` will do. This class extends the `ScaledValuation` interface and provides the following constructor:

- `public Potential(StringVariable[] vars, double[] probs)`.

The first argument lists all variables contained in the domain of this potential. It is not necessary for the user to enter explicitly each configuration. They can be generated automatically because each variable knows its frame and the total number of configurations can be computed by formula (1.2). This number of total configurations is furthermore the expected size of the array given as second argument. This array contains a probability for each configuration. To know which probability corresponds to which configuration, the constructor builds first the configuration table by ordering the configuration in the standard way, then it attaches probability i to table entry i. The mapping between configurations and probabilities depends therefore on the order of the variable and probability array given as argument. We will illustrate the construction of potentials later in this section.
Representing probability potentials explicitly as tables, the operations of combination and variable elimination are reduced to simple search procedures. When the search is terminated, the probabilities of the resulting configurations are either summed up to compute a variable elimination or multiplied for the combination. At last, the table of the new probability potential object is generated. This explains already the implementation of the methods `combine()` and `marginalize(Domain dom)`, the marginalization itself is implemented as a succession of variable eliminations.
We decided that the output-format of this implementation is a simple table listing all configurations and the corresponding probabilities. Hence, the `Potential` class implements the `TextRepresentable` interface.

Example: To close this section, we will do some finger exercises and repeat the computations from section 1.2 with our implementation of probability potentials. The following JAVA code is needed to perform the corresponding computations:

```
// Create a binary frame:
String[] frame = new String[] {"0", "1"};

/**
 * Creating Variables:
 */

StringVariable x = new StringVariable("X", frame);
StringVariable y = new StringVariable("Y", frame);
StringVariable z = new StringVariable("Z", frame);

/**
```

```
 * Creating Potentials:
 */

StringVariable[] vars1 = new StringVariable[] {x, y};
double[] probs1 = new double[] {0.1, 0.3, 0.4, 0.2};
Potential p1 = new Potential(vars1, probs1);

StringVariable[] vars2 = new StringVariable[] {y, z};
double[] probs2 = new double[] {0.3, 0.2, 0.1, 0.4};
Potential p2 = new Potential(vars2, probs2);

/**
 * Calculations & output:
 */

// Print potentials:
System.out.println("Potentials:\n");
System.out.println(p1.getText());
System.out.println(p2.getText());

// Labeling operation:
System.out.println("Labeling:\n");
System.out.println("p1.label() = "+p1.label());
System.out.println("p2.label() = "+p2.label()+"\n");

// Computing Combination:
System.out.println("Combination:\n");
Potential p3 = (Potential)p1.combine(p2);
System.out.println(p3.getText());

// Computing Marginalization:
Domain dom = new Domain(new StringVariable[] {x, y});
System.out.println("Marginalization of the combination onto dom = "+dom+"\n");
Potential p4 = (Potential)p3.marginalize(dom);
System.out.println(p4.getText());
```

This code results in the following output:

```
Potentials:

X Y Probability
0 0 0.1
0 1 0.3
1 0 0.4
1 1 0.2

Y Z Probability
0 0 0.3
0 1 0.2
1 0 0.1
1 1 0.4

Labeling:

p1.label() = {Y X}
p2.label() = {Z Y}

Combination:

X Y Z Probability
0 0 0 0.03
0 0 1 0.02
0 1 0 0.03
0 1 1 0.12
1 0 0 0.12
```

```
1 0 1 0.08
1 1 0 0.02
1 1 1 0.08

Marginalization of the combination onto dom = {Y X}

X Y Probability
0 0 0.05
0 1 0.15
1 0 0.2
1 1 0.1
```

A runnable version of this code is contained in the class `examples.Script`.

5.2 Experimenting with the Medical Example

As promised, we will demonstrate the power of this generic architecture by programming concretely the medical example introduced in section 1.3. This gradual guidance is based on the realization of the Shenoy-Shafer architecture (see section 4.3) and we will take probability potentials to model this network. The presented source code can be found in the class `examples.Asia`.

