Aus dem Departement für Informatik
Universität Freiburg (Schweiz)

A Generic Framework for Local Computation

Inaugural–Dissertation

zur Erlangung der Würde eines *Doctor scientiarum informaticarum*
der Mathematisch-Naturwissenschaftlichen Fakultät
der Universität Freiburg in der Schweiz

vorgelegt von

Marc Pouly

aus

Les Cullayes (Waadt)

Nr. 1603
UniPrint, Freiburg
2008

Von der Mathematisch-Naturwissenschaftlichen Fakultät der Universität Freiburg in der Schweiz angenommen, auf Antrag von:

- Prof. Dr. Ulrich Ultes-Nitsche, Universität Freiburg, Schweiz (Jurypräsident),
- Prof. Dr. Jürg Kohlas, Universität Freiburg, Schweiz,
- Prof. Dr. Rolf Haenni, Universität Bern, Schweiz,
- Prof. Dr. Jacques Pasquier-Rocha, Universität Freiburg, Schweiz.

Freiburg, 7. Mai 2008

Der Leiter der Doktorarbeit:

Prof. Dr. Jürg Kohlas

Der Dekan:

Prof. Dr. Titus Jenny

Acknowledgements

The time will come when I look back on this thesis with the hours of work forgotten. But even then, I will be grateful to the people that supported me during this period. I therefore dedicate my first words to all these people and apologize in the same breath to anyone I may have forgotten to name.

Foremost, I would like to thank Prof. Jürg Kohlas, the director of my thesis, who inspired my scientific curiosity and supported my daily work with many new ideas and valuable feedback. This also concerns Prof. Rolf Haenni, Prof. Jacques Pasquier-Rocha and Prof. Ulrich Ultes-Nitsche who accepted to act as referees for my thesis. Many thanks go further to my collaborators at the departments of informatics in Fribourg and Berne, especially to Bernhard Anrig, David Buchmann, Matthias Buchs, Reinhard Bürgy, Christian Eichenberger, Patrik Fuhrer, Michael Hayoz, Tony Hürlimann, Jacek Jonczy, Dominik Jungo, Reto Kohlas, Jutta Langel, Carolin Latze, Norbert Lehmann, Thierry Nicola, Cesar Schneuwly and Michael Wachter. They all contributed to a friendly and stimulating working ambiance. Further, I want to accent the very close and fruitful collaboration with Cesar Schneuwly which lead to the theory of identity elements developed in the first and third chapter, as well as the numerous meetings with Michael Wachter and Rolf Haenni which introduced me to the marvelous world of knowledge compilation and guided me through a jungle of abbreviations. The results of this collaboration are summarized in Chapter 8. I am also indebted to Antoine de Groote and Pierre Spring, whose useful comments helped me to improve the NENOK framework and to Margrit Brause for her help to ameliorate the language of this thesis.

I do not keep secret that the process of writing a PhD thesis also includes moments where a simple backslapping is of great value. For this support and for making my leisure time enjoyable and relaxing, my thanks go to all members of the badminton clubs of Murten and Düdingen, in particular to Rolf Allemann, Rolf Bertschy, Philippe Etter, Olivier Dürig and Didier Huguenot, but also to Philippe Derhovsepian, Sara Fiechter, Jens Uebersax, Nicole & Beat Rüttimann, Silvia & Ueli Stalder, David Stöckli, Ruth & Fritz Wieland, Ralph Wieland, Andy Zürcher, Judith Zürcher, Margrit & Walter Zürcher and to all my ultimate frisbee teammates. Finally, it is my greatest pleasure to express my gratitude to my family for their absolute support of everything I have done so far: to my parents Jeannette & Charly Pouly, to my sister Janique Pouly & Tobias Aebischer and to my grandparents Hilda & Ernst Leuthold and Hedy Pouly. My last cordial thanks are addressed to my girlfriend Marita Wieland for the fantastic time we spend together and for her absolute belief in my abilities.

Abstract

The distributive law known from arithmetics is one of the best tools for a computer scientist to grapple with the intractable nature of many applications. So, efficient algorithms have been developed for the computation of Fourier and Hadamard transforms, Bayesian networks, database queries, decoding problems and many more. The fact that they all benefit from the same technique suggests that a single generic algorithm must exist which solves all these problems efficiently. This essentially is the promise of local computation. The mathematical requirements for the application of local computation, which are clearly satisfied by the above examples, are pooled in an algebraic framework called valuation algebra. Moreover, this framework is general enough to be seen as an algebraic approach towards a definition of knowledge and information, since it perfectly reflects the adjudicated attributes of these notions. Our first contribution concerns a generalization of the valuation algebra framework that dispenses with the concept of neutral information. This solves the problem that in some formalisms, neutral elements do not have finite representations. Further, we also identify some valuation algebra instances that do not possess neutral information at all. From this generalized framework evolves the generic algorithm in the shape of the traditional local computation architectures. As an interesting side effect, we remark that some computations even become more efficient by the measures taken to renounce neutral information.

A particular subclass of valuation algebras that is extensively studied in this thesis comprises formalisms that map configurations to semiring values. Besides the usefulness of this class to discover new valuation algebra instances, it will also be shown that they give rise to optimization problems under certain conditions. This insight motivated the development of a dynamic programming scheme for the identification of solution configurations that itself exploits local computation. Another far more powerful algorithm for the same task excels through innovation. By use of local computation, solution configurations are compiled into a Boolean function which is well suited to answer a large number of queries efficiently, that go far beyond the pure enumeration of solution configurations.

In parallel to the theoretical considerations, we propose in this thesis a software framework containing generic implementations of the discussed algorithms. This system called NENOK is designed to be used as a service library for all kind of local computation related applications, but may also serve as an experimental workbench for educational purposes, since it provides a multiplicity of possibilities to inspect the local computation process. Additionally, it also takes an as yet disregarded aspect of information into consideration and realizes all local computation schemes as real distributed algorithms. This in turn raises new theoretical questions on how to ensure efficient communication during the computations, which are addressed theoretically and independently of the actual local computation architecture.

Zusammenfassung

Hinter vielen Anwendungen der Informatik verstecken sich rechnerisch höchst anspruchsvolle Aufgaben, welche auch von modernen Computern nur dank einer geschickten Anordnung der Operationen effizient behandelt werden können. Ein geeignetes Hilfsmittel dazu bietet das aus der Arithmetik bekannte Distributivgesetz. So wurden in der Vergangenheit effiziente Verfahren zur Berechnung von Fourier- und Hadamard-Transformationen, Bayesianischen Netzwerken, Datenbankanfragen oder auch Dekodierungsaufgaben entwickelt. Die Tatsache, dass all diese Methoden durch Anwendung der gleichen Technik gefunden wurden, lässt darauf schliessen, dass ihnen ein gemeinsamer generischer Algorithmus zugrunde liegt. Dies wird als lokales Rechnen bezeichnet. Die mathematischen Grundlagen, welche das lokale Rechnen ermöglichen, werden durch das axiomatische System der Valuationsalgebren definiert, welche natürlich auch die obigen Anwendungen abdeckt. Interessanterweise gibt eine solche Struktur alle Eigenschaften wieder, welche gemeinhin den Begriffen *Wissen* und *Information* zugesprochen werden. Daher kann sie auch als Ansatz zu einer algebraischen Definition dieser Begriffe verstanden werden. Der erste Teil dieser Arbeit beschreibt eine verallgemeinerte Valuationsalgebra, welche auf das Konzept von neutraler Information verzichtet. So werden bekannte Probleme mit Formalismen gelöst, deren neutrale Elemente keine endliche Darstellung besitzen. Zusätzlich identifizieren wir neue Instanzen, die überhaupt keine solchen Elemente aufweisen. Basierend auf diesem System wird der generische Algorithmus in Form der bekannten Architekturen entwickelt. Dabei bemerken wir, dass durch den Verzicht auf neutrale Information gar eine Effizienzsteigerung im Algorithmus erreicht wird.

Eine interessante Klasse von Valuationsalgebren beinhaltet Formalismen, die durch eine Abbildung von Konfigurationen auf Semiringwerte charakterisiert sind. Diese Sichtweise erleichtert das Auffinden neuer Instanzen, und es wird gezeigt, dass Semiringvaluationen unter gewissen algebraischen Bedingungen Optimierungsprobleme induzieren. Dies motiviert die Entwicklung eines auf lokalem Rechnen beruhenden Algorithmus zum Auffinden von Lösungskonfigurationen – eine Problemstellung, die gewöhnlich als dynamische Programmierung bezeichnet wird. Alternativ dazu wird ein zweiter Ansatz verfolgt, der durch eine geschickte Verbindung von lokalem Rechnen mit der Theorie der Wissensdarstellung zu einer Methode führt, welche Lösungskonfigurationen nicht explizit auflistet, sondern in eine Boolsche Funktion kompiliert. Durch Anwendung bekannter Abfragetechniken kann diese Funktion dann auf Fragestellungen evaluiert werden, welche die einfache Auflistung von Lösungskonfigurationen bei weitem übersteigt.

Abschliessend wird die Software-Architektur NENOK mit generischen Implementationen der erwähnten Algorithmen präsentiert, wobei zwei Anwendungsszenarien als Programmbibliothek und als Experimentierplattform im Vordergrund stehen. Darüber hinaus ist NENOK als verteilte Applikation konzipiert, was neue Fragen betreffend effizienter Kommunikation aufwirft. Diese werden auf theoretischer Basis und unabhängig von der Architektur des lokalen Rechnens diskutiert.

Contents

List of Figures

Listings

1
Introduction

Designing algorithms is traditionally a problem-oriented process. Based upon a specification of some computational problem and its input, an algorithm must be developed that recognizes the input and delivers the required result. From an economical point of view, this approach is thought to produce an acceptable algorithm with minimum development time effort. However, a direct consequence is that the resulting procedure only applies to the problem that initially motivated its evolution. This in turn leads to the prevalent situation that strong algorithms are reinvented for every problem that appears to be different at first sight. To corroborate this statement, we consider the following three epoch-making algorithms: The *fast Fourier transform* deals with any kind of signal processing task such as image compression. Second, the *Bayesian network algorithm* is used for probabilistic inference and in particular to automate medical diagnoses. Finally, the *query answering algorithm* is executed whenever we ask for information out of a relational database. All three algorithms at first glance apply to very different formalisms and solve semantically very different problems. This makes it all the more astonishing that they indeed pertain to the same *generic algorithm*.

Let us delve deeper into this idea of generic algorithms by the following train of thought. Consider the plain task of writing a program to sort a list of items. Clearly, it is beyond all question to reinvent such a sorting procedure for lists containing numbers, words, colors or even bank accounts. In the same tenor, we do not need fundamentally different algorithms to bring the list in either ascending or descending order. Moreover, if we understand the mathematical requirements for a sorting procedure, namely the total order between list elements, we easily develop a single generic version of a sorting algorithm that allows for all the previously mentioned scenarios. This extract from every programmer's basic training highlights the concrete worth of generic algorithms in general. Treating connatural problems by a single method is preferable for many reasons such as code maintenance or optimization. In addition, the unification of existing procedures under the common roof of a generic algorithm facilitates the transfer of research results. With respect to the above three evergreen algorithms, we may for example improve the processing of Bayesian networks by use of techniques coming from database theory. Finally,

if a new formalism is proposed for some purposes, it is then sufficient to verify the mathematical requirements of the generic algorithm, in order to access a suitable and efficient procedure for its treatment. This eliminates the development phase and in a sense revolutionizes the classical design process for algorithms.

Evidently, the study of generic algorithms demands a fine grasp of its mathematical requirements, which are generally stated in the shape of an axiomatic system. In the case at hand, the presence of two basic operations and six axioms admits the application of the generic algorithm that underlies the above three procedures. This framework called *valuation algebra* furthermore mirrors all natural properties we generally associate with *knowledge* or *information*. Therefore, people rightly regard this system as an algebraic approach towards a definition of these two notions. We adopt this perspective as the starting point for the following listing that outlines some of the catchwords of this thesis, whereas we postpone the delineation of our contributions to Section 1.1.

Knowledge & Information

Giving satisfying definitions of *knowledge* and *information* is an ambitious task. There are nowadays various approaches that describe these notions either quantitatively, semantically or even algorithmically. Our excursion into the necessity of generic algorithms leads to an algebraic conception that arises from a couple of very simple and natural observations: Knowledge and information exist in pieces, and every piece refers to some questions. Different pieces can be aggregated, but also focused on the relevant questions, which corresponds to the extraction of those parts of a knowledge piece that complies with our interest. These two operations, together with six axioms that describe their behaviour, assemble the valuation algebra framework that constitutes our perception of knowledge. Information on the other hand requires an additional property of the aggregation operator which states that a piece of information combined with a part of itself gives nothing new. This natural property draws the distinction between the two concepts. Systems that satisfy the requirements for a valuation algebra are very large in number and include most formalisms that people tend to use for the expression of knowledge and information. Most famous are probability mass functions and relations, but valuation algebras also cover constraint systems, possibility functions, Dempster-Shafer theory, propositional and predicate logic, density functions, systems of linear equations and inequalities, and many more. As a matter of course, the formalisms that are processed by the above three algorithms can also be found among them.

Inference

Inference constitutes the practical value of every system for knowledge representation. From the perspective of valuation algebras, this process consists in the aggregation of all available knowledge pieces, in order to focus the result afterwards on the actual questions of interest. This computational task, known as the *projection problem*, is stated in a purely generic way thanks to the introduced valuation algebra

language. We anticipate that the task of a Fourier transform, as well as a Bayesian network or a database query are all possible instances of such projection problems. Therefore, we chance to conclude that all three applications can be attended to by a single inference algorithm that solves any kind of projection problem. However, the design of such an algorithm demands a great deal as shown in the following example: Consider three non-negative, real-valued functions $f(X_1, X_2)$, $g(X_2, X_3)$ and $h(X_2, X_4, X_5)$, where $X_1, \ldots, X_5$ are variables taking values from a finite set with n elements. Clearly, such functions can be represented in tabular form, which leads to tables with n^2 entries for f and g, and a table with n^3 entries for the function h. If we next identify aggregation with component-wise multiplication and focussing with the summation of unrequested variables, we may define the following projection problem:

$$\sum_{X_3,X_4,X_5} f(X_1, X_2) \cdot g(X_2, X_3) \cdot h(X_2, X_4, X_5). \tag{1.1}$$

If we tackle this computational task by first executing all multiplications, we obtain an intermediate result of size $n^2 \cdot n^2 \cdot n^3 = n^7$. More generally, if functions over m different variables are involved, an intermediate table of n^m entries is produced. Clearly, this process becomes intractable even for up-to-date computers and a moderate number of variables. A well-tried strategy to avoid this complexity problem applies the distributive law to arrange the computations in such a way that all intermediate results stay manageable. Thus, we may rewrite Expression (1.1) as:

$$f(X_1, X_2) \cdot \left(\sum_{X_3} g(X_2, X_3) \right) \cdot \left(\sum_{X_4,X_5} h(X_2, X_4, X_5) \right).$$

Here, the largest intermediate result has only n^2 entries, which is a considerable increase of efficiency. There are essentially two mathematical properties which afford this performance boost. Besides the already mentioned distributive law, we exploit the fact that variables can be summed up in any order. In effect, all three algorithms that accompany us through this introduction benefit from slightly more general versions of these two laws. They are both contained in the valuation algebra axioms and enable the development of the generic inference algorithm.

Dynamic Programming

In the above example, we identified knowledge extraction with the summation of unrequested variables. Alternatively, we could also take the maximum value over these variables and write the computational task as

$$\max_{X_3,X_4,X_5} f(X_1, X_2) \cdot g(X_2, X_3) \cdot h(X_2, X_4, X_5). \tag{1.2}$$

First, we remark that the two expressions (1.1) and (1.2) look very similar which suggests once more their treatment as instantiations of a single task. More concretely, both expressions model projection problems over different valuation algebras. Additionally, the same complexity considerations with respect to the size of intermediate

tables apply, and because the distributive law still holds between multiplication and maximization, we may also perform the same improvement:

$$f(X_1, X_2) \cdot \left(\max_{X_3} g(X_2, X_3) \right) \cdot \left(\max_{X_4, X_5} h(X_2, X_4, X_5) \right).$$

This again motivates the development of a single generic algorithm for the solution of arbitrary projection problems. However, this algorithm will be designed to compute values, but in case of optimization problems as (1.2), we are additionally interested in the configurations that adopt the maximum value. In other words, we need to identify the variable assignment that leads to the optimum value. Such variable assignments are habitually called *solution configurations*, and their identification corresponds to the classical task of *dynamic programming*, whose solution goes beyond the scope of the generic inference mechanism. Nevertheless, existing algorithms for this task also benefit from the same technique, namely from the application of the distributive law, which motivates the development of a second generic algorithm for this new job.

Processing Distributed Knowledge

An as yet unmentioned aspect is the distributed nature of knowledge and information. It is an everyday experience that knowledge pieces come from different sources. They are shared and exchanged between processing units, which in turn suggests that inference mechanisms should be realized as distributed algorithms. The above analysis of the projection problem sensitizes the reader to the necessity of complexity considerations when dealing with valuation algebras, and it is not astonishing that the same concerns also affect the efficiency of communication in a distributed system. Therefore, efficient communication must be declared a central topic for the development of distributed inference algorithms.

NENOK

Another hot topic of this thesis is a software framework called NENOK that offers generic implementations of all presented algorithms. It is realized using an object-oriented programming language and provides an abstract representation of the valuation algebra framework. By instantiating this framework, users specify their own valuation algebra instances and use the predefined inference mechanisms for processing. In addition, NENOK has a communication infrastructure to set up processor networks. Each processor manages a shared memory space which allows to exchange valuation objects between them. The inference algorithms themselves are implemented in a highly transparent way and do not distinguish between local and remote data. Apart from its role as inference routine library, NENOK also qualifies for use as an experimental workbench. In fact, it possesses a large number of tools to inspect the inference process and the involved graphical structures. The information is accessible through a sophisticated user interface that is especially well suited for educational purposes.

1.1 Purpose & Contribution

The axiomatic framework of a valuation algebra first appeared in (Shenoy & Shafer, 1988) and was later revised by (Shafer, 1991) and (Kohlas, 2003). Here, we propose an essential generalization of this system that dispenses with the inclusion of neutral elements. The corresponding theory emerged as a collaborative work with Cesar Schneuwly and is therefore also developed in (Schneuwly, 2007). It will then be shown rigorously that this more general version still possesses enough structure for the execution of local computation. Of prime importance for this purpose is the concept of a covering join tree that furthermore has a positive impact on the efficiency of local computation in general. The four major local computation architectures called Shenoy-Shafer (Shenoy & Shafer, 1990), Lauritzen-Spiegelhalter (Lauritzen & Spiegelhalter, 1988), HUGIN (Jensen *et al.*, 1990) and Idempotent Architecture (Kohlas, 2003) are adapted to this more general setting, including their modifications to compute scaled results. Together with these algebraic and algorithmic results, we come up with an unprecedented catalogue of instances that also include formalisms where neutral elements are either not representable or do not exist at all. This shows the necessity of the above generalizations. In addition, we also introduce the formalism of *distance potentials* that has so far never been described in the context of valuation algebras.

This introduction already touched upon the distributed nature of knowledge and information, which motivates the ambition to process valuations in a distributed manner. (Shenoy & Shafer, 1990) already remarked that local computation architectures qualify for an implementation as distributed algorithms. This basic idea is further developed with a special focus on efficient communication. It turns out that under certain algebraic conditions, the underlying covering join tree may act as an overlay network for the distribution of processing units, which allows indeed to minimize communication costs with low polynomial effort.

A particular valuation algebra subclass pools formalisms that map configurations to values out of a commutative semiring. Originally introduced in (Kohlas, 2004) and (Kohlas & Wilson, 2006), this theory is extensively studied and substantiated with a large number of examples. Based on the work of (Aji & McEliece, 2000), we show that many important applications from signal processing and decoding theory reduce in fact to local computation with semiring valuation algebras. Another subclass identified for the very first time are *set-based semiring valuations* that map configuration sets to semiring values. This includes such important formalisms as those used in Dempster-Shafer theory, but it also yields a multiplicity of new valuation algebras that still wait for a future application.

Semirings with idempotent addition induce formalisms that model optimization problems. In this context, we are especially interested in the identification of configurations that adopt some optimum value. Such configurations are called *solution configurations* and their computation constitutes the problem of *dynamic programming*. (Shenoy, 1992b) presented a local computation architecture for the

identification of single solution configurations that applies to some special class of valuation algebras. We will determine this class more concretely by use of the semiring perspective. Further, we also present an extension of Shenoy's algorithm that enumerates all solution configurations. Another far more powerful algorithm for the same task associates local computation with knowledge compilation using Boolean functions (Darwiche, 2001). The innovative idea behind this algorithm consists in using local computation to compile solution configurations into a Boolean function. The result of this compilation process can then be queried, which gives efficient access to a lot more information than the pure enumeration of solution configurations. This work stems from a collaboration with Michael Wachter and Rolf Haenni.

This summary of the main results is completed by the NENOK software framework that has already been outlined above. Besides its serviceability as software library or local computation workbench, the development process was also guided by an interest in the realization of the algebraic framework. In fact, this undertaking affords the opportunity to study by means of a concrete and non-trivial example how far a mathematical framework can be mirrored in an object-oriented programming language. We provide in this thesis not only a tutorial for the NENOK's main functionalities, but we also focus on how they are realized and what measures must be taken to meet the mathematical requirements as well as possible.

1.2 Overview

This thesis is divided into four major parts:

Part I: The first part starts with the introduction of the mathematical framework of valuation and information algebras, together with their most important variations and properties. All these concepts are exemplified by a large catalogue of instances in Chapter 3. Chapter 4 is dedicated to the solution of projection problems which are first defined in a formal manner. Then, the four most important local computation architectures are introduced. Namely, these are the Shenoy-Shafer architecture, the Lauritzen-Spiegelhalter architecture, the Hugin architecture and the Idempotent architecture. Each architecture is also extended to compute scaled results. We then continue with some complexity considerations of local computation in general and give an outline of the most important improvements. Concluding, we investigate in Chapter 5 local computation on projection problems that contain distributed factors and show how efficient communication can be ensured.

Part II: In the second part, we discuss a particular subclass of valuation algebras that take values from a commutative semiring. For this purpose, we start with a short introduction to semiring theory, followed by a listing of some notable semiring examples. Then, a formal definition of semiring valuation algebras is given, and it is shown how semiring properties relate to valuation algebra properties. The semiring view allows then to identify new valuation algebra instances that extend the catalogue given in Part I.

Additionally, we will see a marvelous application of this theory that essentially shows how signal processing can be done using local computation with semiring valuations. Chapter 7 brings up another family of valuation algebras that is also based on the semiring structure. However, in contrast to usual semiring valuations, we map sets of configurations instead of single configurations to semiring values.

Part III: The third part covers the topic of dynamic programming and starts deriving optimization tasks from projection problems. The importance of these particular semantics is illustrated by numerous applications. Then, the central notion of solution configurations is introduced, whose identification with local computation takes up the remainder of Chapter 8. The first family of algorithms for this purpose enumerate solution configurations explicitly. Alternatively, a completely new approach is presented that compiles solution configurations into a Boolean function which qualifies for the efficient execution of a large number of additional queries. This closes the algebraic and algorithmic part of this thesis.

Part IV: The final part is to its full extent dedicated to the NENOK software framework. We start explaining the idea behind NENOK and its buildup as a layered model. Then, every layer is inspected in detail, starting with the algebraic layer in Chapter 10. This comprises the representation of valuation algebras as generic software framework and focusses in particular on the assistance for the development of user instances. Chapter 11 discusses the remote computing layer that offers the possibility for distributed computing in a processor network. Then, we give a tutorial on the use of the numerous local computation facilities in the NENOK framework. Finally, we discuss the processing of generic input and present a graphical user interface that facilitates the use of this project as experimental workbench for local computation.

Part I

Local Computation in Covering Join Trees

2
Valuation Algebras

As outlined in the introduction, the valuation algebra framework originally resulted from an abstraction process that identified the mathematical requirements for the application of some well-known inference mechanism. This system mirrored all essential properties we naturally associate with the rather imprecise notions of knowledge and information, such that it quickly served as an algebraic approach towards their definition. (Shenoy & Shafer, 1988; Shenoy & Shafer, 1990) published the axiomatic system of a valuation algebra for the very first time. Then, important contributions appeared in (Shafer, 1991) and (Kohlas, 2003), which both insisted on the existence of *neutral elements*, representing neutral knowledge with respect to certain questions. In this thesis, we propose a more general definition of valuation algebras that dispenses with neutral elements. This has some important advantages: On the one hand, it solves some open problems with formalisms where neutral elements do not have finite representations. On the other hand, we also have concrete examples where neutral knowledge does not exist. Both arguments are developed in Chapter 3. For the sake of completeness, valuation algebras with neutral elements are introduced as a specialization of this system, accompanied by many other refinements. However, neutral elements also play an important role in the context of local computation, since they ensure the existence of a so-called *join tree factorization*. This problem is solved in Chapter 4 by weakening the claim to a *covering join tree*, which requires some *placeholder element* in the valuation algebra itself that comes in whenever no other knowledge piece is present. This single element called *identity element* will be adjoined artificially to the valuation algebra, and it will be shown that all properties are conserved under this extension.

The first section of this chapter introduces the axiomatic system of valuation algebras. Then, some variations that consider neutral or contradictory elements are presented in Sections 2.2 and 2.4. Intermediately, Section 2.3 shows how a single identity element is adjoined, whenever no neutral elements exist. Then, an even more general valuation algebra definition is introduced where marginalization (knowledge extraction) is only partially defined. In contrast to (Schneuwly, 2007), we renounced building the whole theory on this more general framework, since it only appears within some technical constructions in this thesis. Section 2.6 is dedicated to the

introduction of a division operator for valuation algebras that later affords a more efficient caching policy during the local computation process. A further variation presented in Section 2.7 covers valuation algebras that support some concept of normalization. Finally, information algebras are stressed in Section 2.8 as another variation, and we close this chapter with a survey of the whole algebraic framework in Figure 2.3.

2.1 Operations & Axioms

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. 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 all valuations ϕ with $d(\phi) = s$. With this notation, the set of all possible valuations for a finite set of variables r can be defined as

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

Let $D = \mathcal{P}(r)$ be the power set of r and Φ a set of valuations with their domains in D. We assume the following operations defined in (Φ, D):

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

These are the three basic operations of a valuation algebra. If we interpret valuations as pieces of knowledge, the labeling operation tells us to which questions such a piece refers. Combination can be understood as aggregation of knowledge and marginalization as focusing or extraction of the part we are interested in. Sometimes this operation is also called *projection*. We now impose the following set of axioms on Φ and D:

(A1) *Commutative Semigroup:* Φ is associative and commutative under $\otimes$.

(A2) *Labeling:* For $\phi, \psi \in \Phi$,

$$d(\phi \otimes \psi) \quad = \quad d(\phi) \cup d(\psi). \tag{2.1}$$

(A3) *Marginalization:* For $\phi \in \Phi$, $x \in D$ and $x \subseteq d(\phi)$,

$$d(\phi^{\downarrow x}) \quad = \quad x. \tag{2.2}$$

(A4) *Transitivity:* For $\phi \in \Phi$ and $x \subseteq y \subseteq d(\phi)$,

$$(\phi^{\downarrow y})^{\downarrow x} \quad = \quad \phi^{\downarrow x}. \tag{2.3}$$

(A5) *Combination:* For ϕ, $\psi \in \Phi$ with $d(\phi) = x$, $d(\psi) = y$ and $z \in D$ such that $x \subseteq z \subseteq x \cup y$,

$$(\phi \otimes \psi)^{\downarrow z} \quad = \quad \phi \otimes \psi^{\downarrow z \cap y}. \tag{2.4}$$

(A6) *Domain:* For $\phi \in \Phi$ with $d(\phi) = x$,

$$\phi^{\downarrow x} \quad = \quad \phi. \tag{2.5}$$

These axioms require natural properties of a valuation algebra regarding knowledge or information modelling. The first axiom indicates that Φ is a commutative semigroup under combination. If information comes in pieces, the sequence of their aggregation does not influence the overall knowledge. The labeling axiom tells us that the combination of valuations yields knowledge about the union of the involved domains. Neither do variables vanish, nor do new ones appear. The marginalization axiom expresses the natural functioning of focusing. Transitivity says that marginalization can be performed in several steps. In order to explain the naturalness of the combination axiom, let us assume that we have some information about a domain in order to answer a certain question. Then, the combination axiom states how the answer is affected if a new information piece arrives. We can either combine the new piece to the given information and focus afterwards to the specified domain, or first remove the uninteresting parts of the new information and combine it afterwards. Both approaches lead to the same result. In fact, we are going to see in Section 6.3 that this axiom requires some generalized distributive law. Finally, the domain axiom ensures that information is not influenced by projecting it to its own domain, which expresses some kind of stability with respect to trivial marginalization.

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

In the first appearance of the valuation algebra axiom system (Shenoy & Shafer, 1990) only Axioms (A1), (A4) and a simpler version of (A5) were listed. Axiom (A2), on the other hand, was simply assumed in the definition of combination. (Shafer, 1991) mentioned Property (A3) for the first time and also remarked that Axiom (A6) cannot be derived from the others. Additionally, this axiomatic system contained so-called neutral valuations. We will introduce valuation algebras with neutral elements in Section 2.2 as a special case of the definition given here.

The following lemma describes a few elementary properties derived directly from the above set of axioms.

Lemma 2.2.

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

$$(\phi \otimes \psi)^{\downarrow x \cap y} \quad = \quad \phi^{\downarrow x \cap y} \otimes \psi^{\downarrow x \cap y}. \tag{2.6}$$

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

$$(\phi \otimes \psi)^{\downarrow z} = (\phi \otimes \psi^{\downarrow x \cap y})^{\downarrow z}. \tag{2.7}$$

Proof.

1. By the transitivity and combination axioms

$$\begin{aligned}(\phi \otimes \psi)^{\downarrow x \cap y} &= ((\phi \otimes \psi)^{\downarrow x})^{\downarrow x \cap y} \\ &= (\phi \otimes \psi^{\downarrow x \cap y})^{\downarrow x \cap y} \\ &= \phi^{\downarrow x \cap y} \otimes \psi^{\downarrow x \cap y}.\end{aligned}$$

2. By the transitivity and combination axioms

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

□

The definition of a valuation algebra given here is based on a set of variables r such that domains correspond to finite sets of variables. On this note, the set of domains D is a *lattice of subsets* of variables. In the literature, this restriction is often weakened in the sense that D may be any *modular lattice* (Shafer, 1991; Kohlas, 2003). However, for our purposes it is a lot more adequate to accept the restriction on variable systems. Moreover, it is sometimes convenient to replace the operation of marginalization by another primitive operation called *variable elimination*. For a valuation $\phi \in \Phi$ and a variable $X \in d(\phi)$, we define

$$\phi^{-X} = \phi^{\downarrow d(\phi) - \{X\}}. \tag{2.8}$$

Some important properties of variable elimination that follow straightly from this definition and the valuation algebra axioms are pooled in the following lemma:

Lemma 2.3.

1. *For* $\phi \in \Phi$ *and* $X \in d(\phi)$ *we have*

$$d(\phi^{-X}) = d(\phi) - \{X\}. \tag{2.9}$$

2. *For* $\phi \in \Phi$ *and* $X, Y \in d(\phi)$ *we have*

$$(\phi^{-X})^{-Y} = (\phi^{-Y})^{-X}. \tag{2.10}$$

3. *For* $\phi, \psi \in \Phi$, $x = d(\phi)$, $y = d(\psi)$, $Y \notin x$ *and* $Y \in y$ *we have*

$$(\phi \otimes \psi)^{-Y} = \phi \otimes \psi^{-Y}. \tag{2.11}$$

Proof.

1. By the labeling axiom

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

2. By Property 1 and the transitivity axiom

$$\begin{aligned}(\phi^{-X})^{-Y} &= (\phi^{\downarrow d(\phi)-\{X\}})^{\downarrow (d(\phi)-\{X\})-\{Y\}} \\ &= \phi^{\downarrow d(\phi)-\{X,Y\}} \\ &= (\phi^{\downarrow d(\phi)-\{Y\}})^{\downarrow (d(\phi)-\{Y\})-\{X\}} \\ &= (\phi^{-Y})^{-X}.\end{aligned}$$

3. Since $Y \notin x$ and $Y \in y$ we have $x \subseteq (x \cup y) - \{Y\} \subseteq x \cup y$. Hence, we obtain by application of the combination axiom

$$\begin{aligned}(\phi \otimes \psi)^{-Y} &= (\phi \otimes \psi)^{\downarrow (x \cup y)-\{Y\}} \\ &= \phi \otimes \psi^{\downarrow ((x \cup y)-\{Y\}) \cap y} \\ &= \phi \otimes \psi^{\downarrow y-\{Y\}} \\ &= \phi \otimes \psi^{-Y}.\end{aligned}$$

□

According to Lemma 2.3 Property 2, variables can be eliminated in any order. Thus, we may define the consecutive elimination of a non-empty set of variables $s = \{X_1, X_2, \ldots, X_n\} \subseteq d(\phi)$ unambiguously as

$$\phi^{-s} = (((\phi^{-X_1})^{-X_2}) \ldots)^{-X_n}. \tag{2.12}$$

This on the other hand permits to express any marginalization by the elimination of all unrequested variables. For $x \subseteq d(\phi)$, we have

$$\phi^{\downarrow x} = \phi^{-(d(\phi)-x)}. \tag{2.13}$$

To sum it up, Equations (2.8) and (2.13) allow to switch between marginalization and variable elimination ad libitum.

Beyond the definition of a valuation algebra given here, there are many variations with further interesting properties. These refinements will be discussed in the remaining part of this chapter. For those readers that prefer to see some concrete valuation algebra examples in place, we refer to Chapter 3 that contains a large catalogue of possible *instances*. The word instance alludes in this context to a mathematical formalism for knowledge representation that provides appropriate definitions of labeling, combination and marginalization, which in turn satisfy the valuation algebra axioms. Naturally, all instances presented in Chapter 3 will also be analysed for the properties that are introduced next.

2.2 Valuation Algebras with Neutral Elements

Following the view of a valuation algebra as a generic representation of knowledge or information, some instances may exist that contain pieces which express neutral information. In this case, such a *neutral element* must exist for every set of questions

$s \in D$, and if we combine those pieces with already existing knowledge of the same domain, we do not gain new information. Hence, there is an element $e_s \in \Phi_s$ for every sub-semigroup Φ_s such that $\phi \otimes e_s = e_s \otimes \phi = \phi$ for all $\phi \in \Phi_s$. We add further a neutrality axiom to the above system which states that a combination of neutral elements leads to a neutral element with respect to the union domain.

(A7) *Neutrality:* For $x, y \in D$,

$$e_x \otimes e_y \quad = \quad e_{x \cup y}. \tag{2.14}$$

If neutral elements exist, they are unique within the corresponding sub-semigroup. In fact, suppose the existence of another element $e'_s \in \Phi_s$ with the identical property that $\phi \otimes e'_s = e'_s \otimes \phi = \phi$ for all $\phi \in \Phi_s$. Then, since e_s and e'_s behave neutrally among each other, $e_s = e_s \otimes e'_s = e'_s$. Furthermore, valuation algebras with neutral elements allow for a simplified version of the combination axiom that was already proposed in (Shenoy & Shafer, 1990).

Lemma 2.4. *In a valuation algebra with neutral elements, the combination axiom is equivalently expressed as:*

(A5) *Combination: For ϕ, $\psi \in \Phi$ with $d(\phi) = x$, $d(\psi) = y$,*

$$(\phi \otimes \psi)^{\downarrow x} \quad = \quad \phi \otimes \psi^{\downarrow x \cap y}. \tag{2.15}$$

Proof. Equation (2.15) follows directly from the combination axiom with $z = x$. On the other hand, let $x \subseteq z \subseteq x \cup y$. Since $z \cap (x \cup y) = z$ we derive from Equation (2.15) and Axiom (A1) that

$$\begin{aligned} (\phi \otimes \psi)^{\downarrow z} &= e_z \otimes (\phi \otimes \psi)^{\downarrow z} \\ &= (e_z \otimes \phi \otimes \psi)^{\downarrow z} \\ &= e_z \otimes \phi \otimes \psi^{\downarrow y \cap z} \\ &= \phi \otimes \psi^{\downarrow y \cap z}. \end{aligned}$$

The last equality holds because

$$d(\phi \otimes \psi^{\downarrow y \cap z}) \;=\; x \cup (y \cap z) \;=\; (x \cup y) \cap (x \cup z) \;=\; z \;=\; d(e_z).$$

□

The presence of neutral elements allows furthermore to extend valuations to larger domains. This operation is called *vacuous extension* and may be accounted as a dual operation to marginalization. For $\phi \in \Phi$ and $d(\phi) \subseteq y$ we define

$$\phi^{\uparrow y} \quad = \quad \phi \otimes e_y. \tag{2.16}$$

Another property that may hold is that we cannot extract any information from a neutral element. This means that neutral elements project to neutral elements as indicated in the following axiom. Such valuation algebras are called *stable* (Shafer, 1991) and although this property seems to be very natural, there are indeed important examples which do not fulfill stability. Examples for both cases are discussed in Chapter 3.

(A8) *Stability:* For all $\phi \in \Phi$ and $x \subseteq y$,

$$e_y^{\downarrow x} = e_x. \tag{2.17}$$

In the case of a valuation algebra with neutral elements, the stability property can also be seen as a strengthening of the domain axiom. Indeed, it follows from the combination axiom that for $\phi \in \Phi$ with $d(\phi) = x$ we have

$$\phi^{\downarrow x} = (\phi \otimes e_x)^{\downarrow x} = \phi \otimes e_x^{\downarrow x} = \phi \otimes e_x = \phi.$$

A further interesting consequence is that under stability, vacuous extension can be revoked. From the combination axiom and $\phi \in \Phi$ with $d(\phi) = x \subseteq y$ it follows

$$(\phi^{\uparrow y})^{\downarrow x} = (\phi \otimes e_y)^{\downarrow x} = \phi \otimes e_y^{\downarrow x} = \phi \otimes e_x = \phi. \tag{2.18}$$

2.3 Adjoining an Identity Element

We have seen that the existence of neutral elements is not mandatory for a valuation algebra. This is in contrast to a large number of publications where neutral elements have always been an integral part of the axiomatic system. The gain is a more general definition where also instances without neutral elements are covered. Such an example will be given in Chapter 3. However, in order to provide the same computational possibilities, a so-called *identity element* is adjoined artificially whenever no neutral elements exist.

Let (Φ, D) be a valuation algebra according to Definition 2.1. We add a new valuation e to Φ and denote the resulting system by (Φ', D). Labeling, combination and marginalization are extended from Φ to Φ' in the following way:

1. *Labeling:* $\Phi' \to D;\ \phi \mapsto d'(\phi)$
   - $d'(\phi) = d(\phi)$, if $\phi \in \Phi$,
   - $d'(e) = \emptyset$;
2. *Combination:* $\Phi' \times \Phi' \to \Phi';\ (\phi, \psi) \mapsto \phi \otimes' \psi$
   - $\phi \otimes' \psi = \phi \otimes \psi$ if $\phi, \psi \in \Phi$,
   - $\phi \otimes' e = e \otimes' \phi = \phi$ if $\phi \in \Phi$,
   - $e \otimes' e = e$.
3. *Marginalization:* $\Phi' \times D \to \Phi';\ (\phi, x) \mapsto \phi^{\downarrow' x}$, for $x \subseteq d(\phi)$
   - $\phi^{\downarrow' x} = \phi^{\downarrow x}$ if $\phi \in \Phi$,
   - $e^{\downarrow' \emptyset} = e$.

If another element e' with the identical property that $e' \otimes \phi = \phi \otimes e' = \phi$ for all $\phi \in \Phi'$ already exists in Φ', there is no need to perform this extension. Indeed, we then have $e = e \otimes e' = e'$. This is in particular the case if the valuation algebra provides neutral elements. We will next see that the proposed extension of (Φ, D) conserves the properties of a valuation algebra.

Lemma 2.5. *(Φ', D) with extended operations d', $\otimes'$ and $\downarrow'$ is a valuation algebra.*

Proof. We verify the axioms on (Φ', D). Since they are fulfilled for the elements in Φ, we can restrict ourselves to those rules that include the identity element e.

(A1) *Commutative Semigroup:* Commutativity follows from the extension of the combination operator. Associativity for $\phi, \psi \in \Phi$ by

$$\begin{aligned} \phi \otimes' (\psi \otimes' e) &= \phi \otimes' \psi &&= (\phi \otimes' \psi) \otimes' e \\ e \otimes' (\psi \otimes' e) &= \psi &&= (e \otimes' \psi) \otimes' e \\ e \otimes' (e \otimes' e) &= e &&= (e \otimes' e) \otimes' e. \end{aligned}$$

(A2) *Labeling:* For $\phi \in \Phi$ we have

$$\begin{aligned} d'(\phi \otimes' e) &= d'(\phi) = d'(\phi) \cup d'(e) \\ d'(e \otimes' e) &= d'(e) = d'(e) \cup d'(e) = \emptyset. \end{aligned}$$

(A3) *Marginalization:* Since $x \in D$ and $x \subseteq d(e)$ we have $x = \emptyset$ and therefore by the extensions of labeling and marginalization

$$d'(e^{\downarrow' \emptyset}) = d'(e) = \emptyset.$$

(A4) *Transitivity:* $x \subseteq y \subseteq d(e)$ implies $x = y = \emptyset$. Hence, we derive from the extension of marginalization

$$(e^{\downarrow' \emptyset})^{\downarrow' \emptyset} = e^{\downarrow' \emptyset}.$$

(A5) *Combination:* For $\phi \in \Phi$ with $d(\phi) = x$, $d(e) = y = \emptyset$ and $z \in D$ such that $x \subseteq z \subseteq x \cup y$ it follows $z = x$ and by the domain axiom in (Φ, D)

$$(\phi \otimes' e)^{\downarrow' z} = \phi^{\downarrow' z} = \phi^{\downarrow x} = \phi = \phi \otimes' e^{\downarrow' \emptyset} = \phi \otimes' e^{\downarrow' z \cap \emptyset}.$$

Second, let $d(e) = x = \emptyset$, $d(\phi) = y$ and $z \in D$ such that $x \subseteq z \subseteq x \cup y$. It follows $z \cap y = z$ and we get

$$(e \otimes' \phi)^{\downarrow' z} = \phi^{\downarrow' z} = e \otimes' \phi^{\downarrow' z} = e \otimes' \phi^{\downarrow' z \cap y}.$$

Finally

$$(e \otimes' e)^{\downarrow' \emptyset} = e = e \otimes' e^{\downarrow' \emptyset \cap \emptyset}.$$

(A6) *Domain:* The extension of marginalization yields directly $e^{\downarrow' \emptyset} = e$.

□

It is generally known that an identity element can be adjoined to any commutative semigroup. Hence, we may add an identity element to every sub-semigroup Φ_x and if we furthermore ensure that Axiom (A7) holds, we obtain a valuation algebra with neutral elements (Shafer, 1991). Alternatively, we propose here to adjoin a neutral element to Φ exclusively. This approach has been developed for the first time in (Schneuwly *et al.*, 2004) where it was shown that all valuation algebra properties are conserved under this extension. Furthermore, the presence of an identity element allows the same computational possibilities as in the case of neutral elements. Both statements will also be justified subsequently.

It was mentioned above that an identity element is adjoined whenever no neutral elements exist. Indeed, the following lemma shows that it is needless to perform the extension if an element with identical properties already exists in (Φ, D). This is for example the case if the valuation algebra provides neutral elements.

Lemma 2.6. *If (Φ, D) is a valuation algebra with neutral elements, then $e_\emptyset \in \Phi$ has the same properties in (Φ, D) as e in (Φ', D).*

Proof. Since $e_\emptyset$ is the neutral element in $\Phi_\emptyset$, its domain is $d(e_\emptyset) = \emptyset$. Further we get by commutativity and the neutrality axiom

$$\phi \;=\; \phi \otimes e_s \;=\; \phi \otimes e_{s\cup\emptyset} \;=\; \phi \otimes e_s \otimes e_\emptyset \;=\; \phi \otimes e_\emptyset. \tag{2.19}$$

The property $e_\emptyset \otimes e_\emptyset = e_\emptyset$ follows by the definition of a neutral element, which behaves neutral to itself. So, the characteristics of e are fulfilled by $e_\emptyset$ regarding combination. Finally, marginalization follows directly from the domain axiom. □

If they are not used to distinguish between the two algebras, we usually identify the operations in (Φ', D) with their counterparts in (Φ, D). Doing so, we write d for d', $\otimes$ for $\otimes'$ and $\downarrow$ for $\downarrow'$.

2.4 Valuation Algebras with Null Elements

Some valuation algebras contain elements that express incompatible, inconsistent or contradictory knowledge according to questions $s \in D$. Such valuations behave absorbingly with respect to combination and are therefore called *absorbing elements* or simply *null elements*. Hence, in a valuation algebra with null elements there is an element $z_s \in \Phi_s$ such that $z_s \otimes \phi = \phi \otimes z_s = z_s$ for all $\phi \in \Phi_s$. Appropriately, we call a valuation $\phi \in \Phi_s$ *consistent*, if and only if $\phi \neq z_s$. It is furthermore a very natural claim that a projection of some consistent valuation produces again a consistent valuation. This requirement is captured by the following additional axiom.

(A9) *Nullity:* For $x, y \in D$, $x \subseteq y$ and $\phi \in \Phi_y$,

$$\phi^{\downarrow x} \;=\; z_x$$

if, and only if, $\phi = z_y$.

According to this definition, null valuations absorb only elements of the same domain. In case of a stable valuation algebra however, this property holds also for any other valuation.

Lemma 2.7. *In a stable valuation algebra with null elements we have for all $\phi \in \Phi$ with $d(\phi) = x$ and $y \in D$*

$$\phi \otimes z_y \quad = \quad z_{x \cup y}. \tag{2.20}$$

Proof. We remark first that from Equation (2.18) it follows

$$\left(z_y^{\uparrow x \cup y}\right)^{\downarrow y} \quad = \quad z_y.$$

Hence, we conclude from the nullity axiom that $z_y^{\uparrow x \cup y} = z_{x \cup y}$ and derive

$$\begin{aligned} \phi \otimes z_y &= (\phi \otimes e_{x \cup y}) \otimes (z_y \otimes e_{x \cup y}) \\ &= \phi^{\uparrow x \cup y} \otimes z_y^{\uparrow x \cup y} \\ &= \phi^{\uparrow x \cup y} \otimes z_{x \cup y} \\ &= z_{x \cup y}. \end{aligned}$$

□

Again, we refer to Chapter 3 for a study of valuation algebra instances with and without null elements.

2.5 Valuation Algebras with Partial Marginalization

The valuation algebra definition given at the beginning of this chapter allows every valuation $\phi \in \Phi$ to be marginalized to any subset of $d(\phi)$. Hence, we may say that ϕ can be marginalized to all domains in the *marginal set* $\mathcal{M}(\phi) = \mathcal{P}(d(\phi))$, where $\mathcal{P}(d(\phi))$ denotes the power set of $d(\phi)$. Valuation algebras with partial marginalization are more general in the sense that not all marginals are necessarily defined. In this view, $\mathcal{M}(\phi)$ may be a strict subset of $\mathcal{P}(d(\phi))$. It is therefore sensible that a fourth operation is needed which produces $\mathcal{M}(d(\phi))$ for all $\phi \in \Phi$. Additionally, all axioms that bear somehow on marginalization must be generalized to take the corresponding marginal sets into consideration.

Thus, let Φ be a set of valuations over domains $s \subseteq r$ and $D = \mathcal{P}(r)$. We assume the following operations defined on (Φ, D):

1. *Labeling:* $\Phi \to D$; $\phi \mapsto d(\phi)$,
2. *Combination:* $\Phi \times \Phi \to \Phi$; $(\phi, \psi) \mapsto \phi \otimes \psi$,
3. *Domain:* $\Phi \to \mathcal{P}(D)$; $\phi \mapsto \mathcal{M}(\phi)$,
4. *Partial Marginalization:* $\Phi \times D \to \Phi$; $(\phi, x) \mapsto \phi^{\downarrow x}$ defined for $x \in \mathcal{M}(\phi)$.

The set $\mathcal{M}(\phi)$ contains therefore all domains $x \in D$ such that the marginal of ϕ relative to x is defined. We impose now the following set of axioms on Φ and D, pointing out that the two Axioms (A1) and (A2) remain equal to the traditional definition of a valuation algebra.

(A1) *Commutative Semigroup:* Φ is associative and commutative under $\otimes$.

(A2) *Labeling:* For $\phi, \psi \in \Phi$,

$$d(\phi \otimes \psi) \quad = \quad d(\phi) \cup d(\psi). \tag{2.21}$$

(A3) *Marginalization:* For $\phi \in \Phi$ and $x \in \mathcal{M}(\phi)$,

$$d(\phi^{\downarrow x}) \quad = \quad x. \tag{2.22}$$

(A4) *Transitivity:* If $\phi \in \Phi$ and $x \subseteq y \subseteq d(\phi)$, then

$$x \in \mathcal{M}(\phi) \Rightarrow x \in \mathcal{M}(\phi^{\downarrow y}) \text{ and } y \in \mathcal{M}(\phi) \text{ and } (\phi^{\downarrow y})^{\downarrow x} = \phi^{\downarrow x}. \tag{2.23}$$

(A5) *Combination:* If $\phi, \psi \in \Phi$ with $d(\phi) = x$, $d(\psi) = y$ and $z \in D$ such that $x \subseteq z \subseteq x \cup y$, then

$$z \cap y \in \mathcal{M}(\psi) \Rightarrow z \in \mathcal{M}(\phi \otimes \psi) \text{ and } (\phi \otimes \psi)^{\downarrow z} = \phi \otimes \psi^{\downarrow z \cap y}. \tag{2.24}$$

(A6) *Domain:* For $\phi \in \Phi$ with $d(\phi) = x$, we have $x \in \mathcal{M}(\phi)$ and

$$\phi^{\downarrow x} \quad = \quad \phi. \tag{2.25}$$

Definition 2.8. *A system (Φ, D) together with the operations of labeling, combination, partial marginalization and domain satisfying these axioms is called a valuation algebra with partial marginalization.*

It is easy to see that this system is indeed a generalization of the traditional valuation algebra, because if $\mathcal{M}(\phi) = \mathcal{P}(d(\phi))$ holds for all $\phi \in \Phi$, the axioms reduce to the system given at the beginning of this chapter. Therefore, the latter is also called *valuation algebra with full marginalization.*

Adjoining an Identity Element

We have seen in Section 2.3 how to adjoin a unique identity element to a valuation algebra. This construction ought to be repeated for the more general case where marginalization is only partially defined. Doing so, a system (Φ', D) is obtained and the operations are extended in the following way.

1. *Labeling:* $\Phi' \to D$; $\phi \mapsto d'(\phi)$
   - $d'(\phi) = d(\phi)$, if $\phi \in \Phi$,
   - $d'(e) = \emptyset$.

2. *Combination:* $\Phi' \times \Phi' \to \Phi'$; $(\phi, \psi) \mapsto \phi \otimes' \psi$
   - $\phi \otimes' \psi = \phi \otimes \psi$ if $\phi, \psi \in \Phi$,
   - $\phi \otimes' e = e \otimes' \phi = \phi$ if $\phi \in \Phi$,
   - $e \otimes' e = e$.
3. *Domain:* $\Phi' \to \mathcal{P}(D)$; $\phi \mapsto \mathcal{M}'(\phi)$
   - $\mathcal{M}'(\phi) = \mathcal{M}(\phi)$, if $\phi \in \Phi$,
   - $\mathcal{M}'(\phi) = \emptyset$, if $\phi = e$.
4. *Marginalization:* $\Phi' \times D \to \Phi'$; $(\phi, x) \mapsto \phi^{\downarrow' x}$
   - $\phi^{\downarrow' x} = \phi^{\downarrow x}$ if $\phi \in \Phi$ and $x \in \mathcal{M}(\phi)$,
   - $e^{\downarrow' \emptyset} = e$.

Lemma 2.9. *(Φ', D) with the extended operations d', $\otimes'$, $\mathcal{M}'$ and $\downarrow'$ is a valuation algebra with partial marginalization.*

Proof. We verify the axioms on (Φ', D). Since they are fulfilled for the elements in Φ, we can restrict ourselves to those rules that include the identity element e. Furthermore, the proofs for the Axioms (A1), (A2), (A3) and (A6) are exactly as in the proof of Lemma 2.5.

(A4) *Transitivity:* $x \subseteq y \subseteq d(e)$ implies $x = y = \emptyset$ and by the extension of marginalization, we have
   - $\emptyset \in \mathcal{M}'(e)$, if, and only if, $\emptyset \in \mathcal{M}'(e^{\downarrow' \emptyset}) = \mathcal{M}'(e)$,
   - $(e^{\downarrow' \emptyset})^{\downarrow' \emptyset} = e^{\downarrow' \emptyset}$.

(A5) *Combination:* For $\phi \in \Phi$ with $d(\phi) = x$, $d(e) = y = \emptyset$ and $z \in D$ such that $x \subseteq z \subseteq x \cup y$ it follows $z = x$. Then, $z \cap y = x \cap \emptyset = \emptyset \in \mathcal{M}'(e)$ "implies" $x \in \mathcal{M}'(\phi \otimes' e) = \mathcal{M}'(\phi)$ by the domain axiom and by the same axiom in (Φ, D) we get

$$(\phi \otimes' e)^{\downarrow' z} = \phi^{\downarrow' z} = \phi^{\downarrow x} = \phi = \phi \otimes' e^{\downarrow' \emptyset} = \phi \otimes' e^{\downarrow' z \cap \emptyset}.$$

On the other hand, let $d(e) = x = \emptyset$, $d(\phi) = y$ and $z \in D$ such that $x \subseteq z \subseteq x \cup y = y$ and it follows $z \cap y = z$. That means, $z \cap y \in \mathcal{M}'(\phi)$ implies $z \in \mathcal{M}'(\phi \otimes' e) = \mathcal{M}'(\phi)$ and we get

$$(e \otimes' \phi)^{\downarrow' z} = \phi^{\downarrow' z} = e \otimes' \phi^{\downarrow' z} = e \otimes' \phi^{\downarrow' z \cap y}.$$

Finally, $\emptyset \in \mathcal{M}'(e)$ implies $\emptyset \in \mathcal{M}'(e \otimes' e) = \mathcal{M}'(e)$ and

$$(e \otimes' e)^{\downarrow' \emptyset} = e = e \otimes' e^{\downarrow' \emptyset \cap \emptyset}.$$

□

This concludes the introduction of valuation algebras with partial marginalization. They become especially important in the next section where certain valuation algebras are embedded into valuation algebras that provide a division operator. In this case, the property of full marginalization may disappear.

2.6 Valuation Algebras with Inverse Elements

A particular important class of valuation algebras are those that contain an *inverse element* for every valuation $\phi \in \Phi$. Hence, these valuation algebras posses the notion of *division* which will later be exploited for a more efficient treatment of valuations.

Definition 2.10. *Valuations $\phi, \psi \in \Phi$ are called inverses, if*

$$\phi \otimes \psi \otimes \phi \;=\; \phi, \quad \textit{and} \quad \psi \otimes \phi \otimes \psi \;=\; \psi. \tag{2.26}$$

We directly conclude from this definition that inverses necessarily have the same domain since the first condition implies that $d(\psi) \subseteq d(\phi)$ and from the second we obtain $d(\phi) \subseteq d(\psi)$. Therefore, $d(\phi) = d(\psi)$ must hold. It is furthermore suggested that inverse elements are not necessarily unique.

Subsequently, we will study under which conditions valuation algebras provide inverse elements. For this purpose, (Kohlas, 2003) distinguished three different cases. *Separativity* is the weakest requirement. It allows the valuation algebra to be embedded into a union of groups which naturally include inverse elements. However, the price we pay is that full marginalization gets lost. Alternatively, *regularity* is a stronger condition. There, the valuation algebra itself is decomposed into groups with inverses such that full marginalization is conserved. Finally, if the third and strongest requirement called *idempotency* holds, every valuation turns out to be the inverse of itself.

Equivalence relations play an important role in all three cases since they will be used for the decomposition of Φ. To make a long story short, an equivalence relation γ is *reflexive*, *symmetric* and *transitive*. Thus, we have for $\phi, \psi, \eta \in \Phi$

1. *Reflexivity:* $\phi \equiv \phi \pmod{\gamma}$.
2. *Symmetry:* $\phi \equiv \psi \pmod{\gamma}$ implies $\psi \equiv \phi \pmod{\gamma}$.
3. *Transitivity:* $\phi \equiv \psi \pmod{\gamma}$ and $\psi \equiv \eta \pmod{\gamma}$ imply $\phi \equiv \eta \pmod{\gamma}$.

Since we want to partition a valuation algebra, equivalence relations that are compatible with the operations in Φ are of particular importance. Such relations are called *congruences* and satisfy the following properties:

1. $\phi \equiv \psi \pmod{\gamma}$ implies $d(\phi) = d(\psi)$.
2. $\phi \equiv \psi \pmod{\gamma}$ implies $\phi^{\downarrow x} \equiv \psi^{\downarrow x} \pmod{\gamma}$ if $x \subseteq d(\phi) = d(\psi)$.
3. $\phi_1 \equiv \psi_1 \pmod{\gamma}$ and $\phi_2 \equiv \psi_2 \pmod{\gamma}$ imply $\phi_1 \otimes \phi_2 \equiv \psi_1 \otimes \psi_2 \pmod{\gamma}$.

An intuitive motivation for the introduction of congruences in valuation algebras is that different valuations sometimes represent the same knowledge. Thus, they may be pooled in equivalence classes that are induced by such a congruence relation γ.

2.6.1 Separative Valuation Algebras

The mathematical notion of a *group* involves a set with a binary operation that is associative, has a unique identity and where each element provides a well-defined inverse. Thus, we may obtain inverse elements in a valuation algebra if the latter allows to be embedded into a union of groups. It is known from semigroup theory (Clifford & Preston, 1967) that the property of *cancellativity* is sufficient to embed a semigroup into a union of groups, and it is therefore reasonable to apply this technique also for valuation algebras (Lauritzen & Jensen, 1997). However, (Kohlas, 2003) remarked that a more restrictive requirement is needed in the case of valuation algebras that also accounts for the operation of marginalization. These prerequisites are given in the following definition of *separativity*. There, we assume a congruence γ that divides Φ into disjoint equivalence classes $[\phi]_\gamma$ given by

$$[\phi]_\gamma \quad = \quad \{\psi \in \Phi : \psi \equiv \phi \pmod{\gamma}\}. \tag{2.27}$$

Definition 2.11. *A valuation algebra (Φ, D) is called* separative*, if*

- *for all $\psi, \psi' \in [\phi]_\gamma$ with $\phi \otimes \psi = \phi \otimes \psi'$, we have $\psi = \psi'$,*
- *there is a congruence γ in (Φ, D), such that for all $\phi \in \Phi$ and $t \subseteq d(\phi)$,*

$$\phi^{\downarrow t} \otimes \phi \quad \equiv \quad \phi \pmod{\gamma}. \tag{2.28}$$

From the properties of a congruence, it follows that all elements of an equivalence class have the same domain. Further, the equivalence classes are closed under combination. For $\phi, \psi \in [\phi]_\gamma$ and thus $\phi \equiv \psi \pmod{\gamma}$, we conclude from Equation (2.28) that

$$\phi \otimes \psi \;\equiv\; \phi \otimes \phi \;\equiv\; \phi \pmod{\gamma}.$$

Additionally, since all $[\phi]_\gamma$ are subsets of Φ, combination within an equivalence class is both associative and commutative. Hence, Φ decomposes into a family of disjoint, *commutative semigroups*

$$\Phi \quad = \quad \bigcup_{\phi \in \Phi} [\phi]_\gamma.$$

Semigroups obeying the first property in the definition of separativity are called *cancellative* (Clifford & Preston, 1967) and it follows from semigroup theory that every cancellative and commutative semigroup $[\phi]_\gamma$ can be embedded into a group. These groups are denoted by $\gamma(\phi)$ and contain pairs (ϕ, ψ) of elements from $[\phi]_\gamma$ (Croisot, 1953; Tamura & Kimura, 1954). Two group elements (ϕ, ψ) and (ϕ', ψ') are identified if $\phi \otimes \psi' = \phi' \otimes \psi$. Further, multiplication within $\gamma(\phi)$ is defined by

$$(\phi, \psi) \otimes (\phi', \psi') \quad = \quad (\phi \otimes \phi', \psi \otimes \psi'). \tag{2.29}$$

Let Φ^* be the union of those groups $\gamma(\phi)$, i.e.

$$\Phi^* \quad = \quad \bigcup_{\phi \in \Phi} \gamma(\phi).$$

We define the combination $\otimes^*$ of elements $(\phi, \psi), (\phi', \psi') \in \Phi^*$ by

$$(\phi, \psi) \otimes^* (\phi', \psi') \quad = \quad (\phi \otimes \phi', \psi \otimes \psi'). \tag{2.30}$$

It can easily be shown that this combination is well defined, associative and commutative. Φ^* is therefore a semigroup under $\otimes^*$. Moreover, the mapping from Φ to Φ^* defined by

$$\phi \quad \mapsto \quad (\phi \otimes \phi, \phi).$$

is a semigroup homomorphism which is furthermore one-to-one due to the cancellativity of the semigroup $[\phi]_\gamma$. In this way, Φ is embedded as a semigroup into Φ^*. Subsequently, we identify $\phi \in \Phi$ with its counterpart $(\phi \otimes \phi, \phi)$ in Φ^*. Since $\gamma(\phi)$ are groups, they contain both inverses and an identity element. Thus, we write

$$\phi^{-1} \quad = \quad (\phi, \phi \otimes \phi)$$

for the inverse of group element ϕ, and denote the identity element within $\gamma(\phi)$ as $f_{\gamma(\phi)}$. Note that neither inverses nor identity elements necessarily belong to Φ but only to Φ^*. The embedding of Φ into a union of groups Φ^* via its decomposition into commutative semigroups is illustrated in Figure 2.1.