We start by creating the eight variables: A (visit to Asia), S (Smoking), T (Tuberculosis), L (Lung cancer), B (Bronchitis), E (Either tuberculosis or lung cancer), X (positive X-ray) and D (Dyspnoea). Each of these variables has a binary frame representing *true* and *false*.

```
// There are only binary variables:
String[] frame = new String[] {"0", "1"};

/**
 * The variables:
 */

// Visit to Asia:
StringVariable a = new StringVariable("A", frame);
// Tuberculosis:
StringVariable t = new StringVariable("T", frame);
// Smoking:
StringVariable s = new StringVariable("S", frame);
// Bronchitis:
StringVariable b = new StringVariable("B", frame);
// Lung cancer:
StringVariable l = new StringVariable("L", frame);
// Either tubercolosis or lung cancer:
StringVariable e = new StringVariable("E", frame);
// Positive X-ray:
StringVariable x = new StringVariable("X", frame);
// Dyspnoea:
StringVariable d = new StringVariable("D", frame);
```

The next step consists in modeling the knowledge-base. For each of the eight valuations, we will create a new probability potential. The corresponding probabilities are taken from [LS88] and listed in table 5.1. We create the potentials by the following code:

```
/**
 * Potentials:
 */
```

$p(\alpha)$	=	0.01	$p(\epsilon\|\lambda,\tau)$	=	1
			$p(\epsilon\|\lambda,\bar{\tau})$	=	1
$p(\tau\|\alpha)$	=	0.05	$p(\epsilon\|\bar{\lambda},\tau)$	=	1
$p(\tau\|\bar{\alpha})$	=	0.01	$p(\epsilon\|\bar{\lambda},\bar{\tau})$	=	0
$p(\sigma)$	=	0.5	$p(\xi\|\epsilon)$	=	0.98
			$p(\xi\|\bar{\epsilon})$	=	0.05
$p(\lambda\|\sigma)$	=	0.1			
$p(\lambda\|\bar{\sigma})$	=	0.01	$p(\delta\|\epsilon,\beta)$	=	0.9
			$p(\delta\|\epsilon,\bar{\beta})$	=	0.7
$p(\beta\|\sigma)$	=	0.6	$p(\delta\|\bar{\epsilon},\beta)$	=	0.8
$p(\beta\|\bar{\sigma})$	=	0.3	$p(\delta\|\bar{\epsilon},\bar{\beta})$	=	0.1

Table 5.1: Probabilities used in the medical example.

```
StringVariable[] vars1 = new StringVariable[] {a};
double[] probs1 = new double[] {0.99, 0.01};
Potential pot1 = new Potential(vars1, probs1);

StringVariable[] vars2 = new StringVariable[] {t, a};
double[] probs2 = new double[] {0.99, 0.95, 0.01, 0.05};
Potential pot2 = new Potential(vars2, probs2);

StringVariable[] vars3 = new StringVariable[] {s};
double[] probs3 = new double[] {0.5, 0.5};
Potential pot3 = new Potential(vars3, probs3);

StringVariable[] vars4 = new StringVariable[] {l, s};
double[] probs4 = new double[] {0.99, 0.9, 0.01, 0.1};
Potential pot4 = new Potential(vars4, probs4);

StringVariable[] vars5 = new StringVariable[] {b, s};
double[] probs5 = new double[] {0.7, 0.4, 0.3, 0.6};
Potential pot5 = new Potential(vars5, probs5);

StringVariable[] vars6 = new StringVariable[] {e, l, t};
double[] probs6 = new double[] {1, 0, 0, 0, 0, 1, 1, 1};
Potential pot6 = new Potential(vars6, probs6);

StringVariable[] vars7 = new StringVariable[] {x, e};
double[] probs7 = new double[] {0.95, 0.02, 0.05, 0.98};
Potential pot7 = new Potential(vars7, probs7);

StringVariable[] vars8 = new StringVariable[] {d, e, b};
double[] probs8 = new double[] {0.9, 0.2, 0.3, 0.1, 0.1, 0.8, 0.7, 0.9};
Potential pot8 = new Potential(vars8, probs8);

// Store all potentials in an array:
Potential[] pots = new Potential[] {pot1, pot2, pot3, pot4, pot5, pot6, pot7, pot8};
```