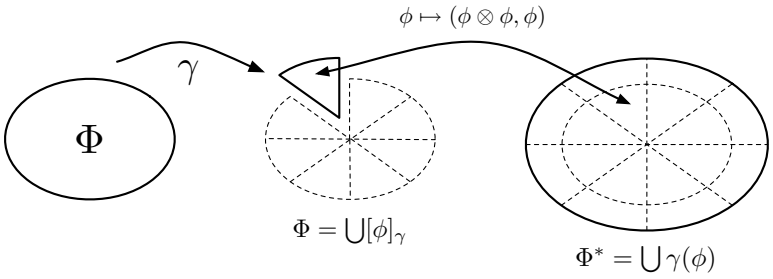


Figure 2.1: A separative valuation algebra Φ is decomposed into disjoint equivalence classes by the congruence γ, and finally embedded into a union of groups Φ^*.

We next introduce a relation between the semigroups $[\phi]_\gamma$. Since γ is a congruence, we may define the combination between congruence classes as

$$[\psi]_\gamma \otimes [\phi]_\gamma = [\phi \otimes \psi]_\gamma, \tag{2.31}$$

and say that

$$[\psi]_\gamma \leq [\phi]_\gamma \quad \text{if} \quad [\psi \otimes \phi]_\gamma = [\phi]_\gamma. \tag{2.32}$$

This relation is a *partial order*, i.e. reflexive, transitive and antisymmetric, as shown in (Kohlas, 2003), and it can be carried over to $\gamma(\phi)$ by defining

$$\gamma(\psi) \leq \gamma(\phi) \quad \text{if} \quad [\psi]_\gamma \leq [\phi]_\gamma.$$

The following properties are proved in (Kohlas, 2003):

Lemma 2.12.

1. *If* $\gamma(\psi) \leq \gamma(\phi)$*, then* $\phi' \otimes f_{\gamma(\psi)} = \phi'$ *for all* $\phi' \in \gamma(\phi)$*.*
2. $\gamma(\phi^{\downarrow t}) \leq \gamma(\phi)$ *for all* $t \subseteq d(\phi)$*.*
3. *For all* $\phi, \psi \in \Phi$ *it holds that* $\gamma(\phi) \leq \gamma(\phi \otimes \psi)$*.*
4. $(\phi \otimes \psi)^{-1} = \phi^{-1} \otimes \psi^{-1}$*.*

It is now possible to extend (partially) the operations of labeling, marginalization and combination from (Φ, D) to (Φ^*, D).

- Labeling d^* for elements in Φ^* is defined for $\eta \in \Phi^*$ by $d^*(\eta) = d(\psi)$ for some $\psi \in \Phi$, if $\eta \in \gamma(\psi)$. Since γ is a congruence, d^* does not depend on the representative ψ of the group $\gamma(\psi)$. So d^* is well defined. Further, the label for an element $\phi \in \Phi$ is $d^*(\phi) = d(\phi)$. In other words, d^* is an extension of d. Since η and η^{-1} are contained in the same group, we directly get $d^*(\eta) = d^*(\eta^{-1})$.
- We have already seen in Equation (2.30) how the combination is carried out among elements in Φ^*. This operator also extends $\otimes$, since

  $$\phi \otimes^* \psi \;=\; (\phi \otimes \phi, \phi) \otimes^* (\psi \otimes \psi, \psi) \;=\; (\phi \otimes \psi \otimes \phi \otimes \psi, \phi \otimes \psi) \;=\; \phi \otimes \psi$$

  for $\phi, \psi \in \Phi$, which are identified by $(\phi \otimes \phi, \phi) \in \Phi^*$ and $(\psi \otimes \psi, \psi) \in \Phi^*$ respectively. With this extension of combination, we further remark that ϕ and ϕ^{-1} are indeed inverses according to Definition 2.10, since

  $$\phi \otimes^* \phi^{-1} \otimes^* \phi \;=\; \phi \otimes^* f_{\gamma(\phi)} \;=\; \phi \tag{2.33}$$

  and

  $$\phi^{-1} \otimes^* \phi \otimes^* \phi^{-1} \;=\; \phi^{-1} \otimes^* f_{\gamma(\phi)} \;=\; \phi^{-1}. \tag{2.34}$$

- Finally, we extend partially the operator of marginalization. Given an element $\eta \in \Phi^*$, the marginal $\downarrow^*$ of η to a domain s is defined by

  $$\eta^{\downarrow^* s} \;=\; \phi^{\downarrow s} \otimes^* \psi^{-1}, \tag{2.35}$$

  if there are $\phi, \psi \in \Phi$ with $d(\psi) \subseteq s \subseteq d(\phi)$ and $\gamma(\psi) \leq \gamma(\phi)$ such that

  $$\eta \;=\; \phi \otimes^* \psi^{-1}.$$

  This definition will be justified using the following lemma.

Lemma 2.13. *Let* $\eta = (\phi, \psi)$ *be the pair representation of an element* $\eta \in \Phi^*$ *with* $\phi, \psi \in \Phi$*. Then* η *can be written as*

$$\eta \;=\; \phi \otimes^* \psi^{-1}.$$

Proof. Because γ is a congruence, we have $d^*(\eta) = d(\phi) = d(\psi)$. Further, we have

$$\begin{aligned}(\phi, \psi) &= (\phi \otimes \phi \otimes \psi, \psi \otimes \phi \otimes \psi) \\ &= (\phi \otimes \phi, \phi) \otimes^* (\psi, \psi \otimes \psi) \\ &= \phi \otimes^* \psi^{-1}.\end{aligned}$$

□

So, any element $\eta \in \Phi^*$ can at least be projected to its own domain. It will next be shown that $\downarrow^*$ is well defined. Take two representations of the same element $\eta_1, \eta_2 \in \Phi^*$ with $\eta_1 = (\phi, \psi)$, $\eta_2 = (\phi', \psi')$ and $\phi \otimes \psi' = \psi \otimes \phi'$ such that $\eta_1 = \eta_2$. Assume that both can be projected to a domain s, i.e. there are $\phi_1, \phi_2, \psi_1, \psi_2 \in \Phi$, $\gamma(\psi_1) \leq \gamma(\phi_1)$, $\gamma(\psi_2) \leq \gamma(\phi_2)$ and $d(\psi_1) \subseteq s \subseteq d(\phi_1)$, $d(\psi_2) \subseteq s \subseteq d(\phi_2)$ such that $\eta_1 = \phi_1 \otimes^* \psi_1^{-1}$ and $\eta_2 = \phi_2 \otimes^* \psi_2^{-1}$. We show that

$$\eta_1^{\downarrow * s} = \eta_2^{\downarrow * s}. \tag{2.36}$$

First, we conclude from $\eta_1 = \eta_2$ that

$$\phi_1 \otimes^* \psi_1^{-1} = \phi_2 \otimes^* \psi_2^{-1}.$$

We multiply both sides with $\psi_1 \otimes^* \psi_2$ and obtain

$$\phi_1 \otimes^* \psi_1^{-1} \otimes^* \psi_1 \otimes^* \psi_2 = \phi_2 \otimes^* \psi_2^{-1} \otimes^* \psi_2 \otimes^* \psi_1,$$

thus

$$\phi_1 \otimes^* f_{\gamma(\psi_1)} \otimes^* \psi_2 = \phi_2 \otimes^* f_{\gamma(\psi_2)} \otimes^* \psi_1.$$

Since $\gamma(\psi_1) \leq \gamma(\phi_1)$ and $\gamma(\psi_2) \leq \gamma(\phi_2)$, Lemma 2.12 implies

$$\phi_1 \otimes^* \psi_2 = \phi_2 \otimes^* \psi_1,$$

and because $\otimes^*$ extends the combination in Φ, $\phi_1 \otimes \psi_2 = \phi_2 \otimes \psi_1$ must hold. By application of the combination axiom in (Φ, D)

$$\phi_1^{\downarrow s} \otimes \psi_2 = (\phi_1 \otimes \psi_2)^{\downarrow s} = (\phi_2 \otimes \psi_1)^{\downarrow s} = \phi_2^{\downarrow s} \otimes \psi_1. \tag{2.37}$$

Next, it will be shown that

$$\gamma(\psi_1) \leq \gamma(\phi_2^{\downarrow s}) \quad \text{and} \quad \gamma(\psi_2) \leq \gamma(\phi_1^{\downarrow s}). \tag{2.38}$$

Since $\gamma(\psi_1) = \gamma(\psi_1^{-1}) \leq \gamma(\phi_1)$, Equation (2.32) implies that $\gamma(\phi_1) = \gamma(\phi_1 \otimes \psi_1) = \gamma(\phi_1 \otimes \psi_1^{-1})$ and therefore

$$\gamma(\phi_1) = \gamma(\phi_1 \otimes^* \psi_1^{-1}) = \gamma(\phi_2 \otimes^* \psi_2^{-1}) = \gamma(\phi_2). \tag{2.39}$$

On the other hand, the combination axiom in (Φ, D) yields

$$\gamma(\phi_1^{\downarrow s}) = \gamma((\phi_1 \otimes \psi_1)^{\downarrow s}) = \gamma(\phi_1^{\downarrow s} \otimes \psi_1).$$

Therefore $\gamma(\psi_1) \leq \gamma(\phi_1^{\downarrow s})$ and using Equation (2.39) we obtain $\gamma(\psi_1) \leq \gamma(\phi_2^{\downarrow s})$. This proves the first relation of (2.38) and the second is obtained by a similar reasoning. Finally, we deduce from multiplying both sides of (2.37) with $\psi_1^{-1} \otimes^* \psi_2^{-1}$

$$\phi_1^{\downarrow s} \otimes^* \psi_2 \otimes^* \psi_1^{-1} \otimes^* \psi_2^{-1} \;=\; \phi_2^{\downarrow s} \otimes^* \psi_1 \otimes^* \psi_1^{-1} \otimes^* \psi_2^{-1},$$

hence

$$\phi_1^{\downarrow s} \otimes^* \psi_1^{-1} \otimes^* f_{\gamma(\psi_2)} \;=\; \phi_2^{\downarrow s} \otimes^* \psi_2^{-1} \otimes^* f_{\gamma(\psi_1)}$$

and due to (2.38) and Lemma 2.12 we finally obtain

$$\eta_1^{\downarrow * s} \;=\; \phi_1^{\downarrow s} \otimes^* \psi_1^{-1} \;=\; \phi_2^{\downarrow s} \otimes^* \psi_2^{-1} \;=\; \eta_2^{\downarrow * s}.$$

This proves that marginalization in (Φ^*, D) is well defined.

It remains to be verified that $\downarrow^*$ is really an extension of $\downarrow$. Let $\phi \in \Phi$. In the first step we have to guarantee that $\phi^{\downarrow * s}$ is defined for all $s \subseteq d(\phi)$. The valuation ϕ can be written as

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

According to Lemma 2.12, $\gamma(\phi^{\downarrow \emptyset}) \leq \gamma(\phi \otimes \phi^{\downarrow \emptyset})$ and $d(\phi^{\downarrow \emptyset}) = \emptyset \subseteq d(\phi \otimes \phi^{\downarrow \emptyset})$. Therefore, the marginal $\downarrow^*$ of ϕ to any domain $s \subseteq d(\phi \otimes \phi^{\downarrow \emptyset}) = d(\phi)$ is defined. Now, using the combination axiom in (Φ, D),

$$\begin{aligned} \phi^{\downarrow * s} &= (\phi \otimes \phi^{\downarrow \emptyset})^{\downarrow s} \otimes^* (\phi^{\downarrow \emptyset})^{-1} \\ &= (\phi^{\downarrow s} \otimes \phi^{\downarrow \emptyset}) \otimes^* (\phi^{\downarrow \emptyset})^{-1} \\ &= \phi^{\downarrow s}. \end{aligned}$$

Note that the extended domain operator $\mathcal{M}^*$ is directly induced by the definition of the new marginalization operator, i.e. for $\eta \in \Phi^*$

$$\begin{aligned} \mathcal{M}^*(\eta) \;=\; \{s : \exists \phi, \psi \in \Phi \text{ such that } d(\psi) \subseteq s \subseteq d(\phi) \text{ and} \\ \eta = \phi \otimes^* \psi^{-1} \text{ and } \gamma(\psi) \leq \gamma(\phi)\}. \end{aligned} \tag{2.40}$$

Theorem 2.14. *(Φ^*, D) with the operations of labeling d^*, combination $\otimes^*$, domain $\mathcal{M}^*$ and marginalization $\downarrow^*$ is a valuation algebra with partial marginalization.*

Proof. We verify the axioms on (Φ^*, D).

(A1) *Commutative Semigroup:* We have already seen that Φ^* is a commutative semigroup under $\otimes^*$.

(A2) *Labeling:* Consider two elements $\eta_1, \eta_2 \in \Phi^*$ with $\eta_1 \in \gamma(\phi)$ and $\eta_2 \in \gamma(\psi)$ for $\phi, \psi \in \Phi$. It follows from the definition of the labeling operator that $d^*(\eta_1) = d(\phi)$ and $d^*(\eta_2) = d(\psi)$. We know from Equation (2.30) that $\eta_1 \otimes^* \eta_2 \in \gamma(\phi \otimes \psi)$. Indeed, η_1 and η_2 can be written as pairs of elements of $[\phi]_\gamma$ and $[\psi]_\gamma$ respectively and therefore $\eta_1 \otimes^* \eta_2$ as pairs of elements of $[\phi]_\gamma \otimes [\psi]_\gamma = [\phi \otimes \psi]_\gamma$. Hence

$$d^*(\eta_1 \otimes^* \eta_2) \;=\; d(\phi \otimes \psi) \;=\; d(\phi) \cup d(\psi) \;=\; d^*(\eta_1) \cup d^*(\eta_2).$$

(A3) *Marginalization:* If $\eta \in \Phi^*$ can be marginalized to s, we have

$$\eta^{\downarrow *s} \;=\; \phi^{\downarrow s} \otimes^* \psi^{-1}$$

with $d(\psi) \subseteq s$. It follows from the labeling axiom

$$d^*(\eta^{\downarrow *s}) \;=\; d^*(\phi^{\downarrow s} \otimes^* \psi^{-1}) \;=\; d(\phi^{\downarrow s}) \cup d^*(\psi^{-1}) \;=\; d(\phi^{\downarrow s}) \;=\; s.$$

(A4) *Transitivity:* Let $\eta \in \Phi^*$, $t \subseteq s \subseteq d(\eta)$ such that $t \in \mathcal{M}^*(\eta)$. We have

$$\eta \;=\; \phi \otimes^* \psi^{-1}$$

with $\phi, \psi \in \Phi$, $d(\psi) \subseteq t$ and $\gamma(\psi) \leq \gamma(\phi)$. Since $d(\psi) \subseteq t \subseteq s \subseteq d(\phi)$ it follows $s \in \mathcal{M}^*(\eta)$. The projection of η to the domain s yields

$$\eta^{\downarrow *s} \;=\; \phi^{\downarrow s} \otimes^* \psi^{-1}.$$

From Lemma 2.12 and $\gamma(\psi) \leq \gamma(\phi)$ we derive $\gamma(\phi) = \gamma(\phi \otimes \psi)$ and obtain by application of the combination axiom in (Φ, D)

$$\gamma(\phi^{\downarrow s}) \;=\; \gamma((\phi \otimes \psi)^{\downarrow s}) \;=\; \gamma(\phi^{\downarrow s} \otimes \psi).$$

Therefore $\gamma(\psi) \leq \gamma(\phi^{\downarrow s})$ must hold. Since $d(\psi) \subseteq t$ we have shown that $t \in \mathcal{M}^*(\eta^{\downarrow *s})$. By the transitivity axiom in (Φ, D) we finally get

$$(\eta^{\downarrow *s})^{\downarrow t} \;=\; (\phi^{\downarrow s})^{\downarrow t} \otimes^* \psi^{-1} \;=\; \phi^{\downarrow t} \otimes^* \psi^{-1} \;=\; \eta^{\downarrow *t}.$$

(A5) *Combination:* Let $\eta_1, \eta_2 \in \Phi^*$ with $s = d(\eta_1)$, $t = d(\eta_2)$ and $s \subseteq z \subseteq s \cup t$. It is first shown that $z \cap t \in \mathcal{M}^*(\eta_2)$ implies $z \in \mathcal{M}^*(\eta_1 \otimes^* \eta_2)$. Assume that $z \cap t \in \mathcal{M}^*(\eta_2)$, i.e.

$$\eta_2 \;=\; \phi_2 \otimes^* \psi_2^{-1}$$

with $\phi_2, \psi_2 \in \Phi$, $d(\psi_2) \subseteq z \cap t$ and $\gamma(\psi_2) \leq \gamma(\phi_2)$. By Lemma 2.13 there are $\phi_1, \psi_1 \in \Phi$ with $\gamma(\phi_1) = \gamma(\psi_1)$ such that

$$\eta_1 \;=\; \phi_1 \otimes^* \psi_1^{-1}.$$

We then get

$$\eta_1 \otimes^* \eta_2 \;=\; (\phi_1 \otimes \phi_2) \otimes^* (\psi_1 \otimes \psi_2)^{-1}$$

with $d(\psi_1 \otimes \psi_2) = d(\psi_1) \cup d(\psi_2) \subseteq z$. Further

$$\gamma(\phi_1 \otimes \phi_2) \;=\; \gamma(\phi_1 \otimes \psi_1 \otimes \phi_2 \otimes \psi_2)$$

and it follows $\gamma(\psi_1 \otimes \psi_2) \leq \gamma(\phi_1 \otimes \phi_2)$. So $z \in \mathcal{M}^*(\eta_1 \otimes^* \eta_2)$. We finally get by the combination axiom in (Φ, D)

$$\begin{aligned} \eta_1 \otimes^* \eta_2^{\downarrow z \cap t} &= (\phi_1 \otimes \phi_2^{\downarrow z \cap t}) \otimes^* \psi_1^{-1} \otimes^* \psi_2^{-1} \\ &= (\phi_1 \otimes \phi_2)^{\downarrow z} \otimes^* (\psi_1 \otimes \psi_2)^{-1} \\ &= (\eta_1 \otimes \eta_2)^{\downarrow *z}. \end{aligned}$$

(A6) *Domain:* Let $\eta \in \Phi^*$ with $t = d^*(\eta)$. By the Lemma 2.13, $t \in \mathcal{M}^*(\eta)$, that is $\eta = \phi \otimes^* \psi^{-1}$ with $d(\psi) \subseteq t$ and $\gamma(\psi) \leq \gamma(\phi)$. By the domain axiom in (Φ, D) we get

$$\eta^{\downarrow * t} \;=\; \phi^{\downarrow t} \otimes^* \psi^{-1} \;=\; \phi \otimes^* \psi^{-1} \;=\; \eta.$$

□

There are cases when we do not need to worry about the existence of the marginals in the combination axiom.

Lemma 2.15. *Let $\eta \in \Phi^*$ and $\psi \in \Phi$ with $s = d^*(\eta)$ and $t = d(\psi)$. For every domain $z \in D$ such that $s \subseteq z \subseteq s \cup t$ we have $z \in \mathcal{M}(\eta \otimes \psi)$.*

Proof. Since ψ can be projected to any domain contained in $d(\psi)$, especially to $z \cap t$, it follows from the combination axiom that $z \in \mathcal{M}(\eta \otimes \psi)$. □

Adjoining an Identity Element

We continue to show that adjoining the identity element e as described in the Section 2.3 does not affect separativity. We extend the congruence relation γ in (Φ, D) to a congruence relation γ' in (Φ', D). We say that $\phi \equiv \psi \pmod{\gamma'}$ if either

- $\phi, \psi \in \Phi$ *and* $\phi \equiv \psi \pmod{\gamma}$, or
- $\phi = \psi = e$.

Lemma 2.16. *γ' is a congruence in the valuation algebra (Φ', D).*

Proof. For $\phi, \psi \in \Phi$ this property is induced by γ. By the definition of γ', it is not possible that $\phi \equiv e \pmod{\gamma'}$ or $e \equiv \phi \pmod{\gamma'}$ if $\phi \neq e$. In order to prove that γ' is an equivalence relation, it is therefore sufficient to check reflexivity of e. But $e \equiv e \pmod{\gamma'}$ follows from the definition of γ'. Let us prove that γ' is a congruence, i.e. that it is compatible with the valuation algebra operations.

1. $e \equiv e \pmod{\gamma'}$ implies trivially $d(e) = d(e)$.
2. $e \equiv e \pmod{\gamma'}$ implies $e^{\downarrow \emptyset} \equiv e^{\downarrow \emptyset} \pmod{\gamma'}$, since $e = e^{\downarrow \emptyset}$.
3. $e \equiv e \pmod{\gamma'}$ and $\phi \equiv \psi \pmod{\gamma'}$ imply $\phi \otimes e \equiv \psi \otimes e \pmod{\gamma'}$, since $\phi \otimes e = \phi$ and $\psi \otimes e = \psi$.

□

The equivalence class $[e]_{\gamma'}$ consists of the single element e.

Lemma 2.17. *If (Φ, D) is a separative valuation algebra according to the congruence γ, then so is (Φ', D) according to γ' with the extended operators d', $\downarrow'$ and $\otimes'$.*

Proof. We have seen in Lemma 2.5 that (Φ', D) is a valuation algebra with the operators d', $\downarrow'$ and $\otimes'$. Further, Lemma 2.16 shows that γ' is a congruence in (Φ', D). It remains to shown that γ' obeys the properties of separativity given in Definition 2.11. For $\phi \in \Phi$ and $t \subseteq d(\phi)$ the congruence γ induces $\phi^{\downarrow t} \otimes \phi \equiv \phi \pmod{\gamma'}$. For e we get the desired result from $e^{\downarrow \emptyset} \otimes e = e$ and reflexivity of γ',

$$e^{\downarrow \emptyset} \otimes e \quad \equiv \quad e \pmod{\gamma'}.$$

Cancellativity is again induced for the elements in Φ by γ. Since $[e]_{\gamma'}$ consists of the single element e, cancellativity of $[e]_{\gamma'}$ is trivial. □

It remains to show that in this case, the valuation algebra (Φ^*, D) which is induced by (Φ', D) provides an identity element too.

Lemma 2.18. *Let (Φ', D) be a separative valuation algebra with an unique identity element e. Then, there is also an identity element e^* in the valuation algebra (Φ^*, D) induced by (Φ', D).*

Proof. Let $e^* = (e \otimes e, e) = (e, e)$. We verify the properties imposed on e^*. We have for $\eta = (\phi, \psi) \in \Phi^*$ with $\phi, \psi \in \Phi$

$$\begin{aligned} d^*(e^*) &= d(e) = \emptyset, \\ \eta \otimes^* e^* &= (\phi, \psi) \otimes^* (e, e) = (\phi \otimes e, \psi \otimes e) = (\phi, \psi) = \eta, \\ e^* \otimes^* e^* &= (e, e) \otimes^* (e, e) = (e \otimes e, e \otimes e) = (e, e) = e^*. \end{aligned}$$

and by the domain axiom $(e^*)^{\downarrow * \emptyset} = e^*$. □

Again, we usually identify the operators in (Φ^*, D) like in (Φ, D), i.e. d^* by d, $\otimes^*$ by $\otimes$ and $\downarrow^*$ by $\downarrow$ if they are not used to distinguish between the two cases.

2.6.2 Regular Valuation Algebras

We have just seen that a separative valuation algebra decomposes into disjoint semigroups which in turn can be embedded into groups. Thus, we obtain an extended valuation algebra where every element has an inverse. However, (Kohlas, 2003) remarked that in some cases, valuation algebras can directly be decomposed into groups instead of only semigroups. This makes life much easier since we can avoid the rather complicated embedding and moreover, full marginalization is conserved. A sufficient condition for this simplification is the property of *regularity* stated in the following definition. Note also that it again extends the usual notion of regularity from semigroup theory to incorporate the operation of marginalization.

Definition 2.19.

- *An element $\phi \in \Phi$ is called* regular*, if there exists for all $t \subseteq d(\phi)$ an element $\chi \in \Phi$ with $d(\chi) = t$, such that*

$$\phi \quad = \quad \phi^{\downarrow t} \otimes \chi \otimes \phi. \tag{2.41}$$

- *A valuation algebra* (Φ, D) *is called* regular*, if all its elements are regular.*

In contrast to the more general case of separativity, we are now looking for a congruence that decomposes Φ directly into a union of groups. For this purpose, we introduce the Green relation (Green, 1951) between valuations:

$$\phi \equiv \psi \pmod{\gamma} \quad \text{if} \quad \phi \otimes \Phi = \psi \otimes \Phi. \tag{2.42}$$

$\phi \otimes \Phi$ denotes the set of valuations $\{\phi \otimes \eta : \eta \in \Phi\}$, i.e. the *principal ideal* generated by ϕ. (Kohlas, 2003) proves that the Green relation is actually a congruence in a regular valuation algebra and that the corresponding equivalence classes $[\phi]_\gamma$ are directly groups which therefore provide inverse elements.

Lemma 2.20. *If* ϕ *is regular with* $\phi = \phi \otimes \chi \otimes \phi$*, then* ϕ *and* $\chi \otimes \phi \otimes \chi$ *are inverses.*

Proof. From Definition 2.10 we obtain

$$\begin{aligned} \phi \otimes (\chi \otimes \phi \otimes \chi) \otimes \phi &= \phi \otimes \chi \otimes (\phi \otimes \chi \otimes \phi) \\ &= \phi \otimes \chi \otimes \phi \\ &= \phi \end{aligned}$$

and

$$\begin{aligned} (\chi \otimes \phi \otimes \chi) \otimes \phi \otimes (\chi \otimes \phi \otimes \chi) &= \chi \otimes (\phi \otimes \chi \otimes \phi) \otimes \chi \otimes \phi \otimes \chi \\ &= \chi \otimes (\phi \otimes \chi \otimes \phi) \otimes \chi \\ &= \chi \otimes \phi \otimes \chi. \end{aligned}$$

□

The equivalence classes $[\phi]_\gamma$ are clearly cancellative since they are groups and therefore contain inverse elements. Further, the Green relation satisfies Equation (2.28) which makes a regular valuation algebra also separative. From this point of view, regular valuation algebras are special cases of separative valuation algebras and since Φ does not need to be extended, full marginalization is preserved. This construction is illustrated in Figure 2.2.

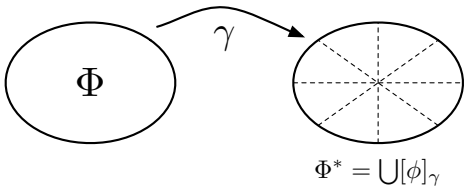


Figure 2.2: A regular valuation algebra Φ decomposes directly into a union of groups Φ^* by the Green relation.

Idempotent elements play an important role in regular valuation algebras. These are elements $f \in \Phi$ such that $f \otimes f = f$. According to Definition 2.10 we may therefore say that idempotent elements are inverse to themselves. Essentially, idempotents are obtained by combining inverses. Thus, if $\phi, \psi \in \Phi$ are inverses, $f = \phi \otimes \psi$ is idempotent since

$$f \otimes f \;=\; (\phi \otimes \psi) \otimes (\phi \otimes \psi) \;=\; (\phi \otimes \psi \otimes \phi) \otimes \psi \;=\; \phi \otimes \psi \;=\; f.$$

These idempotents behave neutral with respect to ϕ and ψ. We have

$$f \otimes \phi \;=\; (\phi \otimes \psi) \otimes \phi \;=\; \phi,$$

and the same holds naturally for ψ. The following lemma states that every principal ideal $\phi \otimes \Phi$ is generated by a unique idempotent valuation.

Lemma 2.21. *In a regular valuation algebra there exists for all $\phi \in \Phi$ a* unique *idempotent $f \in \Phi$ with $\phi \otimes \Phi = f \otimes \Phi$.*

Proof. Since ϕ is regular there is a χ such that $\phi = \phi \otimes \chi \otimes \phi$ and we know from Lemma 2.20 that ϕ and $\chi \otimes \phi \otimes \chi$ are inverses. Thus, $f = \phi \otimes (\chi \otimes \phi \otimes \chi) = \phi \otimes \chi$ is an idempotent such that $f \otimes \phi = \phi$. Therefore, $\phi \otimes \psi = f \otimes (\phi \otimes \psi) \in f \otimes \Phi$ and $f \otimes \psi = \phi \otimes (\chi \otimes \psi) \in \phi \otimes \Phi$. Consequently, $\phi \otimes \Phi = f \otimes \Phi$ must hold. It remains to prove that f is unique. Suppose that $f_1 \otimes \Phi = f_2 \otimes \Phi$. Then, there exists $\chi \in \Phi$ such that $f_2 = f_1 \otimes \chi$. This implies that $f_1 \otimes f_2 = f_1 \otimes (f_1 \otimes \chi) = f_1 \otimes \chi = f_2$. Similarly, we derive $f_1 \otimes f_2 = f_2$ and therefore it follows that $f_1 = f_2$. □

These result about idempotents will take center stage in Section 2.6.3.

Adjoining an Identity Element

We will again show that an identity element e can be adjoined to any regular valuation algebra without affecting the property of regularity.

Lemma 2.22. *If (Φ, D) is a regular valuation algebra, then so is (Φ', D) with the extended operators d', $\downarrow'$ and $\otimes'$.*

Proof. Lemma 2.5 shows that (Φ', D) is a valuation algebra with the operators d', $\downarrow'$ and $\otimes'$. All elements in Φ are regular. This holds also for e, since $e = e^{\downarrow\emptyset} \otimes \chi \otimes e$ with $\chi = e$. □

2.6.3 Idempotent Valuation Algebras

The property of regularity allows the decomposition of a valuation algebras into groups such that inverses exist within Φ directly. We also learned that every such group is generated from a unique idempotent element. The last simplifying condition for the introduction of inverse elements identifies therefore valuation algebras where every element is idempotent. This has been proposed in (Kohlas, 2003) and leads to a decomposition where every valuation forms its own group.

Definition 2.23. *A valuation algebra (Φ, D) is called idempotent if for all $\phi \in \Phi$ and $t \subseteq d(\phi)$, it holds that*

$$\phi \otimes \phi^{\downarrow t} = \phi. \tag{2.43}$$

By choosing $\chi = \phi$ in Definition 2.19, we directly remark that every idempotent valuation algebra is regular too. Then, since principal ideals are spanned by a unique idempotent, each element of an idempotent valuation algebra generates its own principal ideal. Consequently, all groups $[\phi]_\gamma$ consist of the single element ϕ which is therefore also the inverse of itself.

Adjoining an Identity Element

It is not surprising that also the property of idempotency remains conserved if an identity element is adjoined to an idempotent valuation algebra.

Lemma 2.24. *If (Φ, D) is an idempotent valuation algebra, then so is (Φ', D) with the extended operators d', $\downarrow'$ and $\otimes'$.*

Proof. (Φ', D) is a valuation algebra with operations d', $\downarrow'$ and $\otimes'$ according to Lemma 2.5. All elements in Φ are idempotent and this holds in particular for e, since $e \otimes e^{\downarrow \emptyset} = e \otimes e = e$. □

This concludes the study of valuation algebras with inverse elements. The following section will show that *scaled* or *normalized* valuations can be introduced in a separative valuation algebra.

2.7 Scaled Valuation Algebras

Scaling or normalization is an important notion in a couple of formalisms which turn out to be valuation algebra instances. As already mentioned, the algebraic background for the introduction of a scaling operator is a separative valuation algebra, possibly with null elements. It is shown in this section how scaling can be introduced potentially in every valuation algebra that fulfills this mathematical property. Nevertheless, we accent that there must also be a semantical reason that argues for a scaling operator, and this cannot be treated on a purely algebraic level. We readopt this discussion in later parts where concrete examples are studied.

Following (Kohlas, 2003), we define for $\phi \in \Phi$

$$\phi^{\downarrow} = \phi \otimes \left(\phi^{\downarrow \emptyset}\right)^{-1}. \tag{2.44}$$

$\phi^{\downarrow}$ is called the *normalization* or *scale* of ϕ. It is important to note that because separativity is presumed, scaled valuations do not necessarily belong to Φ but only to Φ^*. But even if $\phi^{\downarrow}$ does not belong to Φ, its marginal is defined for any domain $t \subseteq d(\phi)$ due to Equation (2.35).

Multiplying both sides of Equation (2.44) with $\phi^{\downarrow\emptyset}$ leads to

$$\phi \;=\; \phi^{\downarrow} \otimes \phi^{\downarrow\emptyset} \tag{2.45}$$

because $\gamma(\phi) \geq \gamma(\phi^{\downarrow\emptyset})$. Furthermore, it follows from Equation (2.44) that $\gamma(\phi) \leq \gamma(\phi^{\downarrow})$ and from (2.45) that $\gamma(\phi^{\downarrow}) \leq \gamma(\phi)$. Hence $\gamma(\phi^{\downarrow}) = \gamma(\phi)$ which means that a valuation and its scale are contained in the same group. The following lemma lists some further elementary properties of scaling. It is important to note that these properties only apply if the scale $\phi^{\downarrow}$ of a valuation ϕ is again contained in the original algebra Φ. This is required since scaling is only defined for elements in Φ.

Lemma 2.25.

1. *For $\phi \in \Phi$ with $\phi^{\downarrow} \in \Phi$ we have*

$$\left(\phi^{\downarrow}\right)^{\downarrow} \;=\; \phi^{\downarrow}. \tag{2.46}$$

2. *For $\phi, \psi \in \Phi$ with $\phi^{\downarrow}, \psi^{\downarrow} \in \Phi$ we have*

$$(\phi \otimes \psi)^{\downarrow} \;=\; (\phi^{\downarrow} \otimes \psi^{\downarrow})^{\downarrow}. \tag{2.47}$$

3. *For $\phi \in \Phi$ with $\phi^{\downarrow} \in \Phi$ and $t \subseteq d(\phi)$ we have*

$$\left(\phi^{\downarrow}\right)^{\downarrow t} \;=\; \left(\phi^{\downarrow t}\right)^{\downarrow}. \tag{2.48}$$

Proof.

1. By application of the combination axiom

$$\left(\phi^{\downarrow}\right)^{\downarrow\emptyset} \;=\; \left(\phi \otimes \left(\phi^{\downarrow\emptyset}\right)^{-1}\right)^{\downarrow\emptyset} \;=\; \phi^{\downarrow\emptyset} \otimes \left(\phi^{\downarrow\emptyset}\right)^{-1} \;=\; f_{\gamma(\phi^{\downarrow\emptyset})}.$$

Hence

$$\left(\phi^{\downarrow}\right)^{\downarrow} \;=\; \phi^{\downarrow} \otimes \left(\left(\phi^{\downarrow}\right)^{\downarrow\emptyset}\right)^{-1} \;=\; \phi^{\downarrow} \otimes \left(f_{\gamma(\phi^{\downarrow\emptyset})}\right)^{-1} \;=\; \phi^{\downarrow}.$$

2. We first remark using the combination axiom and Equation (2.45) that

$$\begin{aligned}
\phi \otimes \psi &= (\phi^{\downarrow} \otimes \phi^{\downarrow\emptyset}) \otimes (\psi^{\downarrow} \otimes \psi^{\downarrow\emptyset}) \\
&= (\phi^{\downarrow} \otimes \psi^{\downarrow}) \otimes (\phi^{\downarrow\emptyset} \otimes \psi^{\downarrow\emptyset}) \\
&= (\phi^{\downarrow} \otimes \psi^{\downarrow})^{\downarrow} \otimes (\phi^{\downarrow} \otimes \psi^{\downarrow})^{\downarrow\emptyset} \otimes (\phi^{\downarrow\emptyset} \otimes \psi^{\downarrow\emptyset}) \\
&= (\phi^{\downarrow} \otimes \psi^{\downarrow})^{\downarrow} \otimes ((\phi^{\downarrow\emptyset} \otimes \psi^{\downarrow\emptyset}) \otimes (\phi^{\downarrow} \otimes \psi^{\downarrow}))^{\downarrow\emptyset} \\
&= (\phi^{\downarrow} \otimes \psi^{\downarrow})^{\downarrow} \otimes ((\phi^{\downarrow} \otimes \phi^{\downarrow\emptyset}) \otimes (\psi^{\downarrow} \otimes \psi^{\downarrow\emptyset}))^{\downarrow\emptyset} \\
&= (\phi^{\downarrow} \otimes \psi^{\downarrow})^{\downarrow} \otimes (\phi \otimes \psi)^{\downarrow\emptyset}.
\end{aligned}$$

Therefore, we conclude that

$$\begin{aligned}
(\phi \otimes \psi)^{\downarrow} &= (\phi \otimes \psi) \otimes \left((\phi \otimes \psi)^{\downarrow\emptyset}\right)^{-1} \\
&= (\phi^{\downarrow} \otimes \psi^{\downarrow})^{\downarrow} \otimes (\phi \otimes \psi)^{\downarrow\emptyset} \otimes \left((\phi \otimes \psi)^{\downarrow\emptyset}\right)^{-1} \\
&= (\phi^{\downarrow} \otimes \psi^{\downarrow})^{\downarrow}.
\end{aligned}$$

3. From Equation (2.45) we conclude on one hand

$$\phi^{\downarrow t} \;=\; \left(\phi^{\downarrow t}\right)^{\downarrow} \otimes \phi^{\downarrow \emptyset},$$

and on the other hand

$$\phi^{\downarrow t} \;=\; \left(\phi^{\downarrow} \otimes \phi^{\downarrow \emptyset}\right)^{\downarrow t} \;=\; \left(\phi^{\downarrow}\right)^{\downarrow t} \otimes \phi^{\downarrow \emptyset}.$$

Hence

$$\left(\phi^{\downarrow t}\right)^{\downarrow} \otimes \phi^{\downarrow \emptyset} \;=\; \left(\phi^{\downarrow}\right)^{\downarrow t} \otimes \phi^{\downarrow \emptyset}.$$

Since $\gamma(\phi^{\downarrow\emptyset}) \leq \gamma((\phi^{\downarrow})^{\downarrow t}), \gamma((\phi^{\downarrow t})^{\downarrow})$ it follows that

$$\left(\phi^{\downarrow t}\right)^{\downarrow} \;=\; \left(\phi^{\downarrow t}\right)^{\downarrow} \otimes \phi^{\downarrow \emptyset} \otimes \left(\phi^{\downarrow \emptyset}\right)^{-1} \;=\; \left(\phi^{\downarrow}\right)^{\downarrow t} \otimes \phi^{\downarrow \emptyset} \otimes \left(\phi^{\downarrow \emptyset}\right)^{-1} \;=\; \left(\phi^{\downarrow}\right)^{\downarrow t}.$$

□

In a separative valuation algebra with null elements, every null element z_s forms itself a group which is a consequence of cancellativity. Therefore, null elements are inverse to themselves and consequently

$$z_s^{\downarrow} \;=\; z_s \otimes \left(z_s^{\downarrow \emptyset}\right)^{-1} \;=\; z_s \otimes z_\emptyset^{-1} \;=\; z_s \otimes z_\emptyset \;=\; z_s. \tag{2.49}$$

If on the other hand neutral elements exist in (Φ, D), the scale of a neutral element is generally not a neutral element anymore. This property however is guaranteed if the valuation algebra is stable. It follows from Equations (2.19) and (2.45)

$$e_s \;=\; e_s^{\downarrow} \otimes e_s^{\downarrow \emptyset} \;=\; e_s^{\downarrow} \otimes e_\emptyset \;=\; e_s^{\downarrow}. \tag{2.50}$$

Let $\Phi^{\downarrow}$ be the set of all scaled valuations, i.e.

$$\Phi^{\downarrow} \;=\; \{\phi^{\downarrow} : \phi \in \Phi\}. \tag{2.51}$$

The operation of labeling is well defined in $\Phi^{\downarrow}$. If we assume furthermore that all scales are element of Φ, i.e. $\Phi^{\downarrow} \subseteq \Phi$, we know from Lemma 2.25 that $\Phi^{\downarrow}$ is also closed under marginalization. However, since a combination of scaled valuations does generally not yield a scaled valuation again, we define

$$\phi^{\downarrow} \oplus \psi^{\downarrow} \;=\; (\phi^{\downarrow} \otimes \psi^{\downarrow})^{\downarrow}. \tag{2.52}$$

Note that Lemma 2.25 also implies

$$\phi^{\downarrow} \oplus \psi^{\downarrow} \;=\; (\phi \otimes \psi)^{\downarrow}. \tag{2.53}$$

The requirement that all scales are elements of Φ, holds in particular if the valuation algebra is regular. If even idempotency holds, every valuation is the scale of itself,

$$\phi^{\downarrow} \;=\; \phi \otimes \left(\phi^{\downarrow \emptyset}\right)^{-1} \;=\; \phi \otimes \left(\phi^{\downarrow \emptyset}\right) \;=\; \phi. \tag{2.54}$$

Consequently, we have $\Phi^{\downarrow} = \Phi$.

Theorem 2.26. *Assume $\Phi^{\downarrow} \subseteq \Phi$. Then, $(\Phi^{\downarrow}, D)$ with the operations of labeling d, combination $\oplus$, and marginalization $\downarrow$ is a valuation algebra.*

Proof. We verify the axioms on $(\Phi^{\downarrow}, D)$.

(A1) *Commutative Semigroup:* Commutativity of $\oplus$ follows directly from Equation (2.52). Associativity is derived as follows:

$$\begin{aligned}(\phi^{\downarrow} \oplus \psi^{\downarrow}) \oplus \chi^{\downarrow} &= (\phi \otimes \psi)^{\downarrow} \oplus \chi^{\downarrow} = ((\phi \otimes \psi) \otimes \chi)^{\downarrow} = (\phi \otimes (\psi \otimes \chi))^{\downarrow} \\ &= \phi^{\downarrow} \oplus (\psi \otimes \chi)^{\downarrow} = \phi^{\downarrow} \oplus (\psi^{\downarrow} \oplus \chi^{\downarrow}).\end{aligned}$$

(A2) *Labeling:* Since a valuation and its scale both have the same domain,

$$d(\phi^{\downarrow} \oplus \psi^{\downarrow}) = d((\phi^{\downarrow} \otimes \psi^{\downarrow})^{\downarrow}) = d(\phi^{\downarrow} \otimes \psi^{\downarrow}) = d(\phi^{\downarrow}) \cup d(\psi^{\downarrow}).$$

(A3) *Marginalization:* For $\phi^{\downarrow} \in \Phi^{\downarrow}$ with $d(\phi^{\downarrow}) = x$ we have

$$d\left((\phi^{\downarrow})^{\downarrow x}\right) = d\left((\phi^{\downarrow x})^{\downarrow}\right) = d(\phi^{\downarrow x}) = x.$$

(A4) *Transitivity:* For $y \subseteq x \subseteq d(\phi^{\downarrow})$ it follows that

$$\left((\phi^{\downarrow})^{\downarrow x}\right)^{\downarrow y} = \left((\phi^{\downarrow x})^{\downarrow}\right)^{\downarrow y} = \left((\phi^{\downarrow x})^{\downarrow y}\right)^{\downarrow} = (\phi^{\downarrow y})^{\downarrow} = (\phi^{\downarrow})^{\downarrow y}.$$

(A5) *Combination:* For $\phi^{\downarrow}$, $\psi^{\downarrow} \in \Phi^{\downarrow}$ with $d(\phi^{\downarrow}) = x$, $d(\psi^{\downarrow}) = y$ and $z \in D$ such that $x \subseteq z \subseteq x \cup y$, we have

$$\begin{aligned}(\phi^{\downarrow} \oplus \psi^{\downarrow})^{\downarrow z} &= ((\phi \otimes \psi)^{\downarrow})^{\downarrow z} = ((\phi \otimes \psi)^{\downarrow z})^{\downarrow} \\ &= (\phi \otimes \psi^{\downarrow y \cap z})^{\downarrow} = \phi^{\downarrow} \oplus (\psi^{\downarrow})^{\downarrow y \cap z}.\end{aligned}$$

(A6) *Domain:* For $\phi^{\downarrow} \in \Phi^{\downarrow}$ and $d(\phi^{\downarrow}) = x$,

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

□

Accepting the requirement that $\Phi^{\downarrow} \subseteq \Phi$, the mapping $\phi \mapsto \phi^{\downarrow}$ is a homomorphism from (Φ, D) to $(\Phi^{\downarrow}, D)$. Irrespectively, the latter is called *scaled valuation algebra* associated with (Φ, D).

Adjoining an Identity Element

Finally, in a valuation algebra (Φ', D) with adjoined identity element e, the latter is not affected from scaling. From $e = e^{-1}$ follows that

$$e^{\downarrow} = e \otimes \left(e^{\downarrow \emptyset}\right)^{-1} = e \otimes e^{-1} = e \otimes e = e. \tag{2.55}$$

2.8 Information Algebras

At the very beginning of this chapter, we stressed the idea of a valuation algebra as a generic representation of knowledge or information. This association may carelessly be retained for the study of the subsequent chapters. Nevertheless, one often links the notion of information with further properties that are in general not present in a valuation algebra. More concretely, any reasonable formalism for representing information should contain neutral information pieces, and it should clearly be impossible to extract new information from such pieces. This corresponds to the property of stability which has been studied in Section 2.2. Another property that should hold in the context of information is idempotency. If we combine a piece of information with a part of itself, we neither gain nor loose information. Hence, we adopt the following definition of an *information algebra*.

Definition 2.27. *Valuation algebras with neutral elements that are idempotent and stable are called information algebras.*

Information algebras have a lot of interesting properties which are extensively studied in (Kohlas, 2003). Among other things, they allow for example the introduction of a partial order that expresses whether a piece of information is more *informative* than another one. However, because of the strong requirements, the above definition excludes many interesting formalisms, and furthermore it turns out that these properties are not essential for computing tasks. Therefore, information algebras will play only a minor role in the context of this thesis.

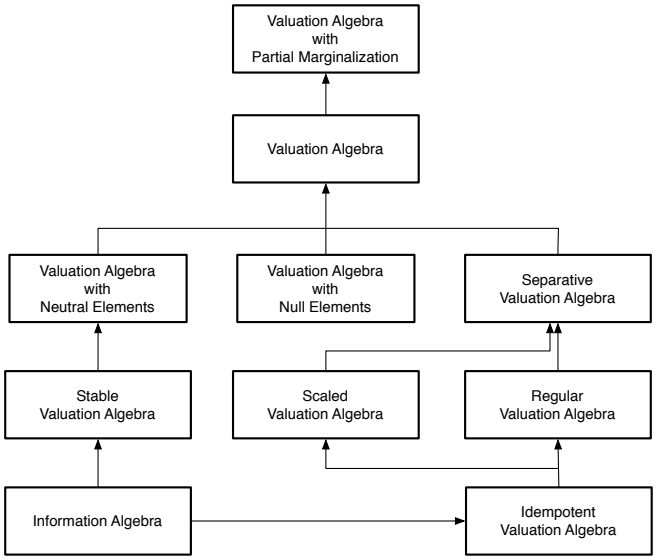


Figure 2.3: The valuation algebra framework.

2.9 Conclusion

This chapter introduced the algebraic framework of a valuation algebra on which all consecutive chapters are based. In contrast to related literature, a very general definition was given which dispenses with the inclusion of neutral elements. Instead, this and many further concepts are introduced as variations of the standard definition. In the belief that a picture says more than a thousand words, Figure 2.3 summarizes graphically how these different valuation algebras are related and which properties they inherit from variations that are located on a higher framework level. Thus, we for example see that an idempotent valuation algebra is also regular, separative and scaled. The following chapter comes up with a first collection of formalisms that adopt the structure of a valuation algebra. Every formalism will be investigated for the algebraic properties we just introduced, and this allows then to connect it with the valuation algebra framework of Figure 2.3. As we will see, most valuation algebras provide multiple connections with this framework. To cite an example, Section 3.4 introduces the formalism of probability potentials which is regular, scaled and which has neutral elements.

3
Valuation Algebra Instances

The previous chapter introduced valuation algebras on a very abstract level at which the exemplification of the various concepts and definitions was set aside. This will now be made up for. Throughout the following sections, a large collection of different formalisms for knowledge representation will be presented, which all turn out to be valuation algebra instances. We give a technical definition of every instance together with the corresponding rules for combination and marginalization and perform a formal verification of the valuation algebra axioms. Then, every example will be investigated for the properties introduced in the foregoing chapter in order to find out how exactly it is connected with the framework of Figure 2.3. It is clear that this selection contains only a tiny subset of all instances that are known today, and even with the examples added in Chapter 6 and Chapter 7, this listing is in fact extensive but in no way complete. Several of these instances, as well as parts of the theory we are going to develop in later chapters, are based on the concepts of configurations and configuration sets. Thus, we shall first treat these notions separately. Then, we study in this order the valuation algebra of indicator functions, relations, probability potentials, set potentials, distance potentials, density functions and propositional logic. All these formalisms are often mentioned in the context of valuation algebras, with the exception of distance potentials which are for the first time investigated as a valuation algebra.

3.1 Configurations & Configuration Sets

Consider a finite set r of variables and for every variable $X \in r$ a set Ω_X of values which can be assigned to X. This set is called the *frame* of variable X. Often, such variable frames are assumed to be finite, but this is not a general requirement. If a frame contains exactly two elements, the corresponding variable is said to be *binary*. Furthermore, if the two elements represent the states *true* and *false*, the variable is called *propositional* or *Boolean*.

- **Configurations:** A *configuration* f with *domain* $s \subseteq r$ is a function that associates a value $f(X) \in \Omega_X$ with each variable $X \in s$. By convention, the single configuration with empty domain is identified by the symbol $\diamond$. If

one does not need to refer to the functional aspect, configurations are often denoted by bold face, lower case letters.

- **Projection of configurations:** Given a configuration f with domain s and $t \subseteq s$. The *projection* of f to t is defined to be a configuration g with domain t such that $g(X) = f(X)$ for all $X \in t$. In this way, we may also write $\mathbf{x}^{\downarrow t}$ for the projection of a configuration $\mathbf{x}$ to domain t. Note that $\diamond$ denotes the projection of any configuration to the empty set. Furthermore, this notation allows to write $\mathbf{x} = (\mathbf{x}^{\downarrow t},\ \mathbf{x}^{\downarrow s-t})$ for the decomposition of $\mathbf{x}$ with respect to $t \subseteq s$. Observe also that $(\mathbf{x}, \diamond) = (\diamond, \mathbf{x}) = \mathbf{x}$.

- **Configuration sets:** Similar to frames of single variables, the frame Ω_s of a set of variables $s \subseteq r$ is defined to be the set of all possible configurations with domain s. In particular, $\Omega_\emptyset = \{\diamond\}$. Then, a *configuration set* with domain s is simply a subset $S \subseteq \Omega_s$. If a configuration set consists of only one element, then it is also called a *singleton*.

- **Projection of configuration sets:** Projection of configurations can simply be carried over to configuration sets $S \subseteq \Omega_s$. We define the *projection* of S to $t \subseteq s$ as $S^{\downarrow t} = \{\mathbf{x}^{\downarrow t} : \mathbf{x} \in S\}$.

- **Join of configuration sets:** The *join* of two configuration sets $S_1 \subseteq \Omega_s$ and $S_2 \subseteq \Omega_t$ is defined by $S_1 \bowtie S_2 = \{\mathbf{x} \in \Omega_{s \cup t} : \mathbf{x}^{\downarrow s} \in S_1,\ \mathbf{x}^{\downarrow t} \in S_2\}$. Observe that $S_1 \bowtie S_2$ has domain $s \cup t$.

- **Extension of configuration sets:** Finally, the *extension* of a configuration set $S \subseteq \Omega_t$ to some superdomain $s \supseteq t$ is defined by $S^{\uparrow s} = S \bowtie \Omega_{s-t}$ which corresponds to the largest set of configurations that projects to S.

These definitions allow for a uniform notation in the following walking-tour through a selected collection of valuation algebra instances. Concluding, we indicate that projection and join of configuration sets are well known operations in database theory. Moreover, configuration sets with these two operations even form a valuation algebra themselves – but this will be the topic of Section 3.3.

3.2 Indicator Functions

The formalism of *indicator functions* is the first valuation algebra instance that will be studied. Intuitively, an indicator function with domain $s \subseteq r$ identifies a subset $I \subseteq \Omega_s$ by specifying for each configuration $\mathbf{x} \in \Omega_s$ whether $\mathbf{x}$ belongs to I or not. If we adopt the usual interpretation of 0 for $\mathbf{x}$ not being an element of I and 1 for $\mathbf{x}$ being in I, an indicator i with domain $d(i) = s$ is a function that maps every configuration $\mathbf{x} \in \Omega_s$ onto a value $i(\mathbf{x}) \in \{0, 1\}$, i.e.

$$i : \Omega_s \to \{0, 1\}. \tag{3.1}$$

These are the valuations in the valuation algebra of indicator functions. Traditionally, we write Φ_s for the set of all indicator functions with domain s and Φ for the

set of all possible indicator functions over subsets of r.

Combination of indicator functions is defined by multiplication. If i_1 and i_2 are indicator functions with domain s and t respectively, we define for $\mathbf{x} \in \Omega_{s \cup t}$

$$i_1 \otimes i_2(\mathbf{x}) \quad = \quad i_1(\mathbf{x}^{\downarrow s}) \cdot i_2(\mathbf{x}^{\downarrow t}). \tag{3.2}$$

The operation of marginalization corresponds to maximization. For an indicator i with domain s and $t \subseteq s$ we define for all $\mathbf{x} \in \Omega_t$

$$i^{\downarrow t}(\mathbf{x}) \quad = \quad \max_{\mathbf{y} \in \Omega_{s-t}} i(\mathbf{x}, \mathbf{y}). \tag{3.3}$$

If we consider only variables with finite frames, an indicator function with domain s can simply be represented as an $|s|$-dimensional table with $|\Omega_s|$ zero-one entries. This together with an application of combination and marginalization is illustrated in the following Example.

Example 3.1. *Consider a set of variables $r = \{A, B, C\}$ with frames $\Omega_A = \{a, \overline{a}\}$, $\Omega_B = \{b, \overline{b}\}$ and $\Omega_C = \{c, \overline{c}\}$. Then, assume two indicator functions i_1 and i_2 with domains $d(i_1) = \{A, B\}$ and $d(i_2) = \{B, C\}$:*

$i_1 =$

A	**B**	
a	b	0
a	$\overline{b}$	1
$\overline{a}$	b	0
$\overline{a}$	$\overline{b}$	0

$i_2 =$

B	**C**	
b	c	1
b	$\overline{c}$	0
$\overline{b}$	c	1
$\overline{b}$	$\overline{c}$	1

We combine i_1 and i_2 and marginalize the result to $\{A\}$

$i_3 = i_1 \otimes i_2 =$

A	**B**	**C**	
a	b	c	0
a	b	$\overline{c}$	0
a	$\overline{b}$	c	1
a	$\overline{b}$	$\overline{c}$	1
$\overline{a}$	b	c	0
$\overline{a}$	b	$\overline{c}$	0
$\overline{a}$	$\overline{b}$	c	0
$\overline{a}$	$\overline{b}$	$\overline{c}$	0

$i_3^{\downarrow\{A\}} =$

A	
a	1
$\overline{a}$	0

$\ominus$

Theorem 3.1. *Indicator functions satisfy the axioms of a valuation algebra.*

The formal proof of this theorem will be given in Section 6.3. Instead, we directly address the search for additional properties in this valuation algebra of indicator functions.

- A neutral element for the domain s is given by $e(\mathbf{x}) = 1$ for all $\mathbf{x} \in \Omega_s$. It holds that $e_s \otimes i = e_s$ for all $i \in \Omega_s$ and the neutrality axiom is also satisfied

$$e_s \otimes e_t(\mathbf{x}) \;=\; e_s(\mathbf{x}^{\downarrow s}) \cdot e_t(\mathbf{x}^{\downarrow t}) \;=\; 1 \cdot 1 \;=\; 1 \;=\; e_{s \cup t}(\mathbf{x}).$$

  Further, neutral elements project to neutral elements again which fulfills the property of stability. Indeed, we have for $t \subseteq s$ and $\mathbf{x} \in \Omega_t$

$$e_s^{\downarrow t}(\mathbf{x}) \;=\; \max_{\mathbf{y} \in \Omega_{s-t}} e_s(\mathbf{x}, \mathbf{y}) \;=\; 1.$$

- Similarly, $z_s(\mathbf{x}) = 0$ for all $\mathbf{x} \in \Omega_s$ is the null valuation for the domain s since $z_s \otimes i = z_s$ for all $i \in \Omega_s$. The nullity axiom is also valid since only null elements project to null elements.

- Finally, indicator functions are idempotent. For $i \in \Phi_s$, $t \subseteq s$ and $\mathbf{x} \in \Omega_s$

$$i \otimes i^{\downarrow t}(\mathbf{x}) \;=\; i(\mathbf{x}) \cdot i^{\downarrow t}(\mathbf{x}^{\downarrow t}) \;=\; i(\mathbf{x}) \cdot \max_{\mathbf{y} \in \Omega_{s-t}} i(\mathbf{x}^{\downarrow t}, \mathbf{y}) \;=\; i(\mathbf{x}).$$

Altogether, indicator functions form an information algebra with null elements.

3.3 Relations

The representation of indicator functions in tabular form is clearly based on the assumption that all variable frames are finite. Alternatively (or if infinite variable frames are accepted), we can keep only those configurations that map on 1. Then, since all listed configurations take the same value, the latter can just as well be ignored. We obtain a representation of indicator functions by a set of configurations $A \subseteq \Omega_s$ as shown by the following illustration.

A	**B**	**C**	
a	b	c	0
a	b	$\overline{c}$	0
a	$\overline{b}$	c	1
a	$\overline{b}$	$\overline{c}$	1
$\overline{a}$	b	c	0
$\overline{a}$	b	$\overline{c}$	0
$\overline{a}$	$\overline{b}$	c	0
$\overline{a}$	$\overline{b}$	$\overline{c}$	0

$\rightsquigarrow$

A	**B**	**C**	
a	$\overline{b}$	c	1
a	$\overline{b}$	$\overline{c}$	1

$\rightsquigarrow$

A	**B**	**C**
a	$\overline{b}$	c
a	$\overline{b}$	$\overline{c}$

The more compact representation allows to do without the arithmetic operations in the definitions of combination and marginalization. More concretely, marginalization becomes equal to the projection rule for configuration sets given in Section 3.1 and combination simplifies to the corresponding join operator. What we have derived is a relational system, i.e. a subset of a *relational algebra* (Maier, 1983; Ullman, 1988). In fact, a relational algebra is a very fundamental formalism for representing knowledge and information. In its usual extent, at least six operations are provided

to manipulate knowledge which in turn is represented as sets of configurations. Respecting the language of relational database theory, variables are called *attributes*, configurations are *tuples*, sets of configurations are named *relations* and the given join operator is called *natural join*. The term projection exists in exactly the same manner but one often writes $\pi_t(R)$ instead of $R^{\downarrow t}$. To sum it up, relations are just another representation of indicator functions and consequently the corresponding valuation algebra satisfies the same properties.

In the algebra of relations, attributes (variables) usually take any string as value and therefore adopt frames of infinite cardinality. This is a very important point since we have seen above that neutral elements contain all possible combinations of frame values. Hence, although neutral elements exist in the valuation algebra of relations, they cannot be represented explicitly. Null elements on the other hand simply correspond to empty configuration sets.

We close this section with an example taken from (Schneuwly *et al.*, 2004) that illustrates combination and marginalization in the valuation algebra of relations.

Example 3.2. *Consider two relations R_1 and R_2 with domain $d(R_1) = \{continent, country\}$ and $d(R_2) = \{country, city\}$:*

$R_1 =$

continent	**country**
Asia	China
Asia	Japan
Europe	Germany
Europe	France

$R_2 =$

country	**city**
France	Paris
France	Lyon
Germany	Berlin
Italy	Rome

Combining R_1 and R_2 gives

$R_3 = R_1 \bowtie R_2 =$

continent	**country**	**city**
Europe	Germany	Berlin
Europe	France	Paris
Europe	France	Lyon

and we obtain for the projection of R_3 to $\{country\}$

$\pi_{\{country\}}(R_3) =$

country
Germany
France

⊖

3.4 Probability Potentials

Probability potentials are perhaps the most cited example of a valuation algebra, and their common algebraic structure with belief functions (Section 3.5) was originally the guiding theme for the abstraction process that lead to the valuation algebra framework (Shenoy & Shafer, 1988). Probability potentials represent *discrete conditional probability mass functions* which are for example used in the context of

Bayesian networks (see Section 4.2.1). However, for the algebraic study, we leave momentarily their usual interpretation and consider them as simple mappings that associate non-negative real values with configurations $\mathbf{x} \in \Omega_s$,

$$p : \Omega_s \to \mathbb{R}_{\geq 0}. \tag{3.4}$$

Note that we consider from the outset only variables with finite frames and identify $d(p) = s$ to be the domain of the potential p. Since variable frames are finite, they are usually represented in tabular-form as we proposed in the previous section.

The combination of two potentials p_1 and p_2 with domain s and t is defined as

$$p_1 \otimes p_2(\mathbf{x}) \quad = \quad p_1(\mathbf{x}^{\downarrow s}) \cdot p_2(\mathbf{x}^{\downarrow t}) \tag{3.5}$$

for $\mathbf{x} \in \Omega_{s \cup t}$. Marginalization consists in summing up all variables to be eliminated. For a potential p with domain s, $t \subseteq s$ and $\mathbf{x} \in \Omega_t$, we define

$$p^{\downarrow t}(\mathbf{x}) \quad = \quad \sum_{\mathbf{y} \in \Omega_{s-t}} p(\mathbf{x}, \mathbf{y}). \tag{3.6}$$

Both operations are illustrated by an example. Again, we abstain from an interpretation of these computations in terms of probability theory and postpone this discussion to Section 4.2.1.