Let us assume that we want to know the probability that the patient suffers from bronchitis. To ensure that our join tree will have a node answering this question we will add artificially the following domain:

```
/**
 * Queries:
 */

Domain query = new Domain(new Variable[] {b});

// Store all queries in an array:
Domain[] queries = new Domain[] {query};
```

To construct a Shenoy-Shafer join tree based on the knowledge-base above, we will first create a new factory instance being able to build join trees of the type "shenoy-shafer". Then, this factory delivers a join tree object.

```
/**
 * Computing and output:
 */

// Create a join tree factory. Its parameter is the join tree architecture class.
JoinTreeFactory factory = new JoinTreeFactory("shenoy_shafer.BSS_JoinTree");

// Create the join tree.
JoinTree jointree = factory.create(pots, queries);
```

The following code line will start the collect and distribute algorithm.

```
// Start collect and distribute algorithm.
jointree.startPropagation();
```

To get the answer to our query, we demand the valuation of the corresponding join tree node.

```
// Answer the query:
Potential answer = (Potential)jointree.answerQuery(query);
System.out.println(answer.getText());
```

The returned answer to the query is:

```
// Query answer:

B Probability
0 0.5499999999999998
1 0.4499999999999999
```

Analogous to the query above, we can formulate other queries for each variable. Figure 5.1 shows the corresponding (rounded) results in the valuation network.

In section 4.4 we described an additional tool named "JoinTree Viewer" to display join trees graphically. To start the viewer, type the following command. The resulting picture is shown in Figure 5.2.

```
// Display the join tree:
new TreeViewer(jointree);
```

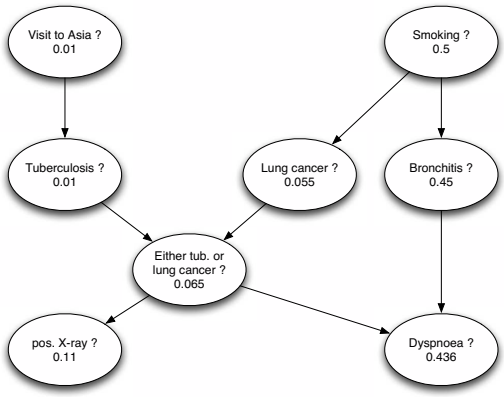


Figure 5.1: Query answers of the medical example.

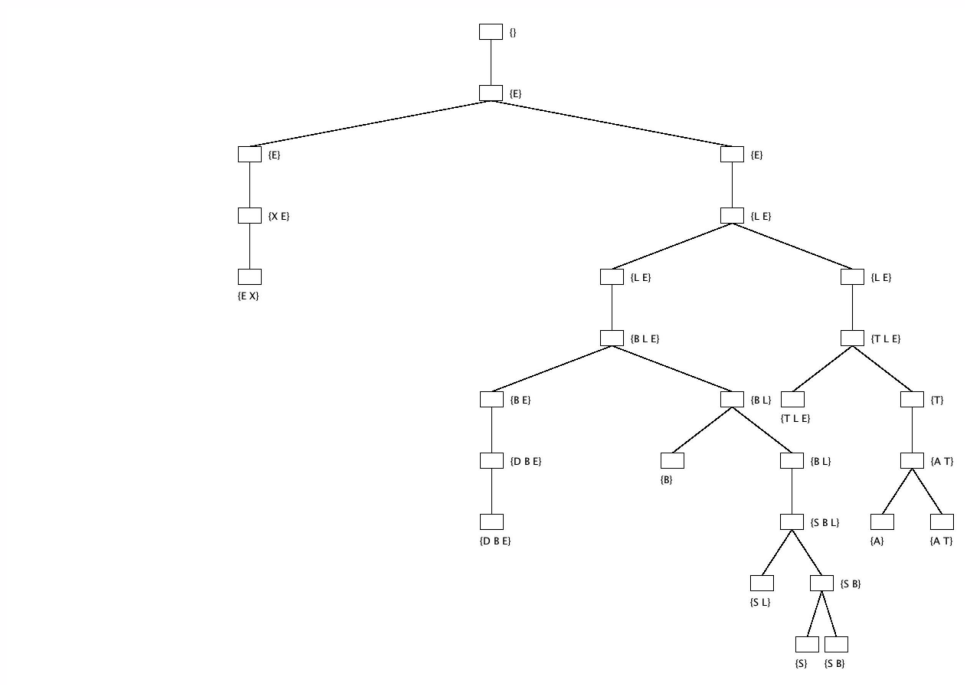


Figure 5.2: Output of the "TreeViewer" tool for the medical example.