Example 3.3. *Consider a set $r = \{A, B, C\}$ of variables with frames $\Omega_A = \{a, \overline{a}\}$, $\Omega_B = \{b, \overline{b}\}$ and $\Omega_C = \{c, \overline{c}\}$. Then, assume two probability potentials p_1 and p_2 with domain $d(p_1) = \{A, B\}$ and $d(p_2) = \{B, C\}$:*

$p_1 =$

A	B	
a	b	0.6
a	$\overline{b}$	0.4
$\overline{a}$	b	0.3
$\overline{a}$	$\overline{b}$	0.7

$p_2 =$

B	C	
b	c	0.2
b	$\overline{c}$	0.8
$\overline{b}$	c	0.9
$\overline{b}$	$\overline{c}$	0.1

We combine p_1 and p_2 and marginalize the result to $\{A, C\}$

$p_3 = p_1 \otimes p_2 =$

A	B	C	
a	b	c	0.12
a	b	$\overline{c}$	0.48
a	$\overline{b}$	c	0.36
a	$\overline{b}$	$\overline{c}$	0.04
$\overline{a}$	b	c	0.06
$\overline{a}$	b	$\overline{c}$	0.24
$\overline{a}$	$\overline{b}$	c	0.63
$\overline{a}$	$\overline{b}$	$\overline{c}$	0.07

$p_3^{\downarrow\{A,C\}} =$

A	C	
a	c	0.48
a	$\overline{c}$	0.52
$\overline{a}$	c	0.69
$\overline{a}$	$\overline{c}$	0.31

$\ominus$

Theorem 3.2. *Probability potentials satisfy the axioms of a valuation algebra.*

We again refer to Section 6.3 for a formal proof of this theorem and turn to the search for additional properties in the algebra of probability potentials.

- Since the combination rules for probability potentials and indicator functions are the same, we again have neutral elements in the form of $e_s(\mathbf{x}) = 1$ for all $\mathbf{x} \in \Omega_s$. However, in case of probability potentials, the stability property is no longer satisfied

$$e_s^{\downarrow t}(\mathbf{x}) \;=\; \sum_{\mathbf{y}\in\Omega_{s-t}} e_s(\mathbf{x},\mathbf{y}) \;=\; |\Omega_{s-t}|.$$

- Null elements are again defined for every domain $s \subseteq r$ and all $\mathbf{x} \in \Omega_s$ by $z_s(\mathbf{x}) = 0$. The nullity axiom is valid because probability potentials take non-negative values and therefore, every term of a zero-sum must itself be zero.

- In addition, probability potentials are regular. Let p be a potential with domain s and $t \subseteq s$. Then, the definition of regularity may be written as:

$$p(\mathbf{x}) \;=\; p^{\downarrow t} \otimes \chi \otimes p(\mathbf{x}) \;=\; p^{\downarrow t}(\mathbf{x}^{\downarrow t}) \cdot \chi(\mathbf{x}^{\downarrow t}) \cdot p(\mathbf{x}).$$

  So, defining

$$\chi(\mathbf{x}) \;=\; \begin{cases} \frac{1}{p^{\downarrow t}(\mathbf{x})} & \text{if } p^{\downarrow t}(\mathbf{x}) > 0 \\ \text{arbitrary} & \text{otherwise} \end{cases}$$

  leads naturally to a solution of this equation. Hence, the inverse of a potential p with domain s and $\mathbf{x} \in \Omega_s$ is given by

$$p^{-1}(\mathbf{x}) \;=\; \begin{cases} \frac{1}{p(\mathbf{x})} & \text{if } p(\mathbf{x}) > 0 \\ 0 & \text{otherwise.} \end{cases} \tag{3.7}$$

- Normalized probability potentials take a preferred position because they correspond to *discrete probability distributions*. Essentially, a probability potential p with domain s is said to be normalized if its values sum up to one,

$$\sum_{\mathbf{x}\in\Omega_s} p(\mathbf{x}) \;=\; 1. \tag{3.8}$$

  A given potential can be normalized using the scaling operator introduced in Section 2.7. Doing so, the scale of the probability potential p is

$$p^{\downarrow}(\mathbf{x}) \;=\; p(\mathbf{x}) \cdot \left(p^{\downarrow\emptyset}\right)^{-1}(\diamond) \;=\; \frac{p(\mathbf{x})}{\sum_{\mathbf{y}\in\Omega_s} p(\mathbf{y})}. \tag{3.9}$$

Example 3.4. *For illustration, we apply the scaling operator to the potential p_3 from Example 3.3.*

$p_3 =$

A	B	C	
a	b	c	0.12
a	b	$\overline{c}$	0.48
a	$\overline{b}$	c	0.36
a	$\overline{b}$	$\overline{c}$	0.04
$\overline{a}$	b	c	0.06
$\overline{a}$	b	$\overline{c}$	0.24
$\overline{a}$	$\overline{b}$	c	0.63
$\overline{a}$	$\overline{b}$	$\overline{c}$	0.07

$p_3^{\downarrow\emptyset} =$

$\diamond$	2.0

$p_3^{\downarrow} =$

A	B	C	
a	b	c	0.060
a	b	$\overline{c}$	0.240
a	$\overline{b}$	c	0.180
a	$\overline{b}$	$\overline{c}$	0.020
$\overline{a}$	b	c	0.030
$\overline{a}$	b	$\overline{c}$	0.120
$\overline{a}$	$\overline{b}$	c	0.315
$\overline{a}$	$\overline{b}$	$\overline{c}$	0.035

$\ominus$

3.5 Set Potentials & Belief Functions

As we have already mentioned, *belief functions* rank among the few instances that originally initiated the abstraction process culminating in the valuation algebra framework (Shenoy & Shafer, 1988). Belief functions are special cases of *set potentials* and therefore we first focus on this slightly more general formalism. The theory is again restricted to finite variables. A set potential

$$m : \mathcal{P}(\Omega_s) \to \mathbb{R}_{\geq 0} \tag{3.10}$$

with domain $d(m) = s$ is a function that assigns non-negative real numbers to all subsets of the total configuration set Ω_s.

The combination of two set potentials m_1 and m_2 with domains s and t is defined for all configuration sets $A \subseteq \Omega_{s\cup t}$ by

$$m_1 \otimes m_2(A) \quad = \sum_{B^{\uparrow s\cup t}\cap C^{\uparrow s\cup t}=A} m_1(B) \cdot m_2(C). \tag{3.11}$$

Marginalization of a set potential m with domain s to $t \subseteq s$ is given for $A \subseteq \Omega_t$ by

$$m^{\downarrow t}(A) \quad = \sum_{B^{\downarrow t}=A} m(B). \tag{3.12}$$

The application of both rules is again explained by an example. Here, it is important to remark that although all variable frames are finite, a simple enumeration of all assignments as proposed for indicator functions or probability potentials is beyond question for large domains, as a result of the power set in Equation (3.10). Moreover, only those entries are listed whose value differs from zero. These configuration sets are usually called *focal sets*.

Example 3.5. *Consider a set of variables $r = \{A, B\}$ with frames $\Omega_A = \{a, \overline{a}\}$ and $\Omega_B = \{b, \overline{b}\}$. Then, assume two set potentials m_1 and m_2 with domains $d(m_1) = \{A, B\}$ and $d(m_2) = \{A\}$:*

$$m_1 = \begin{array}{|c||c|}\hline \{(a,b)\} & 0.7 \\ \{(\overline{a},b),(a,\overline{b})\} & 0.1 \\ \{(a,b),(\overline{a},\overline{b})\} & 0.2 \\ \hline \end{array} \qquad m_2 = \begin{array}{|c||c|}\hline \emptyset & 0.6 \\ \{(a)\} & 0.4 \\ \hline \end{array}$$

The task of computing the combination of m_1 and m_2 is simplified by constructing the following table as an intermediate step. The first column contains m_1 and the top row m_2, both extended to the union domain $d(m_1) \cup d(m_2) = \{A, B\}$. Then, every internal cell consists of the intersection between the two corresponding configuration sets and the multiplication of the corresponding values.

	$\emptyset$, 0.6	$\{(a,b),(a,\overline{b})\}$, 0.4
$\{(a,b)\}$, 0.7	$\emptyset$, 0.42	$\{(a,b)\}$, 0.28
$\{(\overline{a},b),(a,\overline{b})\}$, 0.1	$\emptyset$, 0.06	$\{(a,\overline{b})\}$, 0.04
$\{(a,b),(\overline{a},\overline{b})\}$, 0.2	$\emptyset$, 0.12	$\{(a,b)\}$, 0.08

To complete the combination, it is then sufficient to add the values of all internal cells with equal configuration set. The result of the combination is marginalized afterwards to $\{A\}$

$$m_3 = m_1 \otimes m_2 = \begin{array}{|c||c|}\hline \emptyset & 0.6 \\ \{(a,b)\} & 0.36 \\ \{(a,\overline{b})\} & 0.04 \\ \hline \end{array} \qquad m_3^{\downarrow\{A\}} = \begin{array}{|c||c|}\hline \emptyset & 0.6 \\ \{(a)\} & 0.4 \\ \hline \end{array}$$

⊖

Theorem 3.3. *Set potentials satisfy the axioms of a valuation algebra.*

The proof of this theorem will be given in Section 7.1. Here, we now look for further properties in the valuation algebra of set potentials.

- The neutral element e_s for the domain s is given by $e_s(A) = 0$ for all proper subsets $A \subset \Omega_s$ and $e_s(\Omega_s) = 1$. The properties of neutrality and stability both hold in this valuation algebra, as shown in Section 7.2.
- The null element z_s for the domain $s \subseteq r$ is defined by $z_s(A) = 0$ for all $A \subseteq \Omega_s$. Since marginalization again consists of summing up non-negative values, the nullity axiom is clearly satisfied.

Before actually looking for inverse valuations, we first introduce some alternative ways of representing set potentials. We define for a set potential m

$$b_m(A) = \sum_{B \subseteq A} m(B) \quad \text{and} \quad q_m(A) = \sum_{B \supseteq A} m(B). \tag{3.13}$$

Clearly, both functions b_m and q_m are themselves set potentials and (Shafer, 1976) shows that the two transformation rules are one-to-one with the following inverse transformations

$$m(A) = \sum_{B \subseteq A} (-1)^{|A-B|} b_m(B) \tag{3.14}$$

and

$$m(A) = \sum_{B \supseteq A} (-1)^{|B-A|} q_m(B). \tag{3.15}$$

Consequently, the operations of combination and marginalization can be carried over from m-functions to b-functions and q-functions:

$$\begin{aligned} b_{m_1} \otimes b_{m_2} &= b_{m_1 \otimes m_2}, & b_m^{\downarrow t} &= b_{m^{\downarrow t}}, \\ q_{m_1} \otimes q_{m_2} &= q_{m_1 \otimes m_2}, & q_m^{\downarrow t} &= q_{m^{\downarrow t}}. \end{aligned}$$

To sum it up, m-functions, b-functions and q-functions describe the same system, and since m-functions build a valuation algebra, the same holds for the two others. Additionally, we also deduce that all properties that hold for one of these representations naturally hold for the two others, which is similar to the considerations about indicator functions and relations.

It turns out that the operations of combination and marginalization simplify considerably if they are expressed either in the system of q-functions or b-functions. More concretely, combination reduces to simple multiplication in the system of q-functions and marginalization does not need any computation at all in the system of b-functions. This is the statement of the following theorem which is proved for example in (Kohlas, 2003).

Theorem 3.4.

1. *For q_{m_1} with domain s and q_{m_2} with domain t we have for all $A \subseteq \Omega_{s \cup t}$*

$$q_{m_1} \otimes q_{m_2}(A) = q_{m_1}(A^{\downarrow s}) \cdot q_{m_2}(A^{\downarrow t}). \tag{3.16}$$

2. *For b_m with domain s and $t \subseteq s$ we have for all $A \subseteq \Omega_t$*

$$b_m^{\downarrow t}(A) = b_m(A^{\uparrow s}). \tag{3.17}$$

Next, it will be shown that set potentials are separative. For this purpose, we change into the system of q-functions. Then, a congruence γ is needed that divides Φ into cancellative equivalence classes. Let us therefore introduce the *support* of a q-function $q \in \Phi_s$ by

$$supp(q) = \{A \subseteq \Omega_s : q(A) > 0\}. \tag{3.18}$$

q_1 and q_2 with equal domain are equivalent if they have the same support, i.e.

$$q_1 \equiv q_2 \pmod{\gamma} \quad \text{if} \quad d(q_1) = d(q_2) \wedge supp(q_1) = supp(q_2). \tag{3.19}$$

In particular, we have $q \equiv q \otimes q^{\downarrow t} \pmod{\gamma}$ since

$$\begin{aligned} supp(q \otimes q^{\downarrow t}) &= \{A \subseteq \Omega_s : q(A) \cdot q^{\downarrow t}(A^{\downarrow t}) > 0\} \\ &= \{A \subseteq \Omega_s : q(A) > 0\} \\ &= supp(q). \end{aligned}$$

The second equality holds because q-functions are non-negative and consequently the terms of a sum giving zero must themselves be zero, i.e. $q^{\downarrow t}(A^{\downarrow t}) = 0$ implies $q(A) = 0$. Hence, Φ is divided into disjoint equivalence classes of q-functions with equal domain and support. These classes are clearly cancellative since

$$q(A) \cdot q_1(A) = q(A) \cdot q_2(A)$$

implies that $q_1(A) = q_2(A)$ if q, q_1, q_2 all have the same domain and support. This fulfils all requirements for separativity given in Definition 2.11.

It should be mentioned that not every b-function or q-function transforms into a set potential m using either Equation (3.14) or (3.15). This is because both transformations can create negative values which in turn means that Φ^* is really an extension of the system of q-functions.

A set potential m with domain s is called *normalized*, if

$$m(\emptyset) = 0 \quad \text{and} \quad \sum_{A \subseteq \Omega_s} m(A) = 1. \tag{3.20}$$

In the theory of belief functions, such potentials are called *basic probability assignments* (bpa) or *mass functions*. Intuitively, $m(A)$ represents the part of our belief that the actual world (configuration) belongs to A – without supporting any more specific subset, by lack of adequate information (Smets, 2000). Under this interpretation, the two normalization conditions imply that no belief is held in the empty configuration set and that the total belief has measure 1. Regarding Example 3.5, only factor m_1 is normalized but not m_2. The b-function obtained by transforming a mass function is called *belief function*. Essentially, it is the sum of all values that are assigned to subsets of A. Finally, q-functions are named *commonality functions* and $q(A)$ consists of the sum of all values that are assigned to configuration sets which contain A. We refer to (Shafer, 1976) for a broad discussion of all these concepts.

A given set potential can be normalized using the scaling operator introduced in Section 2.7. For this purpose, we again consider the representation as q-function and obtain for the definition of its scale

$$q^{\downarrow} = q \otimes \left(q^{\downarrow\emptyset}\right)^{-1}.$$

It is important to note that $q^{\downarrow\emptyset}$ consists of two values, $q^{\downarrow\emptyset}(\emptyset)$ and $q^{\downarrow\emptyset}(\{\diamond\})$. Since the empty set can never be obtained from projecting a non-empty configuration set, the formula can be written for the case where $A \neq \emptyset$ and $q^{\downarrow\emptyset}(\{\diamond\}) \neq 0$ as

$$q^{\downarrow}(A) = \frac{q(A)}{q^{\downarrow\emptyset}(\{\diamond\})}. \tag{3.21}$$

For the denominator we obtain

$$q^{\downarrow\emptyset}(\{\diamond\}) \;=\; m^{\downarrow\emptyset}(\{\diamond\}) \;=\; \sum_{A\neq\emptyset} m(A). \tag{3.22}$$

Since

$$\begin{aligned} q^{\downarrow\emptyset}(\emptyset) \;=\; \sum_{A\subseteq\Omega_\emptyset} m^{\downarrow\emptyset}(A) \;&=\; m^{\downarrow\emptyset}(\{\diamond\}) + m^{\downarrow\emptyset}(\emptyset) \\ &=\; \sum_{A\neq\emptyset} m(A) + m(\emptyset) \;=\; \sum_{A} m(A) \;=\; q(\emptyset), \end{aligned}$$

we have for the empty set

$$q^{\downarrow}(\emptyset) \;=\; \frac{q(\emptyset)}{q^{\downarrow\emptyset}(\emptyset)} \;=\; \frac{q(\emptyset)}{q(\emptyset)} \;=\; 1. \tag{3.23}$$

The operator of scaling is translated to m-functions by

$$m^{\downarrow}(A) \;=\; \sum_{B\supseteq A} (-1)^{|B-A|}\; q^{\downarrow}(B) \tag{3.24}$$

$$=\; \frac{\sum_{B\supseteq A}(-1)^{|B-A|}\; q(B)}{\sum_{B\neq\emptyset} m(B)} \;=\; \frac{m(A)}{\sum_{B\neq\emptyset} m(B)} \;=\; \frac{m(A)}{m^{\downarrow\emptyset}(\{\diamond\})},$$

if $A \neq \emptyset$ and $m^{\downarrow\emptyset}(\{\diamond\}) \neq 0$. Finally, we obtain for $m^{\downarrow\emptyset}(\emptyset)$

$$\begin{aligned} m^{\downarrow\emptyset}(\emptyset) \;=\; \sum_{A}(-1)^{|A|}\; q^{\downarrow}(A) \;&=\; \frac{\sum_{A\neq\emptyset}(-1)^{|A|}\; q(A)}{\sum_{A\neq\emptyset} m(A)} + 1 \\ &=\; \frac{\sum_{A}(-1)^{|A|}\; q(A) - q(\emptyset)}{\sum_{A\neq\emptyset} m(A)} + 1 \\ &=\; \frac{m(\emptyset)}{\sum_{A\neq\emptyset} m(A)} - \frac{m(\emptyset) + \sum_{A\neq\emptyset} m(A)}{\sum_{A\neq\emptyset} m(A)} + 1 \;=\; 0. \end{aligned} \tag{3.25}$$

Thus, we can see that the result of applying the scaling operator is a normalized set potential corresponding to Equation (3.20). This procedure of scaling is illustrated in the following Example.

Example 3.6.

$m =$

$\emptyset$	0.6
$\{(a,b)\}$	0.1
$\{(\overline{a},b),(a,\overline{b})\}$	0.1
$\{(a,b),(\overline{a},\overline{b})\}$	0.2

$m^{\downarrow\emptyset} =$

$\emptyset$	0.6
$\{\diamond\}$	0.4

$m^{\downarrow} =$

$\emptyset$	0
$\{(a,b)\}$	0.25
$\{(\overline{a},b),(a,\overline{b})\}$	0.25
$\{(a,b),(\overline{a},\overline{b})\}$	0.50

$\ominus$

We know from various occasions that the combination of two normalized set potentials (mass functions) does generally not lead to a mass function again. However, this can be achieved using Equation (2.47). Assume two mass functions m_1 and m_2 with domains s and t. We have for $\emptyset \subset A \subseteq \Omega_{s\cup t}$

$$m_1 \oplus m_2(A) \;=\; (m_1 \otimes m_2)^{\downarrow}(A) \;=\; \frac{m_1 \otimes m_2(A)}{(m_1 \otimes m_2)^{\downarrow\emptyset}(\{\diamond\})} \;=\; \frac{1}{K}\, m_1 \otimes m_2(A),$$

where

$$\begin{aligned} K &= (m_1 \otimes m_2)^{\downarrow\emptyset}(\{\diamond\}) \\ &= \sum_{B\neq\emptyset} m_1 \otimes m_2(B) \;=\; \sum_{A_1^{\uparrow s\cup t}\cap A_2^{\uparrow s\cup t}\neq\emptyset} m_1(A_1)\cdot m_2(A_2). \end{aligned}$$

We define $m_1 \oplus m_2(\emptyset) = 0$ and if $K = 0$, we set $m_1 \oplus m_2 = z_{s\cup t}$. This operation is called *Dempster's rule of combination* (Shafer, 1976).

3.6 Distance Potentials

Although *distance tables* are an often used formalism in many fields of computer science, they have so far never been studied in the context of valuation algebras. The precursor for this investigation was a master's thesis of Antoine de Groote (de Groote, 2006) where we showed for the very first time how the valuation algebra operations can be defined for this formalism.

Again, let r be a finite set of variables. Due to the close relationship with graph theory, these variables may be considered as graph vertices. Then, a *path length function* $\delta(X, Y)$ expresses the length of the edge between two vertices X and Y that belong to some subset $s \subseteq r$. The set s is called the *domain* of δ and we write $d(\delta) = s$. Naturally, δ is a non-negative function with $\delta(X, Y) = 0$ exactly if $X = Y$. Further, we write by convention $\delta(X, Y) = \infty$ if no edge between X and Y exists. Note also that δ is not assumed to be symmetric. This allows for example to model road maps which possibly contain one-way roads.

The marginal of a path length function δ to a domain $t \subseteq d(\delta)$ is given by

$$\delta^{\downarrow t}(X, Y) \;=\; \delta(X, Y) \tag{3.26}$$

for $X, Y \in t$. If path length functions are represented as square matrices, marginalization consists simply in dropping those rows and columns whose variables are not contained in t. Conversely, the vacuous extension of δ to some superdomain $u \supseteq d(\delta)$ is defined by

$$\delta^{\uparrow u}(X, Y) \;=\; \begin{cases} \delta(X, Y) & \text{if } X, Y \in d(\delta), \\ 0 & \text{if } X = Y, \\ \infty & \text{otherwise.} \end{cases} \tag{3.27}$$

(Bovet & Crescenzi, 1994) propose a dynamic programming technique to compute the shortest distance between any two nodes within the domain s of a path length function δ. To this end, we define for $n \in \mathbb{N}$

$$\delta^n(X,Y) \;=\; \begin{cases} \delta(X,Y) & \text{if } n = 1, \\ \min_{Z \in s}(\delta(X,Z) + \delta^{n-1}(Z,Y)) & \text{otherwise.} \end{cases} \tag{3.28}$$

$\delta^1 = \delta$ contains all direct path lengths. Proceeding one step, $\delta^2(X,Y)$ measures the minimum path length from X to Y that goes through at most one intermediate node. Then, $\delta^3(X,Y)$ is the minimum path length from X to Y going through at most two intermediate nodes, and so on. If δ includes $m = |d(\delta)|$ nodes, δ^{m-1} contains the shortest distances between any two nodes in the domain of δ. In other words, δ^{m-1} is a *fixpoint* of this iteration process, i.e. we have

$$\delta^k \;=\; \delta^{m-1} \tag{3.29}$$

for all $k \geq m-1$. These fixpoints are called *distance potentials* and constitute the valuations under consideration. (Bovet & Crescenzi, 1994) furthermore come up with a particularly efficient way to compute a distance potential from a given path length function δ. For $n = 2k$, it holds that

$$\delta^n(X,Y) \;=\; \min_{Z \in s}\left(\delta^{n/2}(X,Z) + \delta^{n/2}(Z,Y)\right) \;=\; \left(\delta^{n/2}\right)^2(X,Y). \tag{3.30}$$

This allows to compute δ^n with only a logarithmic number of iterations.

We subsequently identify distance potentials by lower case greek letters. Further, it is natural to define $d(\phi) = s$ if ϕ is the distance potential obtained from a path length function δ with domain s. We therefore adopt the usual notation of Φ_s for the set of all distance potentials with domain s and Φ for all distance potentials over subsets of r. Also, we point out that we do not insist on the above construction of distance potentials from path length functions. Moreover, we call ϕ a distance potential if the following requirements are satisfied:

- $\phi(X,Y) \geq 0$ for all $X, Y \in d(\phi)$,
- $\phi(X,Y) = 0$ if, and only if, $X = Y$,
- $\phi(X,Z) \;\leq\; \phi(X,Y) + \phi(Y,Z)$ for $X, Y, Z \in d(\phi)$.

Distance potentials are therefore *quasimetrics*. If furthermore symmetry holds, i.e. $\phi(X,Y) = \phi(Y,X)$ for all $X, Y \in s$, ϕ becomes a *distance* or *metric*. But this is generally not the case.

The following lemma states that fixpoint iteration and vacuous extension are interchangeable for path length functions.

Lemma 3.5. *For a path length function δ with $d(\delta) \subseteq u \subseteq r$ and $n \in \mathbb{N}$ we have*

$$\left(\delta^n\right)^{\uparrow u} \;=\; \left(\delta^{\uparrow u}\right)^n. \tag{3.31}$$

Proof. For $n = 1$, this holds naturally since $\delta^1 = \delta$. We proceed by induction and assume that the property holds for n. We then have for $X, Y \in u$

$$\begin{aligned}\left(\delta^{\uparrow u}\right)^{n+1}(X,Y) &= \min_{Z\in u}\left(\delta^{\uparrow u}(X,Z) + (\delta^{\uparrow u})^n(Z,Y)\right)\\ &= \min_{Z\in u}\left(\delta^{\uparrow u}(X,Z) + (\delta^n)^{\uparrow u}(Z,Y)\right).\end{aligned}$$

We distinguish the following two cases:

1. If either $X \in u - d(\delta)$ or $Y \in u - d(\delta)$, then it follows from (3.27) that

$$\min_{Z\in u}\left(\delta^{\uparrow u}(X,Z) + (\delta^n)^{\uparrow u}(Z,Y)\right) = \infty.$$

2. If $X, Y \in d(\delta)$, we can limit the search for Z to $d(\delta)$ due to (3.27). Therefore,

$$\begin{aligned}\min_{Z\in u}\left(\delta^{\uparrow u}(X,Z) + (\delta^n)^{\uparrow u}(Z,Y)\right) &= \min_{Z\in d(\delta)}\left(\delta^{\uparrow u}(X,Z) + (\delta^n)^{\uparrow u}(Z,Y)\right)\\ &= \min_{Z\in d(\delta)}\left(\delta(X,Z) + \delta^n(Z,Y)\right)\\ &= \delta^{n+1}(X,Y).\end{aligned}$$

Putting all together and applying (3.27), we finally obtain

$$\left(\delta^{\uparrow u}\right)^{n+1}(X,Y) = \left\{\begin{array}{ll}\delta^{n+1}(X,Y) & \text{if } X,Y \in d(\delta)\\ 0 & \text{if } X = Y,\\ \infty & \text{otherwise}\end{array}\right\} = \left(\delta^{n+1}\right)^{\uparrow u}(X,Y).$$

□

The marginals of a distance potential are obtained similarly to those of path length functions. Combination of two distance potentials ϕ and ψ with domains $d(\phi) = s$ and $d(\psi) = t$ is defined by

$$\phi \otimes \psi = \left[\min\left(\phi^{\uparrow s\cup t}, \psi^{\uparrow s\cup t}\right)\right]^n \tag{3.32}$$

for $n = |s \cup t| - 1$. Here, we take the component-wise minimum of ϕ and ψ which are first extended to their common domain. Since the result is no longer a distance potential, we recover this property by a new fixpoint iteration.

For later use, we also remark that δ and δ^m both have the same fixpoint, since

$$\delta^n = \delta^{n+m} = (\delta^m)^n$$

for arbitrary $m \in \mathbb{N}$ and $n = |d(\delta)| - 1$. Then we conclude without presenting a formal proof that for δ_1 and δ_2 with $d(\delta_1) = d(\delta_2) = s$, the two path length functions $\min(\delta_1, \delta_2)$ and $\min(\delta_1^m, \delta_2)$ also converge to the same fixpoint. In other words,

$$\min(\delta_1, \delta_2)^{|s|-1} = \min(\delta_1^m, \delta_2)^{|s|-1}. \tag{3.33}$$

The following example taken from (de Groote, 2006) illustrates the operations of combination and marginalization for distance potentials.

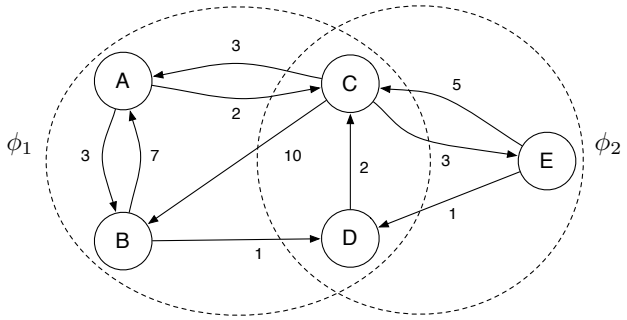


Figure 3.1: A graph with vertices $r = \{A, B, C, D, E\}$ and two distance potentials ϕ_1 and ϕ_2 with domains $d(\phi_1) = \{A, B, C, D\}$ and $d(\phi_2) = \{C, D, E\}$.

Example 3.7. *We consider the two distance potentials extracted from Figure 3.1, i.e. the lengths of the shortest paths within the corresponding scopes:*

$\phi_1 =$

	A	**B**	**C**	**D**
A	0	3	2	4
B	6	0	3	1
C	3	6	0	7
D	5	8	2	0

$\phi_2 =$

	C	**D**	**E**
C	0	4	3
D	2	0	5
E	3	1	0

First, we compute

$\delta = \min(\phi_1^{\uparrow\{A,B,C,D,E\}}, \phi_2^{\uparrow\{A,B,C,D,E\}}) =$

	A	**B**	**C**	**D**	**E**
A	0	3	2	4	∞
B	6	0	3	1	∞
C	3	6	0	4	3
D	5	8	2	0	5
E	∞	∞	3	1	0

and remark immediately that δ is not a distance potential anymore since

$$\delta(A, C) + \delta(C, E) = 5 < \delta(A, E) = \infty.$$

We compute the fixpoint $\phi_3 = \phi_1 \otimes \phi_2 = \delta^4$ and marginalize the result to $\{B, C\}$

$\phi_3 = \phi_1 \otimes \phi_2 =$

	A	**B**	**C**	**D**	**E**
A	0	3	2	4	5
B	6	0	3	1	6
C	3	6	0	4	3
D	5	8	2	0	5
E	6	9	3	1	0

$\phi_3^{\downarrow\{B,C\}} =$

	B	**C**
B	0	3
C	6	0

$\ominus$

Theorem 3.6. *Distance potentials satisfy the axioms of a valuation algebra.*

Proof. We will verify the axioms of a valuation algebra given in Section 2.1. With the interpretation of marginalization as the elimination of rows and columns, we directly conclude that the axioms of marginalization (A3), transitivity (A4) and domain (A6) are satisfied. Furthermore, the labeling axiom follows from Equation (3.32) and we therefore restrict ourselves to a formal proof of Axioms (A1) and (A5).

(A1) *Commutative Semigroup:* Commutativity of combination follows directly from its definition since component-wise minimization is clearly commutative. To prove associativity, we assume three distance potentials ϕ, ψ and χ with domains $d(\phi) = s$, $d(\psi) = t$ and $d(\chi) = u$. Applying the definition of combination and Lemma 3.5 we obtain

$$\begin{aligned}
(\phi \otimes \psi) \otimes \chi &= \min\left(\phi^{\uparrow s\cup t}, \psi^{\uparrow s\cup t}\right)^{|s\cup t|-1} \otimes \chi \\
&= \min\left[\left(\min(\phi^{\uparrow s\cup t}, \psi^{\uparrow s\cup t})^{|s\cup t|-1}\right)^{\uparrow s\cup t\cup u}, \chi^{\uparrow s\cup t\cup u}\right]^{|s\cup t\cup u|-1} \\
&= \min\left[\left(\min(\phi^{\uparrow s\cup t}, \psi^{\uparrow s\cup t})^{\uparrow s\cup t\cup u}\right)^{|s\cup t|-1}, \chi^{\uparrow s\cup t\cup u}\right]^{|s\cup t\cup u|-1} \\
&= \min\left[\min(\phi^{\uparrow s\cup t\cup u}, \psi^{\uparrow s\cup t\cup u})^{|s\cup t|-1}, \chi^{\uparrow s\cup t\cup u}\right]^{|s\cup t\cup u|-1}.
\end{aligned}$$

Next, we apply Equation (3.33),

$$\begin{aligned}
&= \min\left[\min(\phi^{\uparrow s\cup t\cup u}, \psi^{\uparrow s\cup t\cup u}), \chi^{\uparrow s\cup t\cup u}\right]^{|s\cup t\cup u|-1} \\
&= \min\left[\phi^{\uparrow s\cup t\cup u}, \psi^{\uparrow s\cup t\cup u}, \chi^{\uparrow s\cup t\cup u}\right]^{|s\cup t\cup u|-1}.
\end{aligned}$$

In a similar way, we can derive exactly the same expression from $\phi \otimes (\psi \otimes \chi)$ which proves associativity.

(A5) *Combination:* Assume two distance potentials ϕ and ψ with domains $d(\phi) = x$, $d(\psi) = y$ and $x \subseteq z \subseteq x \cup y$. We have

$$\begin{aligned}
\min(\phi^{\uparrow x\cup y}, \psi^{\uparrow x\cup y}) &\leq \min(\phi^{\uparrow x\cup y}, (\psi^{\downarrow y\cap z})^{\uparrow x\cup y}) \\
&= \min(\phi^{\uparrow z}, (\psi^{\downarrow y\cap z})^{\uparrow z})^{\uparrow x\cup y}.
\end{aligned}$$

The relation $\leq$ applies component-wise and the inequality holds because marginalization followed by vacuous extension consists in replacing some values by ∞. This carries over to their fixpoints and we conclude from Lemma 3.5 that

$$\begin{aligned}
\min(\phi^{\uparrow x\cup y}, \psi^{\uparrow x\cup y})^{|x\cup y|-1} &\leq \left(\min(\phi^{\uparrow z}, (\psi^{\downarrow y\cap z})^{\uparrow z})^{\uparrow x\cup y}\right)^{|x\cup y|-1} \\
&= \left(\min(\phi^{\uparrow z}, (\psi^{\downarrow y\cap z})^{\uparrow z})^{|x\cup y|-1}\right)^{\uparrow x\cup y} \\
&= \left(\min(\phi^{\uparrow z}, (\psi^{\downarrow y\cap z})^{\uparrow z})^{|z|-1}\right)^{\uparrow x\cup y}.
\end{aligned}$$

Marginalizing both sides of the inequality to z gives

$$\Big(\min(\phi^{\uparrow x\cup y}, \psi^{\uparrow x\cup y})^{|x\cup y|-1}\Big)^{\downarrow z} \quad \leq \quad \min(\phi^{\uparrow z}, (\psi^{\downarrow y\cap z})^{\uparrow z})^{|z|-1},$$

which is equivalent to

$$(\phi \otimes \psi)^{\downarrow z} \quad \leq \quad \phi \otimes \psi^{\downarrow y\cap z}.$$

Let us now assume that the inequality is strict for some $X, Y \in z$, thus

$$\min(\phi^{\uparrow x\cup y}, \psi^{\uparrow x\cup y})^{|x\cup y|-1}(X, Y) \quad < \quad \min(\phi^{\uparrow z}, (\psi^{\downarrow y\cap z})^{\uparrow z})^{|z|-1}(X, Y).$$

The left hand side of this inequality is equal to

$$\min_{Q\in x\cup y}\big(\min(\phi^{\uparrow x\cup y}, \psi^{\uparrow x\cup y})(X, Q) + \min(\phi^{\uparrow x\cup y}, \psi^{\uparrow x\cup y})^{|x\cup y|-2}(Q, Y)\big).$$

Observe that within the domain z, both path length functions $\min(\phi^{\uparrow x\cup y}, \psi^{\uparrow x\cup y})$ and $\min(\phi^{\uparrow z}, (\psi^{\downarrow y\cap z})^{\uparrow z})$ are equal. Therefore, Q must be contained in the domain $y - (y \cap z)$ and the expression simplifies to

$$\psi^{\uparrow x\cup y}(X, Q) + \min(\phi^{\uparrow x\cup y}, \psi^{\uparrow x\cup y})^{|x\cup y|-2}(Q, Y).$$

Somewhat informally, we may thus say that the shorter path from X to Y leaves the domain z. But then, since $Y \in z$, this path must somewhere return into the domain z. In other words, there must be some entry $\psi(U, V) < \infty$ with $U \in y - (y \cap z)$ and $V \in y \cap z$ that is part of the shorter path from X to Y. But ψ is a shortest distance function which implies that $\psi(X, V)$ contains the total length of this path via Q and U. Additionally, $X, Y \in z$ and consequently this distance is also contained in the right hand expression of the above inequality. We conclude that

$$\min(\phi^{\uparrow x\cup y}, \psi^{\uparrow x\cup y})^{|x\cup y|-1}(X, Y) \quad = \quad \min(\phi^{\uparrow z}, (\psi^{\downarrow y\cap z})^{\uparrow z})^{|z|-1}(X, Y).$$

which contradicts the assumption of strict inequality. This proves the combination axiom.

□

Next, we look for further properties in this algebra of distance potentials.

- The neutral element $e_s \in \Phi_s$ is given for $X, Y \in s$ by

$$e_s(X, Y) \;=\; \begin{cases} 0 & \text{if } X = Y, \\ \infty & \text{otherwise.} \end{cases} \tag{3.34}$$

Indeed, for all $\phi \in \Phi_s$ we have

$$e_s \otimes \phi \;=\; \min(e_s, \phi)^{|s|-1} \;=\; \phi^{|s|-1} \;=\; \phi.$$

The neutrality axiom holds naturally, and in the same breath we remark that the stability property is also satisfied.

- The valuation algebra of distance potentials is idempotent. For $\phi \in \Phi_s$ and $t \subseteq s$ we have

$$\phi \otimes \phi^{\downarrow t} \;=\; \min\left(\phi, (\phi^{\downarrow t})^{\uparrow s}\right)^{|s|-1} \;=\; \phi^{|s|-1} \;=\; \phi.$$

- Finally, the valuation algebra of distance potentials does not possess null elements because the definition $z_s(X, Y) = 0$ for all $X, Y \in s$ would violate the requirements for a quasidistance.

3.7 Densities & Gaussian Potentials

Another popular valuation algebra instance is that of density functions. Consider an index set $r = \{1, \ldots, n\}$. Then, a *density function* $f : \mathbb{R}^{|s|} \to \mathbb{R}$ with domain $d(f) = s \subseteq r$ is a continuous, non-negative valued function whose integral is finite,

$$\int_{-\infty}^{\infty} f(\mathbf{x}) d\mathbf{x} \;<\; \infty. \tag{3.35}$$

Here, $\mathbf{x} \in \Omega_s$ denotes a vector of dimension $|s|$ consisting of real numbers. Again, we denote by Φ_s the set of all densities with domain s and write Φ for all densities over subsets of r.

The combination of two densities $f \in \Phi_s$ and $g \in \Phi_t$ is defined for $\mathbf{x} \in \Omega_{s \cup t}$ by

$$f \otimes g(\mathbf{x}) \;=\; f(\mathbf{x}^{\downarrow s}) \cdot g(\mathbf{x}^{\downarrow t}). \tag{3.36}$$

Additionally, we define the marginalization of a density f with domain $d(f) = s$ to $t \subseteq s$ by the integral

$$f^{\downarrow t}(\mathbf{x}) \;=\; \int_{-\infty}^{\infty} f(\mathbf{x}, \mathbf{y}) d\mathbf{y}, \tag{3.37}$$

for $\mathbf{x} \in \Omega_t$ and $\mathbf{y} \in \Omega_{s-t}$.

Theorem 3.7. *Density functions satisfy the axioms of a valuation algebra.*

We refer to (Kohlas, 2003) for a formal verification of the axiomatic system and continue by searching for further properties within this algebra of density functions.

- Since combination corresponds to multiplication, $e_s(\mathbf{x}) = 1$ for $\mathbf{x} \in \Omega_s$ would be the candidate for the neutral element in Φ_s. However, this is clearly not a density function since its integral is infinite. Therefore, densities form a valuation algebra without neutral elements.

- On the other hand, the null element for the domain s is given by $z_s(\mathbf{x}) = 0$ for all $\mathbf{x} \in \Omega_s$. This is a density function with respect to our definition and since densities are non-negative, the nullity axiom is also satisfied.

- In order to show that density functions are separative, we proceed similarly to commonality functions in Section 3.5 and define the *support* of a density $f \in \Phi_s$ by

$$supp(f) \quad = \quad \{\mathbf{x} \in \Omega_s : f(\mathbf{x}) > 0\}. \tag{3.38}$$

  Then we may say that two densities f and g with equal domain are equivalent if they have the same support, i.e.

$$f \equiv g \pmod{\gamma} \quad \text{if} \quad d(f) = d(g) \wedge supp(f) = supp(g). \tag{3.39}$$

  Again, we have $f \equiv f \otimes f^{\downarrow t} \pmod{\gamma}$ since

$$\begin{aligned} supp(f \otimes f^{\downarrow t}) &= \{\mathbf{x} \in \Omega_s : f(\mathbf{x}) \cdot f^{\downarrow t}(\mathbf{x}^{\downarrow t}) > 0\} \\ &= \{\mathbf{x} \in \Omega_s : f(\mathbf{x}) > 0\} \\ &= supp(f). \end{aligned}$$

  The second equality holds because densities are non-negative and therefore $f^{\downarrow t}(\mathbf{x}^{\downarrow t}) = 0$ implies $f(\mathbf{x}) = 0$. Hence, Φ is divided into disjoint equivalence classes of densities with equal domain and support. These classes are cancellative, which fulfils all requirements for separativity given in Definition 2.11.

- Continuous probability distributions are a particularly important class of density functions and spark our interest in normalized densities. According to Equation (2.44), the scale of a density function $f \in \Phi_s$ is obtained by

$$f^{\downarrow}(\mathbf{x}) \quad = \quad f(\mathbf{x}) \otimes \left(f^{\downarrow \emptyset}\right)^{-1}(\diamond) \quad = \quad \frac{f(\mathbf{x})}{\int f(\mathbf{x}) d\mathbf{x}} \tag{3.40}$$

  for $\mathbf{x} \in \Omega_s$. Consequently, we have for a scaled density

$$\int f^{\downarrow}(\mathbf{x}) d\mathbf{x} \quad = \quad 1. \tag{3.41}$$

The definition of density functions given above only requires a finite integral and does not narrow down how exactly the function is shaped. A notably important class of such density functions are *Gaussian densities* which model multidimensional, continuous probability distributions. For $\Omega_s = \mathbb{R}^{|s|}$, they take the form

$$f(\mathbf{x}) \quad = \quad \sqrt{\frac{|\det(K)|}{(2\pi)^n}} e^{-\frac{1}{2}(\mathbf{x}-\mu)^T K(\mathbf{x}-\mu)}. \tag{3.42}$$

Here, μ denotes the *mean vector* in $\mathbb{R}^{|s|}$, and K stands for the *concentration matrix* associated with the *variance-covariance matrix* Σ. We have $K = \Sigma^{-1}$ where both matrices are symmetric and positive definite in $\mathbb{R}^{|s|} \times \mathbb{R}^{|s|}$.

Looking at the above definition, we quickly remark that a Gaussian density with domain s is completely determined by its mean vector and concentration matrix.

Therefore, we define a *Gaussian potential* with domain s by the pair (μ, K) where $\mu \in \mathbb{R}^{|s|}$ and $K \in \mathbb{R}^{|s|} \times \mathbb{R}^{|s|}$. These Gaussian potentials are closed under combination and marginalization and form a separative sub-algebra of the valuation algebra of densities. Interestingly, this valuation algebra does not even possess null elements, due to the claim for a positive definite concentration matrix. Finally, we observe that Gaussian potentials are naturally scaled:

$$(\mu, K)^{\downarrow} \;=\; (\mu, K) \otimes (\mu, K)^{\downarrow\emptyset} \;=\; (\mu, K) \otimes (\diamond, \diamond) \;=\; (\mu, K).$$

A profound study of Gaussian potentials as valuation algebra instance can be found in (Eichenberger, 2007).

3.8 Propositional Logic

Propositional logic ranks among the most traditional formalisms for knowledge representation. We give a very short introduction to propositional calculus, confined to the concepts that are required to understand propositional logic as valuation algebra instance. It should be mentioned that multiple approaches exist for deriving a valuation algebra from propositional logic, and they can essentially be grouped into two perceptions which we call model and language level.

The language of propositional logic $\mathcal{L}$ is constructed over a finite set of propositional variables r (also called *propositions* and denoted by $X, Y, \ldots$) and consists of all *well-formed formulae* or *sentences* defined in the following way:

1. Each proposition as well as $\top$ and $\bot$ are well-formed formulae.
2. If γ and δ are well-formed, then so are $\neg\gamma$, $\gamma \vee \delta$, $\gamma \wedge \delta$, $\gamma \rightarrow \delta$ and $\gamma \leftrightarrow \delta$.

Thus, all elements of $\mathcal{L}$ are generated by starting with Rule 1 and applying Rule 2 finitely many times. Further, we omit parenthesis using the following precedence order of negation and junctors: $\neg, \wedge, \vee$. It is seldom the case that all propositions of r do indeed occur in a given sentence. On this account, we denote by $vars(\gamma)$ all propositions that are actually contained in a sentence $\gamma \in \mathcal{L}$. Note that $vars(\top) = vars(\bot) = \emptyset$.

The meaning of a propositional sentence γ with $vars(\gamma) = s$ is defined inductively by first assuming a mapping $i : s \rightarrow \{0, 1\}$ that assigns a truth value to all propositions in s. Such a mapping is called *interpretation* relative to s. Then, the extended truth value $i(\gamma)$ for sentences $\gamma \in \mathcal{L}$ is obtained according to Figure 3.2. Interpretations under which γ evaluates to 1 are called *models* of γ, and we write $I(\gamma)$ for the set of all models of $\gamma \in \mathcal{L}$. It is self-evident that model sets correspond to sets of configurations as introduced in Section 3.1. If on the other hand γ evaluates to 0, then we refer to the corresponding interpretation as *counter-models*. Further, a sentence $\gamma \in \mathcal{L}$ is called *satisfiable* if its model set $I(\gamma)$ is non-empty, and it is called *tautologic* if $I(\gamma) = \Omega_s$ with $s = vars(\gamma)$. This means that γ evaluates to 1 under all interpretations. Conversely, if $I(\gamma) = \emptyset$, γ is called *contradictory*.

γ	δ	$\top$	$\bot$	$\neg\gamma$	$\gamma \vee \delta$	$\gamma \wedge \delta$	$\gamma \rightarrow \delta$	$\gamma \leftrightarrow \delta$
0	0	1	0	1	0	0	1	1
0	1	1	0	1	1	0	1	0
1	0	1	0	0	1	0	0	0
1	1	1	0	0	1	1	1	1

Figure 3.2: Truth values of propositional sentences.

Defining the equivalence of propositional sentences is sometimes intricate since different sentences may represent the same information. From the above construction rule, we obtain for example two sentences $X \vee (Y \wedge Z)$ and $(X \vee Y) \wedge (X \vee Z)$ which both have exactly the same set of models. These sentences should be considered as equivalent. On the other hand, it is also preferable that X and $X \wedge (Y \vee \neg Y)$, which at first glance seem different as they contain different propositions, are equivalent. These considerations will subsequently take center stage.

Valuation Algebras on Model Level

With the simplifying concession that sentences over different propositions are regarded as different information pieces, we can define two sentences as equivalent, if they adopt the same set of models,

$$\gamma \equiv \delta \text{ if, and only if, } vars(\gamma) = vars(\delta) \text{ and } I(\gamma) = I(\delta). \tag{3.43}$$

This leads to a decomposition of $\mathcal{L}$ into equivalence classes $[\gamma]$ of sentences with identical model sets. These equivalence classes are considered as valuations. Since every model set corresponds to an equivalence class of sentences and vice versa, we reencounter the algebra of relations discussed in Section 3.3. This view of propositional logic has been worked out in (Langel, 2004).

This first approach to deriving a valuation algebra suffers from the semantical defect that only sentences with identical propositions can be considered as equivalent. To meet these concerns, the following relation considers sentences as equivalent if their vacuously extended model sets are equal,

$$\gamma \equiv \delta \quad \text{if, and only if,} \quad I(\gamma)^{\uparrow r} = I(\delta)^{\uparrow r}. \tag{3.44}$$

Then, we may identify every equivalence class $[\gamma]$ with a model set $I([\gamma]) \subseteq \Omega_r$ which leads to a so-called *domain-free valuation algebra* discussed in (Kohlas, 2003). An alternative approach that remains within the limits of labeled valuation algebras developed in Chapter 2 has been proposed by (Wilson & Mengin, 1999) and identifies every equivalence class $[\gamma]$ with the smallest model set on which all sentences $\delta \in [\gamma]$ agree. Hence, we define

$$d([\gamma]) \quad = \quad \bigcap \{vars(\delta) : \delta \in [\gamma]\} \tag{3.45}$$

and represent $[\gamma]$ by the corresponding model set that refers to variables in $d([\gamma])$. Again, we meet the valuation algebra of relations discussed in Section 3.3.

The model view furthermore allows to define an entailment relation between propositional sentences. We say that γ *entails* another sentence δ if every model of γ is also a model of δ,

$$\gamma \models \delta \quad \text{if, and only if,} \quad I(\gamma)^{\uparrow r} \subseteq I(\delta)^{\uparrow r}. \tag{3.46}$$

In this case, δ is also called a *logical consequence* of γ.

Valuation Algebras on Language Level

For computational reasons, it is often more convenient to abandon the calculus with model sets and to treat problems directly on the language level. To do so, a standardized shape of propositional sentences is needed – here, we elect the *conjunctive normal form* although other normal forms are possible as well. A *positive literal* X is a proposition out of r and accordingly we name its negation $\neg X$ a *negative literal*. A *clause* φ is a disjunction of either positive or negative literals $\varphi = l_1 \vee \ldots \vee l_m$. Such clauses are called *proper* if every proposition appears at most once. By convention we write $\bot$ for the empty clause. Then, a sentence is said to be in conjunctive normal form if it is written as conjunction of proper clauses φ_i, formally $f = \varphi_1 \wedge \ldots \wedge \varphi_n$. It is known that every sentence can be transformed into a conjunctive normal form (Chang & Lee, 1973) and it is common to represent them as *sets of clauses* $\Sigma = \{\varphi_1, \ldots, \varphi_n\}$ with $vars(\Sigma) = vars(\varphi_1) \cup \ldots \cup vars(\varphi_n)$.

Sets of clauses are subject to the same problem as usual sentences concerning their equivalence. (Kohlas *et al.*, 1999) distinguish clause sets as a matter of principle if they contain different propositions. Then, sets of clauses Σ are directly taken as valuations with domain $d(\Sigma) = vars(\Sigma)$. Combination of two clause sets Σ_1 and Σ_2 reduces to simple set union

$$\Sigma_1 \otimes \Sigma_2 \quad = \quad \mu(\Sigma_1 \cup \Sigma_2), \tag{3.47}$$

where μ denotes the subsumption operator that eliminates non-minimal clauses. More concretely, a clause $\varphi \in \Sigma$ is subsumed by another clause $\varphi' \in \Sigma$ if every literal in φ' is also contained in φ.

For the definition of marginalization, it is generally more suitable to use variable elimination when dealing with propositional logic. For this purpose, we first remark that every clause set Σ can be decomposed into disjoint subsets with respect to a proposition $X \in d(\Sigma)$ since it contains only proper clauses:

$$\begin{aligned}
\Sigma_X &= \{\varphi \in \Sigma : \varphi \text{ contains } X \text{ as positive literal}\},\\
\Sigma_{\overline{X}} &= \{\varphi \in \Sigma : \varphi \text{ contains } X \text{ as negative literal}\},\\
\Sigma_{\dot{X}} &= \{\varphi \in \Sigma : \varphi \text{ does not contain } X \text{ at all}\}.
\end{aligned}$$

Then, the elimination of a variable $X \in d(\Sigma)$ is defined by

$$\Sigma^{-X} = \mu(\Sigma_{\dot{X}} \cup R_X(\Sigma)), \tag{3.48}$$

where

$$R_X(\Sigma) = \{\vartheta_1 \vee \vartheta_2 : X \vee \vartheta_1 \in \Sigma_X \text{ and } \neg X \vee \vartheta_2 \in \Sigma_{\overline{X}}\}. \tag{3.49}$$

Theorem 3.8. *Clause sets satisfy the axioms of a valuation algebra.*

This theorem is proved in (Haenni *et al.*, 2000). Here, we restrict ourselves to finding the properties of this valuation algebra:

- From Equation (3.47) we directly conclude that the valuation algebra of clause sets is idempotent.
- Because clause sets are interpreted conjunctively, neutral elements are sets where every clause is tautologic. Further, a clause is tautologic exactly if at least one proposition occurs as a positive and negative literal. This in turn cannot be a proper clause and is therefore syntactically forbidden. Thus, we conclude that only the sub-semigroup with empty domain has a (syntactically valid) neutral element and therefore, the valuation algebra of clause sets has no neutral elements with respect to Section 2.2.
- The situation for null elements is very similar. Here, possible candidates are contradictory clause sets and this is the case if, and only if, at least two clauses are mutually contradictory. However, because subsummation is performed in every combination, it is generally impossible to identify a contradictory clause set z_s such that $\mu(\Sigma \cup z_s) = z_s$ holds for all Σ with $d(\Sigma) = s$. Ergo, null elements do not exist.

Finally, we point out that we can also define a valuation algebra for clause sets that insists on the semantically more proper equivalence of sentences (or clause sets) given in Equation (3.44). For this purpose, we introduce the notion of prime implicates. A proper clause φ is called *implicate* of a sentence $\gamma \in \mathcal{L}$, if $\gamma \models \varphi$. An implicate φ of γ is then a *prime implicate* if no proper sub-clause of φ is also an implicate of γ. In other words, prime implicates of some sentence γ are the logically strongest consequences of γ. The set of all prime implicates of γ defines a conjunctive normal form denoted by $\Phi(\gamma)$. Observe that $\gamma \equiv \Phi(\gamma)$ always holds. Further, we also write $\Phi(\Sigma)$ if γ is itself in conjunctive normal form with corresponding clause set Σ. The task of computing prime implicates is discussed in (Haenni *et al.*, 2000).

The importance of prime implicates becomes apparent considering that equivalent sentences adopt the same prime implicates. We have $\gamma \equiv \delta$ if, and only if, $\Phi(\gamma) = \Phi(\delta)$ and therefore conclude that

$$\Phi(\gamma) = \Phi(\delta) \Leftrightarrow \gamma \equiv \delta \Leftrightarrow I(\gamma)^{\uparrow r} = I(\delta)^{\uparrow r}. \tag{3.50}$$

Thus, every equivalence class is represented by a set of prime implicates. These sets are the valuations to which combination and marginalization for clause sets directly

apply.

We close the discussion of propositional logic with an example that illustrates combination and marginalization on language level for arbitrary clause sets:

Example 3.8. *Consider propositional variables* $r = \{U, V, W, X, Y, Z\}$ *and two clause sets* Σ_1 *and* Σ_2 *defined as*

$$\Sigma_1 = \{X \vee Y, X \vee \neg Y \vee Z, Z \vee \neg W\} \quad \textit{and} \quad \Sigma_2 = \{U \vee \neg W \vee Z, \neg X \vee W, X \vee V\}.$$

Combining Σ_1 *and* Σ_2 *gives*

$$\begin{aligned} \Sigma_1 \otimes \Sigma_2 &= \mu\{X \vee Y, X \vee \neg Y \vee Z, Z \vee \neg W, U \vee \neg W \vee Z, \neg X \vee W, X \vee V\} \\ &= \{X \vee Y, X \vee \neg Y \vee Z, Z \vee \neg W, \neg X \vee W, X \vee V\}. \end{aligned}$$

To eliminate variable X*, the clause set is partitioned as follows:*

$$\begin{aligned} (\Sigma_1 \otimes \Sigma_2)_X &= \{X \vee Y, X \vee \neg Y \vee Z, X \vee V\}, \\ (\Sigma_1 \otimes \Sigma_2)_{\overline{X}} &= \{\neg X \vee W\}, \\ (\Sigma_1 \otimes \Sigma_2)_{\dot{X}} &= \{Z \vee \neg W\}. \end{aligned}$$

Then we obtain

$$\begin{aligned} (\Sigma_1 \otimes \Sigma_2)^{-X} &= \mu(\{Z \vee \neg W\} \cup \{Y \vee W, \neg Y \vee Z \vee W, V \vee W\}) \\ &= \{Z \vee \neg W, Y \vee W, \neg Y \vee Z \vee W, V \vee W\}. \end{aligned}$$

⊖

3.9 Conclusion

To conclude this catalogue of valuation algebra instances, we compare in the table below all formalisms described in this chapter with the properties for which they have been analysed. A checkmark always indicates that the property is algebraically present within the valuation algebra, even though it may be trivial. Scaling within an idempotent valuation algebra is typically such a case as we may conclude from Equation (2.54). On the other hand, ∘ means that the property does not hold due to algebraic reasons. As mentioned already in the introduction of this chapter, we will extend this collection of valuation algebra instances in Chapters 6 and 7. Nevertheless, there are numerous known formalism which also fit into the valuation algebra framework but which remain undiscussed in this thesis. Some additional examples such as Spohn potentials, Gaussian hints, linear equations and inequalities are described in (Kohlas, 2003). Predicate logic in the perspective of valuation algebras is handled in (Langel & Kohlas, 2007).

	Neutral Elements	Stability	Null Elements	Separativity	Regularity	Idempotency	Scalability
Indicator Functions	√	√	√	√	√	√	√
Relations	√	√	√	√	√	√	√
Probability Potentials	√	○	√	√	√	○	√
Belief Functions	√	√	√	√	○	○	√
Distance Functions	√	√	○	√	√	√	√
Densities	○	○	√	√	○	○	√
Gaussian Potentials	○	○	○	√	○	○	√
Clause Sets	○	○	○	√	√	√	√

Figure 3.3: Instance property map.

4
Local Computation

With the valuation algebra framework introduced in the first chapter, we have a language that unifies a very large number of formalisms for knowledge representation. We mentioned in the introduction of this thesis that performing inference or deduction is the central computational problem of any knowledge representation system. Hence, we are next going to express this inference task in the abstract language of valuation algebras, which leads to a generic problem description that is actually independent of the underlying formalism. This computational task is called *projection problem* and is the starting point of this chapter. Motivated by the intractability of a direct problem solution, we dip into the derivation of local computation algorithms that solve projection problems efficiently on a purely generic level. From the classical literature, we may identify two families of local computation schemes. On the one hand, the *fusion* (Shenoy, 1992b) and *bucket elimination algorithms* (Dechter, 1999) start directly from a given set of valuations called *knowledgebase*. They remove a selection of factors, combine them, perform a marginalization and reinsert the result into the knowledgebase. This procedure is repeated until the knowledgebase consists of a single valuation that expresses the result of the projection problem. However, the major drawback of this approach is that only *single-query projection problems* can be answered. Therefore, a second family of local computation algorithms was proposed that first distribute the knowledgebase factors on a graphical structure called *join tree* to organize afterwards the computations in a message passing fashion. So, we are allowed to compute multiple projection problems over the same knowledgebase at once by establishing a caching policy to reuse intermediate results (messages). There are in fact different approaches to define this caching policy, which gives raise to the four major architectures of local computation, namely the Shenoy-Shafer (Shenoy & Shafer, 1990), Lauritzen-Spiegelhalter (Lauritzen & Spiegelhalter, 1988), HUGIN (Jensen *et al.*, 1990) and Idempotent Architecture (Kohlas, 2003). All these architectures will be studied in detail in this chapter, and we are also going to present some small modifications that directly yield scaled results.

The most important requirement for this second family of local computation algorithms is that the domains of the involved valuations form a so-called *join tree factorization*. This is a very restrictive condition, but it can always be guaranteed

thanks to the presence of neutral elements. However, we pointed out a more general definition of valuation algebras in Chapter 2 that does without such neutral elements. Instead, we adjoined artificially a unique identity element and justified this measure by its future use for computational purposes. In this chapter, it will at long last be shown that the identity element allows to circumvented the claim for a join tree factorization. On the one hand, this means that local computation can now be applied to formalisms without neutral elements, or to formalisms where neutral elements are not finitely representable. On the other hand, we will also see that local computation even becomes more efficient without the explicit use of neutral elements.

Since the solution of projection problems takes central stage in this chapter, we reasonably start with a formal introduction of this concept. Although all studies will be based on the abstract valuation algebra language, we nevertheless explore the semantics of the projection problem for a small selection of instances in Section 4.2. This includes Bayesian networks, shortest path routing and satisfiability tests. After some complexity considerations, we establish in Section 4.3 the basic prerequisites for local computation in the shape of the covering join tree that substitutes the classical join tree factorization. Then, the four major local computation architectures are introduced in Sections 4.4 to 4.7 and their variations to compute scaled results in Section 4.8. Finally, we fathom the complexity of local computation in general and close this chapter by delineating a few possibilities for further improvements.

4.1 Knowledgebases

Consider a valuation algebra (Φ, D) with identity element e and a set of valuations $\{\phi_1, \ldots, \phi_n\} \subseteq \Phi$. This set stands for the available knowledge about some topic which constitutes the starting point of any deduction process. Accordingly, we refer to such a set as *knowledgebase*. It is often very helpful to represent knowledgebases graphically, since this uncovers a lot of hidden structure. There are numerous possibilities for this task, but most of them are somehow related to a particular instance. Here, we restrict ourselves to an informal introduction of *valuation networks*, *hypergraphs* and *primal graphs*, because they can all be used to model generic knowledgebases.

4.1.1 Valuation Networks

A valuation network (Shenoy, 1992b) is a graphical structure where variables are represented by circular nodes and valuations by rectangular nodes. Further, each valuation is connected with all variables that are contained in its domain. Valuation networks highlight the structure of a factorization and are therefore also called *factor graphs* in the literature. Figure 4.1 shows a valuation network for the knowledgebase $\{\phi_1, \phi_2, \phi_3\}$ with $d(\phi_1) = \{A, C\}$, $d(\phi_2) = \{B, C, D\}$ and $d(\phi_3) = \{B, D\}$.

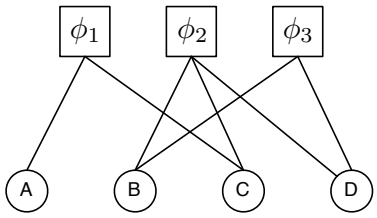


Figure 4.1: A valuation network for the knowledgebase $\{\phi_1, \phi_2, \phi_3\}$ with $d(\phi_1) = \{A, C\}$, $d(\phi_2) = \{B, C, D\}$ and $d(\phi_3) = \{B, D\}$.

4.1.2 Hypergraphs & Primal Graphs

Hypergraphs are essentially sets of nonempty subsets which in turn contain elements from a finite set. Obviously, a hypergraph can be extracted from a knowledgebase if we build the set of all nonempty factor domains. Doing so, the hypergraph for the knowledgebase visualized in Figure 4.1 is $\{\{A, C\}, \{B, C, D\}, \{B, D\}\}$. Representing hypergraphs graphically is challenging. A common way is shown by the left hand drawing in Figure 4.2, but the representation by primal graphs is more convenient. Here, variables correspond to the nodes of a primal graph and two nodes are linked together if, and only if, the two variables appear in the same set of the hypergraph. This is shown in the right hand drawing of Figure 4.2.