Chapter 6

Outlook & Conclusions

Regarding valuation algebras as a common roof implies that local computation can be applied without specifying the underlying information structure. That is exactly what we have done in this software project. Initially, we defined variables and valuations by some small interfaces and these interfaces offer now an abstract description of all imaginable kinds of instances. This is more or less an exact implementation of the mathematical concept behind valuation algebras and opens the possibility to realize the local computation scheme upon this natural level of abstraction. But the goal of creating a computation architecture demanded more than this because a further layer of abstraction concerning local computation suggested itself. In the theory part of this report, we described the Shenoy-Shafer architecture in detail but having the existence of other architectures at the back of our mind leads to the request of creating a system allowing to replace architecture types without further restrictions. This was certainly the most challenging part of the programming work because we have to guarantee the needed flexibility to meet all requirements of the different architecture types. Nevertheless, these first steps will be trendsetting in the implementation of the other architecture types and it will be possible to switch between different architecture types without changing existing code, although it can not be excluded that the actual version does not satisfy all the needs of the architectures added in the near future.

This software project in its current state offers a framework allowing to include and combine instances of valuation algebras with different architecture types of local computation. Describing the further development process is therefore straightforward. We will implement on one hand a variety of valuation algebra instances to access the versatility of applications and on the other hand, we will implement all known architecture types to compare their way of performing local computation. Another point is optimization. Until now, we developed this architecture manly for educational reasons but investigating performance and optimization approaches will perhaps open a more extensive application field. A personal point of interest concerns parallel computing. Computations on join tree are well-suited to be distributed over multiple processors. One can imagine two ways to realize it: either horizontally, meaning that each node of the join

tree is assigned to a processor in such a way that all computation on one tree level is performed in parallel; or vertically, such that each processor is responsible for the computation in a certain sub-tree. Thinking in this direction we remark a positive implication of using the JAVA programming language. JAVA offers itself a lot of functionalities and facilities to do parallel computations with processor pools and applying it on join tree computations will perhaps result in new ideas about local computation in general.

Appendix A

Program Structure

The listing below summarizes the source code packages and the included classes:

- package `informationmachine`
  - package `localComputation`
    - package `vvll`
      - class `VariableValuationLinkList`
      - class `VariableContainer`
      - class `ValuationContainer`
      - class `Heap`
      - class `EmptyValuation`
    - package `graphics`
      - class `treeViewer`
      - class `Scroller`
      - class `DrawingPanel`
      - class `NodeShape`
      - class `InfoFrame`
      - class `AboutFrame`
      - final class `Constants`
    - abstract class `JoinTree`
    - class `JoinTreeFactory`
    - interface `Node`
    - abstract class `NaryNode`
  - package `valuationAlgebra`
    - interface `Variable`
    - interface `Valuation`
    - interface `ScaledValuation`
    - interface `Representable`
    - interface `TextRepresentable`
    - class `Domain`

  - class `ExceptionHandler`

- package `shenoy_shafer`
  - class `BSS_JoinTree`
  - class `BSS_Node`

- package `probability_potentials`
  - class `StringVariable`
  - class `Potential`

- package `examples`
  - class `Asia`
  - class `Script`

Appendix B

The Big Picture

Figure B shows a class diagram of the whole architecture. To avoid an overloaded and confusing illustration, all method and field names of the classes are left out. Additionally, the content of the `graphics`-package is not listed explicitly because the user will never be confronted directly with this code. For the interested reader we refer to the detailed Javadoc [Pou04]. Additionally, only those arrows are drawn which stand for implemented interfaces (dashed lines) or extensions.

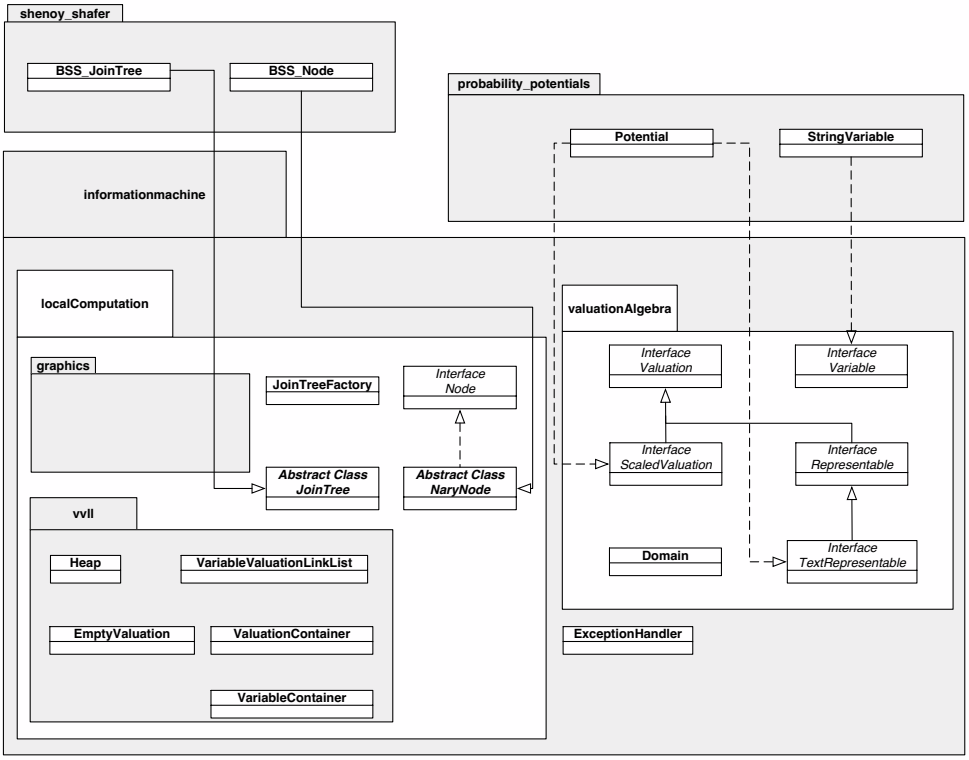


Figure B.1: A class diagram of the whole architecture.

Appendix C

Disc Content

- Directory *source* contains the complete source code of the implementation.
- Directory *executables* contains a compiled version for JDK 1.4.2_03.
- The file *javadoc.html* leads to the index-page of the Javadoc.
- Additionally, there are start-scripts (Windows & Unix) for both examples treated in the documentation and a *pdf*-version of the documentation.

Bibliography

[ACP87] S. Arnborg, D. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. Technical report, 1987.

[Bai98] D. A. Bailey. *Data Structures in JAVA for the principled programmer.* Mc Graw-Hill, 1998.

[Hae02] R. Haenni. Ordered valuation algebras: a generic framework for approximating inference. Technical report, 2002.

[JLO90] F. V. Jensen, S. L. Lauritzen, and K. G. Olesen. Bayesian updating in causal probabilistic networks by local computation. Technical report, 1990.

[Koh03] J. Kohlas. *Information Algebras: Generic Structures for Inference.* Springer-Verlag, 2003.

[Leh01] N. Lehmann. *Argumentation System and Belief Functions.* PhD thesis, Department of Informatics, University of Fribourg, 2001.

[LS88] S. L. Lauritzen and D. J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. Technical report, J. of Royal Stat. Soc., 1988.

[Pou04] M. Pouly. Javadoc. /javadoc.html, 2004.

[SS90] P. P. Shenoy and G. Shafer. Axioms of probability and belief-function propagation. In R. D. Shachter, T. S. Levitt, L. N. Kanal, and J.F. Lemmer, editors, *Uncertainty in Artificial Intelligence 4*, pages 169–198, Amsterdam, 1990. North-Holland.