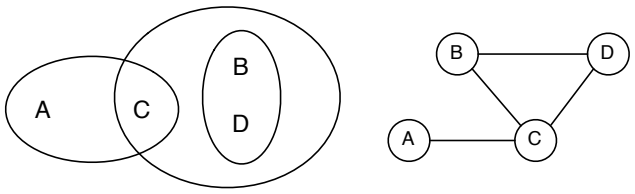


Figure 4.2: A hypergraph and a corresponding primal graph for the knowledgebase $\{\phi_1, \phi_2, \phi_3\}$ with $d(\phi_1) = \{A, C\}$, $d(\phi_2) = \{B, C, D\}$ and $d(\phi_3) = \{B, D\}$.

4.2 The Projection Problem

A knowledgebase is an important component of a projection problem. As hinted at in the introduction of this chapter, projection problems consist in marginalizing some *joint valuation* ϕ, obtained by the consecutive combination of all factors within a given knowledgebase, onto a set of *queries* which represent the sets of questions we are interested in. This is the statement of the following definition.

Definition 4.1. *The task of computing*

$$\phi^{\downarrow x_i} = (\phi_1 \otimes \cdots \otimes \phi_n)^{\downarrow x_i} \tag{4.1}$$

for a given knowledgebase $\{\phi_1, \ldots, \phi_n\} \subseteq \Phi$ *and queries* $x = \{x_1, \ldots, x_s\}$ *where* $x_i \subseteq d(\phi_1 \otimes \ldots \otimes \phi_n)$ *is called* projection problem *or* inference problem*.*

Historically, the joint valuation ϕ is sometimes also called *objective function*. Projection problems with $|x| = 1$ are usually called *single-query projection problems*, and we speak about *multi-query projection problems* if $|x| > 1$. Further, it is implicitly assumed throughout this thesis that the queries of any projection problem are well-defined, i.e. can be computed. This is not necessarily the case in a valuation algebra with partial marginalization where legal marginals are given by the domain operator. We therefore assume that $x \subseteq \mathcal{M}(\phi)$ if marginalization is only partially defined. In order to give an idea of how different instances give rise to projection problems, we touch upon some typical and popular fields of application. This listing will later be extended in Chapter 6 and 8.

4.2.1 Bayesian Networks

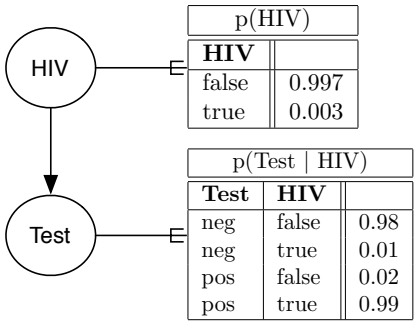
A *Bayesian network*, as for instance in (Pearl, 1988), is a graphical representation of a *joint probability distribution* over a set of variables $\{X_1, \ldots, X_n\}$. The network itself is a *directed acyclic graph* that reflects the conditional independencies among variables, which in turn are associated with a node of the network. Additionally, each node contains a *conditional probability table* that quantifies the influence between variables. These tables are modeled using the formalism of probability potentials. Then, the joint probability distribution p of a Bayesian network is given by

$$p(X_1, \ldots, X_n) = \prod_{i=1}^{n} p(X_i|pa(X_i)) \tag{4.2}$$

where $pa(X_i)$ denotes the parents of X_i in the network. To clarify how Bayesian networks give rise to projection problems, we consider the following example.

The ELISA test was the first screening test commonly employed for HIV. In order to model this test using a Bayesian network, we assume two binary variables: *HIV* with states *true* and *false* to denote whether a test person is infected, and *Test* with values *positive* and *negative* for the test result. The medical specification of ELISA declares a *test sensitivity* (probability of a positive test among patients with disease) of 0.99 and a *test specificity* (probability of a negative test among patients without disease) of 0.98. Finally, we know that approximately 0.3% of Swiss people are HIV infected. This is modeled by the Bayesian network in Figure 4.3. Next, assume that we are interested in the reliability of a positive test result. Applying the Bayesian rule, we get

$$p(true|positive) = \frac{p(positive|true) \cdot p(true)}{p(positive)} = \frac{p(positive \wedge true)}{p(positive)}. \tag{4.3}$$



p(HIV)

HIV	
false	0.997
true	0.003

p(Test | HIV)

Test	HIV	
neg	false	0.98
neg	true	0.01
pos	false	0.02
pos	true	0.99

Figure 4.3: A Bayesian network for the ELISA test example.

Consequently, the task consists in computing $p(positive \wedge true)$ and $p(positive)$, which requires the solution of a projection problem with two queries:

$$\begin{aligned} p(Test, HIV) &= \Big(p(HIV) \otimes p(Test|HIV)\Big)^{\downarrow\{Test,HIV\}} \\ &= p(HIV) \otimes p(Test|HIV) \end{aligned}$$

and

$$p(Test) = \Big(p(HIV) \otimes p(Test|HIV)\Big)^{\downarrow\{Test\}}.$$

Worryingly, the computation gives $p(true|positive) \approx 13\%$. This is related to the assumed marginal probability $p(HIV)$ as we see if the same test is applied in Botswana, where approximately 40% of the population are infected. We then obtain $p(true|positive) \approx 97\%$ in this case. The ELISA test is currently used as a bulk test since it is cheap and its evaluation is fast. In case of a positive test result, the more sensitive but also more expensive Western-Blood-Test is applied.

More generally, if a Bayesian network has to be evaluated for some variable $X \in \{X_1, \ldots, X_n\}$ and *evidence* $e \subseteq \{X_1, \ldots, X_n\} - \{X\}$, we have

$$p(X|e) = \frac{p(X, e)}{p(e)}$$

where $p(X, e)$ and $p(e)$ are obtained by solving a two-query projection problem.

4.2.2 Shortest Path Routing

Efficient routing is an important issue in *packet-switched communication networks*, and because networks are huge and dynamic, it is generally impossible to have a global and static table that contains all shortest distances between any two network hosts. Moreover, every device originally knows only some distances in its very close

neighborhood, and by mutually exchanging data, they learn about larger parts of the network topology. The distance potentials ϕ_i therefore constitute the knowledgebase of this projection problem, and their combination would lead to the global distance table, from which a final marginalization extracts the required entry. Such an application is implemented for example in (de Groote, 2006) to find shortest distances between different cities of Western Europe.

4.2.3 Satisfiability Problems

A knowledgebase can be derived from a set of propositional sentences by representing every sentence as valuation, i.e. either as model or clause set according to Section 3.8. Thus, propositional knowledgebases are interpreted conjunctively, which means that they are satisfiable exactly if every element is satisfiable. Consequently, a satisfiability test reduces to the solution of a single-query projection problem to the empty domain,

$$\phi^{\downarrow\emptyset} \quad = \quad (\phi_1 \otimes \cdots \otimes \phi_n)^{\downarrow\emptyset} .$$

If we act on model level, $\phi^{\downarrow\emptyset} = \{\diamond\}$ implies that the knowledgebase is satisfiable. Otherwise, we obtain $\phi^{\downarrow\emptyset} = \emptyset$. Alternatively, if we act on language level, $\phi^{\downarrow\emptyset} = \emptyset$ stands for a satisfiable knowledgebase and we obtain $\phi^{\downarrow\emptyset} = \{\bot\}$ if the knowledge is contradictory. Many further important tasks in propositional logic can be reduced to the solution of projection problems. *Theorem proving* is such an application. Given a *hypothesis* h and a propositional knowledgebase (clause set) Σ, testing whether $\Sigma \models h$ is equivalent to verifying whether $\Sigma \cup \{\neg h\}$ is contradictory. Another application called *consequence finding* is described in (Langel, 2004). Further interpretations for other formalisms than propositional logic will be discussed in Section 8.1.

4.2.4 Complexity Considerations

The perhaps simplest approach to solve a projection problem is to directly carry out the combinations followed by the final marginalization, but this straightforward strategy is usually beyond all question. First combining all knowledgebase factors leads to the joint valuation $\phi = \phi_1 \otimes \cdots \otimes \phi_n$ with domain $s = d(\phi_1) \cup \cdots \cup d(\phi_n)$. Let us, for example, analyse the amount of memory that is used to store ϕ and express this measure by $\omega(\phi)$. Then, the following worst case estimations for $\omega(\phi)$ with respect to an assortment of valuation algebra instances may be given.

- If ϕ is a distance potential: $\omega(\phi) \in \mathcal{O}(|s|^2)$.
- For probability potentials and connatural formalisms: $\omega(\phi) \in \mathcal{O}(2^{|s|})$.
- If ϕ is a belief function: $\omega(\phi) \in \mathcal{O}(2^{2^{|s|}})$.

These first considerations are very sketchy since they only regard the memory consumption and completely ignore the complexity of the valuation algebra operations involved in the solution of the projection problem. Nevertheless, they are sufficient to back up the following conclusions. Apart from distance potentials, the

complexity of ω increases exponentially or even super exponentially with the size of the domain of ϕ. Thus, the domain acts as the crucial point for complexity, and it may be said that projection problems are intractable unless algorithms are used which confine the domain size of all intermediate results and operations. This essentially is the promise of local computation.

Before actually starting to discuss local computation schemes, we would like to point out that the low polynomial complexity of distance functions may be a reason why this formalism has never been studied in the context of valuation algebras. In fact, the efficiency of *Dijkstra's algorithm* (Dijkstra, 1959) (or *Bellman-Ford* if negative distances need to be considered) keeps people from considering shortest path routing as projection problem. For our perception however, it is notably interesting that even such good-natured formalisms adopt the structure of a valuation algebra.

4.3 Covering Join Trees

Local computation algorithms operate on an underlying graphical structure called *covering join trees* to which this section is dedicated. Very generally, join trees are *undirected graphs* $G = (V, E)$ which are specified by a set of *vertices* or *nodes* V and a set of *edges* E. Edges are simply pairs of nodes. A sequence of consecutive edges within a graph is called a *path*. A graph G is said to be *connected* if there is a path between any two nodes in G. Finally, a *tree* is a connected, undirected graph with exactly one path between any two nodes. In other words, if we remove a single edge from a tree, the resulting graph is not connected anymore. These concepts are illustrated in Example 4.1 where nodes are identified by numbers. Without loss of generality, we subsequently adopt the convention that $V \subseteq \mathbb{N}$.

Example 4.1. *Consider the two graphs in Figure 4.4. The left-hand graph* $G_1 = (V_1, E_1)$ *is specified by* $V_1 = \{1, 2, 3, 4\}$ *with* $E_1 = \{(1, 2), (2, 3), (3, 1), (4, 4)\}$. *This graph is not connected. On the right,* $G_2 = (V_2, E_2)$ *is given by* $V_2 = \{1, 2, 3, 4, 5\}$ *and* $E_2 = \{(5, 1), (1, 2), (1, 3), (3, 4)\}$. *The only path from vertex* 1 *to vertex* 4 *in* G_2 *is* $((1, 3), (3, 4))$. *Moreover, there is exactly one path between any two nodes. Therefore,* G_2 *a tree which is clearly not the case for* G_1. ⊖

Figure 4.4: Unconnected and connected graphs.

Next, we consider trees where every node possesses a *label* out of the power set $D = \mathcal{P}(r)$ of a set of variables r.

Definition 4.2. *A* labeled tree (V, E, λ, D) *is a tree* (V, E) *together with a mapping*

$$\lambda : V \to D. \tag{4.4}$$

Of particular interest are labeled trees that satisfy the *running intersection property*. This property is sometimes also called *Markov property* or simply *join tree property*. Accordingly, such trees are named *join trees* or *Markov trees*.

Definition 4.3. *A labeled tree* (V, E, λ, D) *satisfies the* running intersection property *if for two nodes* $i, j \in V$ *and* $X \in \lambda(i) \cap \lambda(j)$, $X \in \lambda(k)$ *for all nodes* k *on the path between* i *and* j.

Example 4.2. *Figure 4.5 reconsiders the tree* G_2 *from Example 4.1 equipped with a labeling function* λ *based on* $r = \{A, B, C, D\}$. *The labels are:* $\lambda(1) = \{A, C, D\}$, $\lambda(2) = \{A, D\}$, $\lambda(3) = \{D\}$, $\lambda(4) = \{C\}$ *and* $\lambda(5) = \{A, B\}$. *This labeled tree is not a join tree since* $C \in \lambda(1) \cap \lambda(4)$ *but* $C \notin \lambda(3)$. *However, if variable* C *is added to node label* $\lambda(3)$, *the result will be a join tree as shown in Example 4.3.* ⊖

Definition 4.4. *Let* $\mathcal{T} = (V, E, \lambda, D)$ *be a join tree.*

- $\mathcal{T}$ *is said to* cover *the domains* $s_1, \ldots, s_m \in D$ *if there is for every* s_i *a node* $j \in V$ *with* $s_i \subseteq \lambda(j)$.
- $\mathcal{T}$ *is called* covering join tree *for* $\phi_1 \otimes \cdots \otimes \phi_m$ *if it covers the domains* $s_i = d(\phi_i) \in D$ *for* $1 \leq i \leq m$.

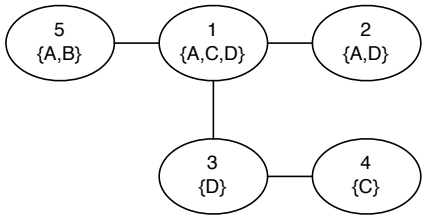


Figure 4.5: This labeled tree is not a join tree since variable C is missing in node 3.

In a covering join tree, every factor of the factorization ϕ is covered by at least one node. Thus, it is possible to assign every factor ϕ_i to a node j which covers its domain.

Definition 4.5. *Let* $\mathcal{T} = (V, E, \lambda, D)$ *be a covering join tree for the factorization* $\phi = \phi_1 \otimes \cdots \otimes \phi_m$. *A function*

$$a : \{1, \ldots, m\} \to V$$

is called an assignment mapping *for* ϕ *regarding* V *if for every* $i \in \{1, 2, \ldots, m\}$ *we have* $d(\phi_i) \subseteq \lambda(a(i))$.

This shall again be illustrated by an example.

Example 4.3. *We consider a factorization $\phi_1 \otimes \phi_2 \otimes \phi_3 \otimes \phi_4$ with domains $d(\phi_1) = \{C, D\}$, $d(\phi_2) = \{B\}$, $d(\phi_3) = \{D\}$, $d(\phi_4) = \{A, D\}$. Figure 4.6 shows a possible assignment of these factors to the nodes of the join tree mentioned in Example 4.2. We have $a(1) = 3$, $a(2) = 5$, $a(3) = a(4) = 2$.* ⊖

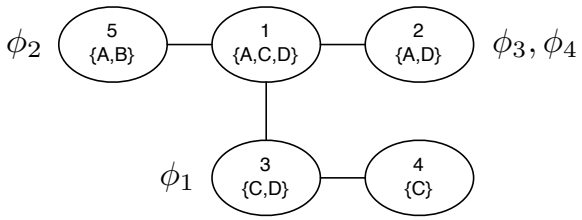


Figure 4.6: A join tree with factor assignment.

Surjectivity of an assignment mapping can always be achieved by adding a sufficient number of identity elements e to the factorization ϕ and assigning such an element to all nodes that do not yet possess a factor. This is the statement of the following lemma.

Lemma 4.6. *Let a' be an assignment mapping regarding V which is not surjective for a factorization ϕ with m factors. Then, there exists a* surjective *assignment mapping $a : \{1, \ldots, m+k\} \to V$ for a factorization of the form*

$$\phi \quad = \quad \phi_1 \otimes \cdots \otimes \phi_m \otimes e_{m+1} \otimes \cdots \otimes e_{m+k}$$

with $e_i = e$ for all i, $m+1 \leq i \leq m+k$.

Proof. Let V'' be the subset of nodes in V given by

$$V'' = \bigcup_{i \in V : \nexists j \text{ with } a'(j) = i} i$$

of cardinality $k = |V''|$ and $a'' : \{m+1, \ldots, m+k\} \to V''$ be a *surjective* assignment mapping for $e = e_{m+1} \otimes \cdots \otimes e_{m+k}$ with $e_i = e$ for all i, $m+1 \leq i \leq m+k$. Such a surjective mapping a'' does clearly exist, since $d(e) = \emptyset$ and the number of the factors equal the cardinality of the nodes in V''. The mappings a' and a'' then lead to a surjective assignment mapping $a : \{1, \ldots, m+k\} \to V$ for the factorization

$$\phi = \phi_1 \otimes \cdots \otimes \phi_m \otimes e_{m+1} \otimes \cdots \otimes e_{m+k}$$

by

$$a(j) = \begin{cases} a'(j) & \text{if } j \in \{1, \ldots, m\}, \\ a''(j) & \text{otherwise.} \end{cases}$$

□

It is worth noting that such assignment mappings are usually not injective. However, a surjective assignment of factors to join tree nodes leads naturally to a new factorization whose factors are defined as follows:

Definition 4.7. *Let a be a surjective assignment mapping for a factorization ϕ regarding the nodes V of a covering join tree (V, E, λ, D). The* factor assigned to node $i \in V$ *by a is defined by*

$$\psi_i \quad = \quad e \otimes \bigotimes_{j:a(j)=i} \phi_j. \tag{4.5}$$

The factors ψ_i are often called *join tree factors* and each of them corresponds either to a single factor of the original factorization, a combination of some of them, or to the identity element if the factor set of the combination in Equation (4.5) is empty. Since every valuation in the original factorization is assigned to exactly one node of V, it holds that

$$\phi \;=\; \psi_1 \otimes \cdots \otimes \psi_n \;=\; \bigotimes_{i \in V} \psi_i.$$

Definition 4.8. *Let a be a surjective assignment mapping for a factorization ϕ regarding the nodes of a covering join tree (V, E, λ, D). The* domain *of a node $i \in V$ is defined by*

$$\omega_i \quad = \quad d(\psi_i)$$

if ψ_i denotes the valuation assigned to the node i by a.

It is important to stress the difference between the label $\lambda(i)$ and the domain ω_i of a node $i \in V$. The domain refers to the domain of the factor that is actually kept by the current node. The label on the other hand represents the largest possible domain of a factor that would fit into this node. A direct consequence is that the first is included in the latter, $\omega_i \subseteq \lambda(i)$.

Example 4.4. *Regarding Example 4.3, we detect $\psi_1 = e$, $\psi_2 = \phi_3 \otimes \phi_4$, $\psi_3 = \phi_1$, $\psi_4 = e$, $\psi_5 = \phi_2$ with $\omega_1 = \emptyset$, $\omega_2 = \{A, D\}$, $\omega_3 = \{C, D\}$, $\omega_4 = \emptyset$, $\omega_5 = \{B\}$.* ⊖

Covering join trees constitute the graphical structure upon which local computation algorithms run. For that purpose, a covering join tree needs to be found that goes with the projection problem one currently wants to solve.

Definition 4.9. *A join tree $\mathcal{T} = (V, E, \lambda, D)$ is a* covering join tree for the projection problem

$$(\phi_1 \otimes \cdots \otimes \phi_n)^{\downarrow x_i}$$

with $x_i \in \{x_1, \ldots, x_s\}$ if the following conditions are satisfied:

- *$\mathcal{T}$ is a covering join tree for the factorization $\phi_1 \otimes \cdots \otimes \phi_n$.*
- *$\mathcal{T}$ covers the queries $\{x_1, \ldots, x_s\}$.*
- *D corresponds to the power set $\mathcal{P}(d(\phi_1 \otimes \cdots \otimes \phi_n))$.*

Note that Condition 3 demands a covering join tree without *free variables*. This means that each variable in the node labels must also occur somewhere in the domains of the projection problem factors.

4.3.1 Local Computation Base

A common characteristic of all local computation algorithms is that they are regarded as message passing schemes. Messages are sent between the nodes of the join tree with the nodes acting as virtual processors (Shenoy & Shafer, 1990). They process incoming messages, compute new messages and send them to other neighboring nodes of the join tree. The underlying algorithms schedule these messages according to a numbering of the nodes which is introduced beforehand. First, we fix an arbitrary node as root node which gets the number $m = |V|$. For a single-query projection problem, we always choose the root node in such a way that it covers the query. Then, by directing all edges towards the root node m, it is possible to determine a numbering in such a way that if j is a node on the path from node i to m, then $j > i$. Formally, let (V, E, λ, D) be a join tree with $|V| = m$ nodes and determine a permutation π of the elements in V such that

- $\pi(k) = m$, if k is the root node;
- $\pi(j) > \pi(i)$ for every node $j \in V$ lying on the path between i and m.

The result is a renumbered join tree (V, E', λ, D) with edges $E' = \{(\pi(i), \pi(j)) : (i, j) \in E\}$. (V, E', λ, D) is usually called a *directed join tree* towards the root node m. Note also that such numberings are not unique. It is furthermore convenient to introduce the notions of *parents*, *child*, *neighbors* and *leaves* with respect to this node numbering:

Definition 4.10. *Let (V, E, λ, D) be a directed join tree towards root node m.*

- *The* parents $pa(i)$ *of a node i are defined by the set*

$$pa(i) \;=\; \{j : j < i \text{ and } (i, j) \in E\}.$$

- *Nodes without parents are called* leaves.
- *The* child $ch(i)$ *of a node $i < m$ is the unique node j with $j > i$ and $(i, j) \in E$.*
- *The* neighbors $ne(i)$ *of a node i form the set of nodes*

$$ne(i) \;=\; \{j : (i, j) \in E\} \;=\; pa(i) \cup \{ch(i)\}.$$

The definition of a tree implies that whenever one of its edges $(i, ch(i))$ is removed, the tree splits into two parts where the one that contains the node i is called *sub-tree rooted to node i*, abbreviated by $\mathcal{T}_i$. Clearly, the running intersection property remains satisfied if join trees are cut in two.

Example 4.5. *Continuing with Example 4.3, the node with label* $\{A, B\}$ *is chosen as root node and all edges are directed towards it. A possible node numbering is:* $\pi(5) = 5$, $\pi(4) = 1$, $\pi(3) = 2$, $\pi(2) = 3$, $\pi(1) = 4$ *as shown in Figure 4.7. Further, we have for node* 4 *in particular:* $ch(4) = 5$, $pa(4) = \{2, 3\}$ *and* $ne(4) = \{2, 3, 5\}$. *The leaves of this join tree are* $\{1, 3\}$. ⊖

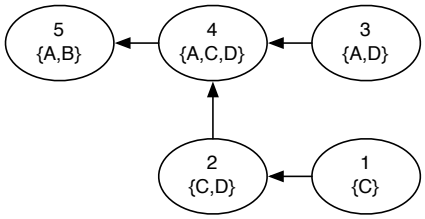


Figure 4.7: A numbered and directed join tree.

Finally, all required components for solving a projection problem using local computation techniques are resumed in the following definition. Remember that every assignment mapping with respect to a given projection problem and a join tree can be made surjective, which leads to the required join tree factorization. This step is implicitly presumed in this definition.

Definition 4.11. *A* local computation base *for a projection problem*

$$(\psi_1 \otimes \cdots \otimes \psi_m)^{\downarrow x_i}$$

with $x_i \in \{x_1, \ldots, x_s\}$ *is a quintuple* (V, E, λ, D, a), *where*

- $\mathcal{T} = (V, E, \lambda, D)$ *is a covering join tree for the given projection problem;*
- $\mathcal{T}$ *is directed towards the root node* $m = |V|$*;*
- a *is a surjective assignment mapping regarding* $\mathcal{T}$ *and* $\psi_1 \otimes \cdots \otimes \psi_m$.

We also want to point out that in this thesis the knowledgebase of the projection problem contained in a local computation base will never be subject to any modification. This problem of *updating* has been addressed in (Schneuwly, 2007).

4.3.2 The Benefit of Covering Join Trees

A local computation base can be regarded as the common setup for all local computation procedures introduced in this thesis. The requirement of a covering join tree is far less restrictive compared with traditional literature where it is generally claimed that the hypergraph (see Section 4.1.2) of the knowledgebase is acyclic (Shenoy & Shafer, 1990). Such hypergraphs are called *hypertrees*. In other words, this enforces

that the domain of every factor directly corresponds to a node label. If the valuation algebra contains neutral elements, this can always be achieved by extending ψ_i vacuously to coincide with the corresponding node label

$$d(\psi_i \otimes e_{\lambda(i)}) \quad = \quad \lambda(i). \tag{4.6}$$

Loosely spoken, the content is blown up to fill the node completely. However, there are several drawbacks of this approach.

- First, it cannot be applied to every valuation algebra since neutral elements do not necessarily exist. See for example the valuation algebra of density functions introduced in Section 3.7.
- But even if neutral elements exist, they are perhaps not finitely representable. Here we refer to the valuation algebra of relations discussed in Section 3.3 where neutral elements correspond to infinite tables.
- Finally, we will later identify the size of the node labels as the crucial factor for complexity. When using a covering join tree, this complexity reduces to an upper bound which makes local computation more efficient.

All these facts speak for the use of covering join trees for the introduction of local computation methods.

4.4 Shenoy-Shafer Architecture

This section introduces the first and most general local computation scheme called *Shenoy-Shafer architecture* which has been described for the first time in (Shenoy & Shafer, 1990). Its generality is due to the fact that the architecture contents itself with the structure of a valuation algebra and does not require further properties. We follow a two-step procedure by first presenting an algorithm for the solution of single-query projection problems called *collect algorithm*. Afterwards, it will be shown that this algorithm can be extended for answering multiple queries at once, which constitutes the Shenoy-Shafer architecture.

4.4.1 Collect Algorithm

The starting point for the description of the collect algorithm is a local computation base for a single-query projection problem. The corresponding covering join tree keeps a factor ψ_j on every node j, distributed by the surjective assignment mapping a. Hence, ψ_j represents the initial content of node j. The collect algorithm can then be described by the following set of rules:

R1: Each node sends a message to its child when it has received all messages from its parents. This implies that leaves can send their messages right away.

R2: When a node is ready to send, it computes the message by marginalizing its current content to the intersection of its domain and its child's node label.

R3: When a node receives a message, it updates its current content by combining it with the incoming message.

According to these rules, each node waits until it has received a message from all of its parents. Incoming messages are combined with the current node content and then the node computes a message itself and sends it in turn to its child node. This procedure is repeated up to the root node. It follows from this first sketch that the content of each node changes during the algorithm's run. To incorporate this dynamic behavior, the following notation is introduced.

- $\psi_j^{(1)} = \psi_j$ is the initial content of node j.
- $\psi_j^{(i)}$ is the content of node j before step i of the collect algorithm.

A similar notation is adopted to refer to the domain of a node:

- $\omega_j^{(1)} = \omega_j = d(\psi_j)$ is the initial domain of node j.
- $\omega_j^{(i)} = d(\psi_j^{(i)})$ is the domain of node j before step i of the collect algorithm.

The very particular way of numbering the nodes of the directed join tree implies that at step i, node i can send a message to its child. This allows the following specification of the collect algorithm.

- At step i, node i computes the message

$$\mu_{i \to ch(i)} \quad = \quad \psi_i^{(i)\downarrow \omega_i^{(i)} \cap \lambda(ch(i))}. \tag{4.7}$$

  This message is sent to the child node $ch(i)$ with node label $\lambda(ch(i))$.

- The receiving node $ch(i)$ combines the message with its node content:

$$\psi_{ch(i)}^{(i+1)} \quad = \quad \psi_{ch(i)}^{(i)} \otimes \mu_{i \to ch(i)}. \tag{4.8}$$

  Its node domain changes to:

$$\omega_{ch(i)}^{(i+1)} \;=\; d(\psi_{ch(i)}^{(i+1)}) \;=\; \omega_{ch(i)}^{(i)} \cup \left(\omega_i^{(i)} \cap \lambda(ch(i))\right). \tag{4.9}$$

  The content of all other nodes does not change at step i,

$$\psi_j^{(i+1)} \quad = \quad \psi_j^{(i)} \tag{4.10}$$

  for all $j \neq ch(i)$, and the same holds for the node domains: $\omega_j^{(j+1)} = \omega_j^{(i)}$.

The justification of the collect algorithm is formulated by the following theorem. Remember that in case of single-query projection problems, the root node has been chosen in such a way that it covers the query.

Theorem 4.12. *At the end of the collect algorithm, the root node m contains the marginal of ϕ relative to $\lambda(m)$,*

$$\psi_m^{(m)} \quad = \quad \phi^{\downarrow \lambda(m)}. \tag{4.11}$$

The following lemma will be useful to prove this important theorem.

Lemma 4.13. *Define*

$$y_i \;=\; \bigcup_{j=i}^{m} \omega_j^{(i)} \qquad\qquad i = 1, \ldots, m. \tag{4.12}$$

Then, for $i = 1, \ldots, m-1$,

$$\left(\bigotimes_{j=i}^{m} \psi_j^{(i)} \right)^{\downarrow y_{i+1}} \;=\; \bigotimes_{j=i+1}^{m} \psi_j^{(i+1)} \;=\; \phi^{\downarrow y_{i+1}}. \tag{4.13}$$

Proof. First, it has to be ensured that $y_{i+1} \subseteq y_i$ holds in order to guarantee that the marginalization in Equation (4.13) is well defined:

$$\begin{aligned} y_i &= \omega_i^{(i)} \cup \omega_{ch(i)}^{(i)} \cup \bigcup_{j=i+1, j\neq ch(i)}^{m} \omega_j^{(i)} \\ y_{i+1} &= \omega_{ch(i)}^{(i+1)} \cup \bigcup_{j=i+1, j\neq ch(i)}^{m} \omega_j^{(i+1)}. \end{aligned}$$

From (4.8) it follows that

$$\omega_{ch(i)}^{(i+1)} \;=\; \omega_{ch(i)}^{(i)} \cup (\omega_i^{(i)} \cap \lambda(ch(i))) \;\subseteq\; \omega_{ch(i)}^{(i)} \cup \omega_i^{(i)} \tag{4.14}$$

and since $\omega_j^{(i+1)} = \omega_j^{(i)}$ for all $j \neq ch(i)$, $y_{i+1} \subseteq y_i$ must hold.

Next, the following property will be proved:

$$\omega_i^{(i)} \cap y_{i+1} \;=\; \omega_i^{(i)} \cap \lambda(ch(i)). \tag{4.15}$$

Assume first that $X \in \omega_i^{(i)} \cap \lambda(ch(i))$. Then, Equation (4.14) implies that $X \in \omega_{ch(i)}^{(i+1)}$ and by the definition of y_{i+1}, $X \in y_{i+1}$. Hence $X \in \omega_i^{(i)} \cap y_{i+1}$. On the other hand, assume that $X \in \omega_i^{(i)} \cap y_{i+1}$. Then, by the running intersection property and the definition of y_{i+1}, $X \in \lambda(ch(i))$ and therefore $X \in \omega_i^{(i)} \cap \lambda(ch(i))$.

An immediate consequence of Equation (4.9) and (4.10) is that

$$\begin{aligned} y_{i+1} &= \omega_{ch(i)}^{(i+1)} \cup \bigcup_{j=i+1, j\neq ch(i)} \omega_j^{(i+1)} \\ &\supseteq \omega_{ch(i)}^{(i)} \cup \bigcup_{j=i+1, j\neq ch(i)} \omega_j^{(i)}. \end{aligned}$$

Therefore, by application of the combination axiom and together with Property (4.15) we obtain:

$$\begin{aligned}
\left(\bigotimes_{j=i}^{m} \psi_j^{(i)}\right)^{\downarrow y_{i+1}} &= \left(\psi_i^{(i)} \otimes \left(\psi_{ch(i)}^{(i)} \otimes \bigotimes_{j=i+1, j\neq ch(i)}^{m} \psi_j^{(i)}\right)\right)^{\downarrow y_{i+1}} \\
&= \psi_i^{(i)\downarrow \omega_i^{(i)} \cap y_{i+1}} \otimes \psi_{ch(i)}^{(i)} \otimes \bigotimes_{j=i+1, j\neq ch(i)}^{m} \psi_j^{(i)} \\
&= \psi_i^{(i)\downarrow \omega_i^{(i)} \cap \lambda(ch(i))} \otimes \psi_{ch(i)}^{(i)} \otimes \bigotimes_{j=i+1, j\neq ch(i)}^{m} \psi_j^{(i)} \\
&= \psi_{ch(i)}^{(i+1)} \otimes \bigotimes_{j=i+1, j\neq ch(i)}^{m} \psi_j^{(i+1)} \\
&= \bigotimes_{j=i+1}^{m} \psi_j^{(i+1)}.
\end{aligned}$$

This proves the first equality of (4.13). The second is shown by induction over i. For $i = 1$, the equation is satisfied since

$$\left(\bigotimes_{j=1}^{m} \psi_j^{(1)}\right)^{\downarrow y_2} = \left(\bigotimes_{j=1}^{m} \psi_j\right)^{\downarrow y_2} = \phi^{\downarrow y_2}.$$

Let us assume that the equation holds for i,

$$\bigotimes_{j=i}^{m} \psi_j^{(i)} = \phi^{\downarrow y_i}.$$

Then, by transitivity of marginalization,

$$\bigotimes_{j=i+1}^{m} \psi_j^{(i+1)} = \left(\bigotimes_{j=i}^{m} \psi_j^{(i)}\right)^{\downarrow y_{i+1}} = (\phi^{\downarrow y_i})^{\downarrow y_{i+1}} = \phi^{\downarrow y_{i+1}}$$

which proves (4.13) for all i. □

Theorem 4.12 can now be verified.

Proof. By application of Equation (4.13) for $i = m - 1$, it follows that

$$y_m = \omega_m^{(m)}. \tag{4.16}$$

It remains to prove that $\omega_m^{(m)} = \lambda(m)$. For this purpose, it is sufficient to show that if $X \in \lambda(m)$ then $X \in \omega_m^{(m)}$ since $\omega_m^{(m)} \subseteq \lambda(m)$. Let $X \in \lambda(m)$. Then, according to the definition of the covering join tree for a projection problem, there exists a factor ψ_j with $X \in d(\psi_j)$. ψ_j is assigned to node $r = a(j)$ and therefore $X \in \omega_r^{(r)}$. Equation (4.8) implies that $X \in \omega_{ch(r)}^{(r+1)}$ and $X \in \omega_m^{(m)}$ follows by repeating this argument up to the root node m. □

The marginal $\phi^{\downarrow\lambda(m)}$ can now be used to solve the single-query projection problem. Moreover, it is sufficient to perform one last marginalization to the query x, since the latter is covered by the root node m.

$$\phi^{\downarrow x} = \left(\phi^{\downarrow\lambda(m)}\right)^{\downarrow x}. \tag{4.17}$$

A partial result of the collect theorem can also be applied for each sub-tree:

Corollary 4.14. *At the end of the collect algorithm, node i contains*

$$\psi_i^{(i)} = \left(\bigotimes_{j\in\mathcal{T}_i}\psi_j\right)^{\downarrow\omega_i^{(i)}}. \tag{4.18}$$

Proof. Node i is the root of the sub-tree $\mathcal{T}_i$. So, due to Equation (4.16), node i contains the marginal to $\omega_i^{(i)}$ of the factors associated with $\mathcal{T}_i$. □

Note that only inclusion between $\omega_i^{(i)}$ and $\lambda(i)$ holds, because it is in no way guaranteed that a corresponding factor for each variable in $\lambda(i)$ has been assigned to a node in the sub-tree $\mathcal{T}_i$. In other words, the root node i of $\mathcal{T}_i$ is not necessarily filled. However, as the following theorem states, the node labels can be scaled down to coincide with the domain of their content and the resulting tree will again be a join tree.

Theorem 4.15. *At the end of the collect algorithm executed on a join tree $\mathcal{T} = (V, E, \lambda, D)$, the labeled tree $\mathcal{T}^* = (V, E, \lambda^*, D)$ with $\lambda^*(i) = \omega_i^{(i)} = \omega_i^{(m)}$ for $i = 1, \ldots, m$ is a join tree.*

Proof. It will be shown that the running intersection property is still satisfied between the nodes of the labeled tree $\mathcal{T}^*$. Let i and j be two nodes whose reduced labels contain X, i.e. $X \in \lambda^*(i) \cap \lambda^*(j)$. Because $\mathcal{T}$ is a join tree, there exists a node k with $X \in \lambda(k)$ and $i, j \leq k$. By the same token, $X \in \lambda(ch(i))$. Then, from

$$\lambda^*(ch(i)) = \omega_{ch(i)}^{(i)} \cup (\lambda^*(i) \cap \lambda(ch(i)))$$

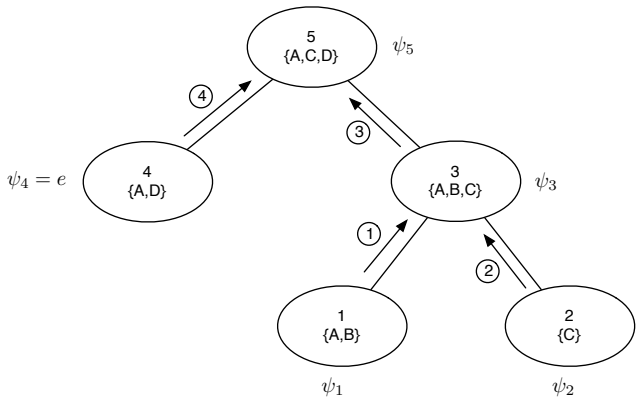
follows that $X \in \lambda^*(ch(i))$ and by induction $X \in \lambda^*(k)$. The same argument applies to the nodes on the path from j to k and therefore the running intersection property holds for $\mathcal{T}^*$. □

Figure 4.8 shows a complete run of the collect algorithm.

The following property regarding the relationship between node domains and labels is proved for later use.

Lemma 4.16. *It holds that*

$$\omega_i^{(m)} \cap \omega_{ch(i)}^{(m)} = \omega_i^{(m)} \cap \lambda(ch(i)). \tag{4.19}$$



	Message content:	**Node content:**	**Node domain:**
$\mu_{1\to 3}$	$\psi_1^{\downarrow d_1 \cap \lambda(3)}$	$\psi_1^{(5)} = \psi_1$	$d_1 = \omega_1^{(5)} = d(\psi_1)$
$\mu_{2\to 3}$	$\psi_2^{\downarrow d_2 \cap \lambda(3)}$	$\psi_2^{(5)} = \psi_2$	$d_2 = \omega_1^{(5)} = d(\psi_2)$
$\mu_{3\to 5}$	$(\psi_3 \otimes \mu_{1\to 3} \otimes \mu_{2\to 3})^{\downarrow d_3 \cap \lambda(5)}$	$\psi_3^{(5)} = \psi_3 \otimes \mu_{1\to 3} \otimes \mu_{2\to 3}$	$d_3 = \omega_3^{(3)} = d(\psi_3) \cup d(\mu_{1\to 3}) \cup d(\mu_{2\to 3})$
$\mu_{4\to 5}$	e	$\psi_4^{(5)} = e$	$d_4 = \omega_4^{(4)} = \emptyset$
-	-	$\psi_5^{(5)} = \psi_5 \otimes \mu_{3\to 5} = \phi^{\downarrow \lambda(5)}$	$d_5 = \omega_5^{5} = \lambda(5)$

Figure 4.8: A complete run of the collect algorithm.

Proof. The left hand part of this equation is clearly contained in the right hand part, because $\omega_{ch(i)}^{(m)} \subseteq \lambda(ch(i))$. The reversed inclusion is derived as follows, remembering that $\omega_i^{(i)} = \omega_i^{(m)}$ for $i = 1, \ldots, m$:

$$\begin{aligned} \omega_i^{(m)} \cap \omega_{ch(i)}^{(m)} &\supseteq \omega_i^{(m)} \cap \omega_{ch(i)}^{(i+1)} \\ &= \omega_i^{(m)} \cap \left(\omega_{ch(i)}^{(i)} \cup \left(\omega_i^{(i)} \cap \lambda(ch(i))\right)\right) \\ &= \left(\omega_i^{(m)} \cap \omega_{ch(i)}^{(i)}\right) \cup \left(\omega_i^{(m)} \cap \lambda(ch(i))\right) \\ &= \omega_i^{(m)} \cap \lambda(ch(i)). \end{aligned}$$

□

4.4.2 Shenoy-Shafer Architecture

The collect algorithm offers an adequate method for the solution of single-query projection problems. However, if more than one query has to be computed, collect must be repeated for every single query. On the other hand, one can always take the same but redirected join tree since all queries of the projection problem are covered by the latter. This causes a lot of redundant computation because only a few messages change between two runs of the collect algorithm with different root nodes. More precisely, the join tree can be directed to a new root node just by changing the direction of all edges connecting the new with the old root. Consequently, only

those messages that go along this path will change. This process of changing the root node is illustrated in Figure 4.9.

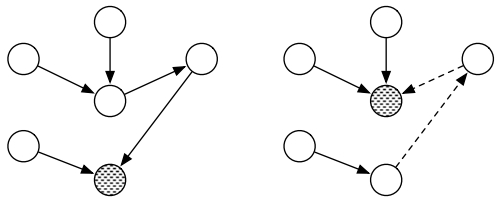


Figure 4.9: The root of a join tree may be changed by redirecting all edges on the path between the old and the new root node.

A widely-used technique for benefiting from already computed messages is to store them for later reuse. The Shenoy-Shafer architecture (Shenoy & Shafer, 1990) organizes this caching by installing *mailboxes* between neighboring nodes which store the exchanged messages. Figure 4.10 illustrates this concept schematically. The Shenoy-Shafer algorithm can be described by the following rules:

R1: Node i sends a message to its neighbor j as soon as it has received all messages from its other neighbors. This implies that leaves can send their messages right away.

R2: When node i is ready to send a message to neighbor j, it combines its initial node content with all messages from all other neighbors. The message is computed by marginalizing this result to the intersection of the result's domain and the receiving neighbor's node label.

The algorithm stops when every node has received all messages from its neighbors.

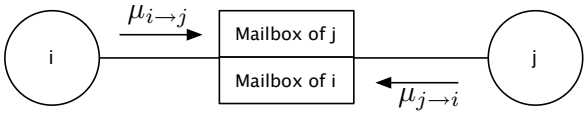


Figure 4.10: The Shenoy-Shafer architecture assumes mailboxes between neighboring nodes to store the exchanged messages.

In order to specify this algorithm formally, a new notation is introduced that determines the domain of the valuation described in R2. If i and j are neighbors, the domain of node i at the time where it sends a message to j is given by

$$\omega_{i\to j} \;=\; \omega_i \cup \bigcup_{k\in ne(i), j\neq k} d(\mu_{k\to i}). \tag{4.20}$$

- The message sent from node i to node j is

$$\mu_{i\to j} \quad = \quad \left(\psi_i \otimes \bigotimes_{k\in ne(i), j\neq k} \mu_{k\to i}\right)^{\downarrow \omega_{i\to j}\cap\lambda(j)} . \tag{4.21}$$

Theorem 4.17. *At the end of the message passing in the Shenoy-Shafer architecture, we obtain at node i*

$$\phi^{\downarrow\lambda(i)} \quad = \quad \psi_i \otimes \bigotimes_{j\in ne(i)} \mu_{j\to i}. \tag{4.22}$$

Proof. The important point is that the messages $\mu_{k\to j}$ do not depend on the actual schedule used to compute them. Due to this fact, an arbitrary node i can be selected as root node. Then, the edges are directed towards this root and the nodes are renumbered. Consequently, the message passing corresponds to the collect algorithm and the proposition follows from Theorem 4.12. □

Answering the queries of the projection problem from this last result demands one additional marginalization per query x_i,

$$\phi^{\downarrow x_i} \quad = \quad \left(\phi^{\downarrow\lambda(i)}\right)^{\downarrow x_i} . \tag{4.23}$$

4.4.3 Collect & Distribute Phase

For a previously fixed node numbering, Theorem 4.17 implies that

$$\phi^{\downarrow\lambda(i)} \;=\; \psi_i \otimes \bigotimes_{j\in pa(i)} \mu_{j\to i} \otimes \mu_{ch(i)\to i} \;=\; \psi_i^{(m)} \otimes \mu_{ch(i)\to i}. \tag{4.24}$$

This shows that the collect algorithm is extended by a single message coming from the child node in order to obtain the Shenoy-Shafer architecture. In fact, it is always possible to schedule a part of the messages in such a way that their sequence corresponds to the execution of the collect algorithm. The root node m will then be the first node which has received the messages of all its neighbors. It suggests itself to name this first phase of the Shenoy-Shafer architecture *collect phase* or *inward phase* since the messages are propagated from the leaves towards the root m. It furthermore holds that

$$\omega_{i\to ch(i)} \quad = \quad \omega_i^{(i)}$$

for this particular scheduling. Nevertheless, note that collect algorithm and collect phase are not identical. They adopt the same message sequence but differ in the way messages are treated. The collect algorithm combines incoming messages directly with the node content whereas the collect phase stores them in mailboxes and delays their combination to the end of the Shenoy-Shafer message passing. The messages that remain to be sent after the collect phase constitute the *distribute phase*. Clearly, the root node m will be the only node that may initiate this phase since it has all

necessary messages. Next, the parents of the root node will be able to send their messages and this propagation continues until the leaves are reached. Therefore, the distribute phase is also called *outward phase*.

Again, we list some consequences for later use.

Lemma 4.18. *It holds that*

$$\lambda(i) \quad = \quad \omega_i^{(m)} \cup (\lambda(i) \cap \lambda(ch(i))). \tag{4.25}$$

Proof. We have

$$\begin{aligned} \lambda(i) &= \lambda(i) \cup (\lambda(i) \cap \lambda(ch(i))) \\ &= \omega_i^{(m)} \cup (\omega_{ch(i)\to i} \cap \lambda(i)) \cup (\lambda(i) \cap \lambda(ch(i))) \\ &= \omega_i^{(m)} \cup (\lambda(i) \cap \lambda(ch(i))). \end{aligned}$$

The second equality follows from Equation (4.24). □

Lemma 4.19. *It holds that*

$$\omega_i^{(m)} - \left(\omega_i^{(m)} \cap \lambda(ch(i))\right) \quad = \quad \lambda(i) - (\lambda(i) \cap \lambda(ch(i)))\,. \tag{4.26}$$

Proof. Assume X contained in the right hand part of the equation. Thus, $X \in \lambda(i)$ but $X \notin \lambda(i) \cap \lambda(ch(i))$. From Equation (4.25) it can therefore be deduced that $X \in \omega_i^{(m)}$. Since

$$\omega_i^{(m)} \cap \lambda(ch(i)) \quad \subseteq \quad \lambda(i) \cap \lambda(ch(i)),$$

$X \notin \omega_i^{(m)} \cap \lambda(ch(i))$. This proves that

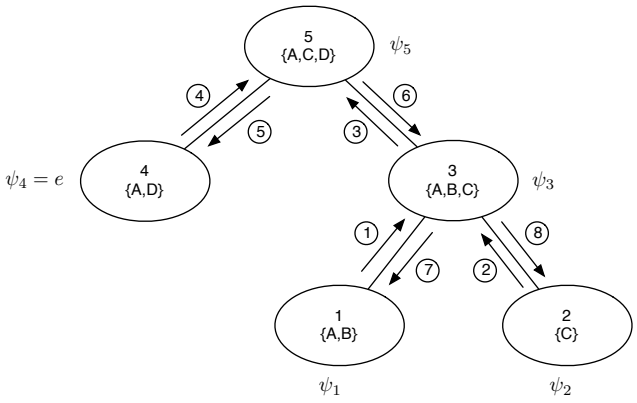
$$\omega_i^{(m)} - \left(\omega_i^{(m)} \cap \lambda(ch(i))\right) \quad \supseteq \quad \lambda(i) - (\lambda(i) \cap \lambda(ch(i)))\,.$$

It will next be shown that this inclusion cannot be strict. Assume to the contrary that X is contained in the left hand part but does not occur in the right hand set. So, $X \in \omega_i^{(m)} \subseteq \lambda(i)$ but $X \notin \omega_i^{(m)} \cap \lambda(ch(i))$ which in turn implies that $X \notin \lambda(ch(i))$ and consequently $X \notin \lambda(i) \cap \lambda(ch(i))$. Thus, $X \in \lambda(i) - (\lambda(i) \cap \lambda(ch(i)))$ which contradicts the assumption. Therefore, equality must hold. □

Figure 4.11 illustrates the message passing in the Shenoy-Shafer architecture. Note that only the messages are given in this picture since updating of the node content is postponed to the end of the message passing.

4.5 Lauritzen-Spiegelhalter Architecture

The Shenoy-Shafer architecture answers multi-query projection problems on any valuation algebra with adjoined identity element and is therefore the most general



	Message content:	**Node domain:**
$\mu_{1\to 3}$	$\psi_1^{\downarrow \omega_{1\to 3}\cap\lambda(3)}$	$\omega_{1\to 3} = d(\psi_1)$
$\mu_{2\to 3}$	$\psi_2^{\downarrow \omega_{2\to 3}\cap\lambda(3)}$	$\omega_{2\to 3} = d(\psi_2)$
$\mu_{3\to 5}$	$(\psi_3 \otimes \mu_{1\to 3} \otimes \mu_{2\to 3})^{\downarrow \omega_{3\to 5}\cap\lambda(5)}$	$\omega_{3\to 5} = d(\psi_3) \cup d(\mu_{1\to 3}) \cup d(\mu_{2\to 3})$
$\mu_{4\to 5}$	e	$\omega_{4\to 5} = d(e)$
$\mu_{5\to 4}$	$(\psi_5 \otimes \mu_{3\to 5})^{\downarrow \omega_{5\to 4}\cap\lambda(4)}$	$\omega_{5\to 4} = d(\psi_5) \cup d(\mu_{3\to 5})$
$\mu_{5\to 3}$	$(\psi_5 \otimes \mu_{4\to 5})^{\downarrow \omega_{5\to 3}\cap\lambda(3)}$	$\omega_{5\to 3} = d(\psi_5) \cup d(\mu_{4\to 5})$
$\mu_{3\to 1}$	$(\psi_3 \otimes \mu_{5\to 3} \otimes \mu_{2\to 3})^{\downarrow \omega_{3\to 1}\cap\lambda(1)}$	$\omega_{3\to 1} = d(\psi_3) \cup d(\mu_{5\to 3}) \cup d(\mu_{2\to 3})$
$\mu_{3\to 2}$	$(\psi_3 \otimes \mu_{5\to 3} \otimes \mu_{1\to 3})^{\downarrow \omega_{3\to 2}\cap\lambda(2)}$	$\omega_{3\to 2} = d(\psi_3) \cup d(\mu_{5\to 3}) \cup d(\mu_{1\to 3})$

Figure 4.11: A complete run of the Shenoy-Shafer architecture.

local computation architecture. Alternatively, research in the field of Bayesian networks has yielded another architecture called the *Lauritzen-Spiegelhalter architecture* (Lauritzen & Spiegelhalter, 1988) which can be applied if the valuation algebra has a division operator. Thus, the starting point is a local computation base for a multi-query projection problem with factors out of a separative valuation algebra (Φ^*, D). The knowledgebase factors $\phi_1, \ldots, \phi_n$ are elements of Φ^* but it is supposed that their combination $\phi = \phi_1 \otimes \cdots \otimes \phi_n$ is contained in Φ. This guarantees that ϕ can be projected to any possible query.

Remember, the inward phase of the Shenoy-Shafer architecture is almost equal to the collect algorithm with the only difference that incoming messages are not combined with the node content but kept in mailboxes. This is indispensable for the correctness of the Shenoy-Shafer architecture since otherwise, node i would get back its own message sent during the inward phase as part of the message obtained in the outward phase. In other words, some knowledge would be considered twice in every node, and this would falsify the final result. In case of a division operation, we can divide this doubly treated message out, and this idea is exploited by the Lauritzen-Spiegelhalter architecture as well as by the HUGIN architecture (Jensen

et al., 1990) which is the topic of Section 4.6.

The Lauritzen-Spiegelhalter architecture starts executing the collect algorithm towards root node m. Due to Section 4.4.1, node i contains just before computing its message for $ch(i)$

$$\psi_i' \;=\; \psi_i^{(m)} \;=\; \psi_i^{(i)} \;=\; \psi_i \otimes \bigotimes_{j \in pa(i)} \mu_{j \to i}. \tag{4.27}$$

Then, the message for the child node $ch(i)$ is computed as

$$\mu_{i \to ch(i)} \;=\; \left(\psi_i'\right)^{\downarrow \omega_i \cap \lambda(ch(i))}.$$

At this point, division comes into play. As soon as node i has sent its message, it divides the message out of its own content:

$$\psi_i' \;\otimes\; \mu_{i \to ch(i)}^{-1}. \tag{4.28}$$

This is repeated up to the root node. Thereupon, the outward propagation proceeds in a similar way. A node sends a message to all its parents when it has received a message from its child. In contrast, however, no division is performed during the outward phase. So, if node j is ready to send a message towards i and its current content is ψ_j', the message is

$$\mu_{j \to i} \;=\; (\psi_j')^{\downarrow \lambda(j) \cap \lambda(i)}$$

which is combined directly with the content of the receiving node i.

Theorem 4.20. *At the end of the Lauritzen-Spiegelhalter architecture, node i contains $\phi^{\downarrow \lambda(i)}$, provided that all messages during an execution of the Shenoy-Shafer architecture can be computed.*

Proof. Let μ' denote the messages sent during an execution of the Shenoy-Shafer architecture. We know that

$$\mu_{i \to j} \;=\; \mu_{i \to j}'$$

during the inward propagation and the theorem holds for the root node as a result of the collect algorithm. We prove that it is correct for all nodes by induction over the outward propagation phase using the correctness of Shenoy-Shafer. For this purpose, we schedule the outward propagation phase by taking the reverse node numbering. When a node j is ready to send a message towards i, node i stores

$$\psi_i \;\otimes\; (\mu_{i \to j}')^{-1} \;\otimes \bigotimes_{k \in ne(i), k \neq j} \mu_{k \to i}'. \tag{4.29}$$

By the induction hypothesis, the incoming message at step i is

$$\begin{aligned}\mu_{j\to i} &= \phi^{\downarrow\lambda(j)\cap\lambda(i)} \\ &= \left(\psi_j \otimes \bigotimes_{k\in ne(j)} \mu'_{k\to j}\right)^{\downarrow\lambda(j)\cap\lambda(i)} \\ &= \left(\psi_j \otimes \bigotimes_{k\in ne(j),k\neq i} \mu'_{k\to j}\right)^{\downarrow\omega_{j\to i}\cap\lambda(i)} \otimes \mu'_{i\to j} \\ &= \mu'_{j\to i}\otimes\mu'_{i\to j} \;=\; \mu'_{j\to i}\otimes\mu_{i\to j}, \end{aligned} \tag{4.30}$$

The third equality follows from the combination axiom, and the assumption is needed to ensure that $\mu'_{j\to i}$ exists. So we obtain at node i, when the incoming message $\mu_{j\to i}$ is combined with the actual content and using Theorem 4.17,

$$\psi_i \otimes \bigotimes_{k\in ne(i),k\neq j} \mu'_{k\to i}\otimes(\mu'_{i\to j})^{-1}\otimes\mu'_{j\to i}\otimes\mu'_{i\to j} \;=\; \phi^{\downarrow\lambda(i)}\otimes f_{\gamma(\mu'_{i\to j})}.$$

Finally, it follows from Lemma 2.12 that

$$\gamma(\mu'_{i\to j}) \;\leq\; \gamma(\mu'_{j\to i}\otimes\mu'_{i\to j}) \;=\; \gamma(\phi^{\downarrow\lambda(j)\cap\lambda(i)}) \;\leq\; \gamma(\phi^{\downarrow\lambda(i)}),$$

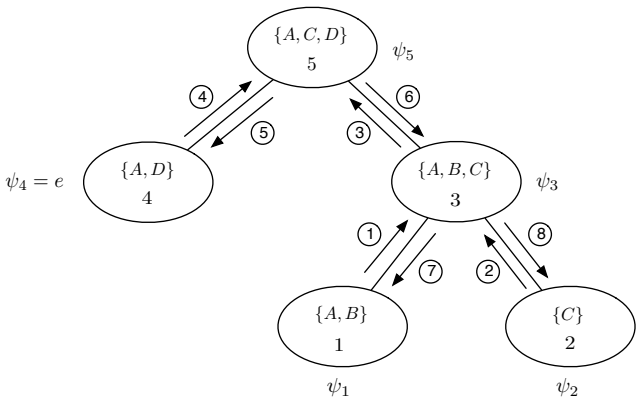
which proves the theorem. □

The proof is based on the messages used in the Shenoy-Shafer architecture. If they exist, Lauritzen-Spiegelhalter gives the correct results because the inward messages are identical in both architectures. Further, the Shenoy-Shafer messages are needed in Equation (4.30) for the application of the combination axiom. However, (Schneuwly, 2007) weakens this requirement by proving that the existence of the inward messages is in fact sufficient to make the Shenoy-Shafer architecture and therefore also Lauritzen-Spiegelhalter work. In case of a regular valuation algebra, however, all factors are elements of Φ and consequently all marginals exist. Theorem 4.20 can therefore be simplified by dropping this assumption.

We complete this section by an example showing a complete run of the Lauritzen-Spiegelhalter architecture. Since the messages of the inward phase correspond to the collect algorithm, the values for d_i are taken from Figure 4.8.

4.6 HUGIN Architecture

The HUGIN architecture (Jensen *et al.*, 1990) is a modification of Lauritzen-Spiegelhalter to make the divisions less costly. It postpones division to the outward phase such that the inward propagation corresponds again to the collect algorithm with the only difference that every message $\mu_{i\to j}$ is stored in the *separator* situated between all neighboring nodes i and j. These separators have the label $\lambda(i)\cap\lambda(j)$,



	Message content:	Node content:
$\mu_{1\to 3}$	$\psi_1^{\downarrow d_1 \cap \lambda(3)}$	$\psi_1' = \psi_1 \otimes \mu_{1\to 3}^{-1}$
$\mu_{2\to 3}$	$\psi_2^{\downarrow d_2 \cap \lambda(3)}$	$\psi_2' = \psi_2 \otimes \mu_{2\to 3}^{-1}$
$\mu_{3\to 5}$	$(\psi_3 \otimes \mu_{1\to 3} \otimes \mu_{2\to 3})^{\downarrow d_3 \cap \lambda(5)}$	$\psi_3' = \psi_3 \otimes \mu_{1\to 3} \otimes \mu_{2\to 3} \otimes \mu_{3\to 5}^{-1}$
$\mu_{4\to 5}$	e	$\psi_4' = e$
-	-	$\phi^{\downarrow \lambda(5)} = \psi_5' = \psi_5 \otimes \mu_{3\to 5}$
$\mu_{5\to 4}$	$\phi^{\downarrow \lambda(5) \cap \lambda(4)}$	$\phi^{\downarrow \lambda(4)} = \psi_4' \otimes \mu_{5\to 4}$
$\mu_{5\to 3}$	$\phi^{\downarrow \lambda(5) \cap \lambda(3)}$	$\phi^{\downarrow \lambda(3)} = \psi_3' \otimes \mu_{5\to 3}$
$\mu_{3\to 1}$	$\phi^{\downarrow \lambda(3) \cap \lambda(1)}$	$\phi^{\downarrow \lambda(1)} = \psi_1' \otimes \mu_{3\to 1}$
$\mu_{3\to 2}$	$\phi^{\downarrow \lambda(3) \cap \lambda(2)}$	$\phi^{\downarrow \lambda(2)} = \psi_2' \otimes \mu_{3\to 2}$

Figure 4.12: A complete run of the Lauritzen-Spiegelhalter architecture.

and we denote by S the set of all separators. In the following outward phase, the messages are computed identically to Lauritzen-Spiegelhalter. During transmission, however, they have to pass through the separator lying between the sending and receiving nodes. The separator becomes activated by the crossing message and holds it back in order to divide out its current content. Finally, the modified message is delivered to the destination, and the original incoming message becomes the new separator content.

Formally, the message sent from node i to $ch(i)$ during the inward phase is

$$\mu_{i\to ch(i)} = \left(\psi_i \otimes \bigotimes_{j \in pa(i)} \mu_{j\to ch(i)} \right)^{\downarrow \omega_i \cap \lambda(ch(i))} .$$

This message is stored in the separator. In the outward propagation phase node $ch(i)$ sends the message

$$\mu'_{ch(i)\to i} = \left(\psi_{ch(i)} \otimes \bigotimes_{k \in ne(i)} \mu_{k\to ch(i)} \right)^{\downarrow \lambda(i) \cap \lambda(ch(i))}$$

towards i. This messages arrives at the separator where it is altered to

$$\mu_{ch(i)\to i} \quad = \quad \mu'_{ch(i)\to i} \otimes \mu^{-1}_{i\to ch(i)}. \tag{4.31}$$

The message $\mu_{ch(i)\to i}$ is sent to node i and combined with the node content. This formula also shows the advantage of the HUGIN architecture in contrast to Lauritzen-Spiegelhalter. Divisions are performed in the separators exclusively and these have in general smaller domains than the join tree nodes.

Theorem 4.21. *At the end of the computations in the HUGIN architecture, each node $i \in V$ stores $\phi^{\downarrow\lambda(i)}$ and every separator $j \in S$ the marginal $\phi^{\downarrow\lambda(j)}$, provided that all messages during an execution of the Shenoy-Shafer architecture can be computed.*

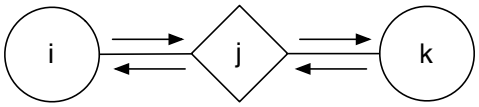


Figure 4.13: Separator in the HUGIN architecture.

Proof. The proof is based on the correctness of the Lauritzen-Spiegelhalter architecture and illustrated in Figure 4.13. First we consider the separators in the join tree (V, E, λ, D) as real nodes. Thus, let (V', E', λ', D) be such a modified join tree. We then adapt the assignment mapping a making it again surjective. We simply assign to every node in $(V' - V)$ the identity e and the extended assignment mapping is named a'. Now, the execution of the Lauritzen-Spiegelhalter architecture is started using $(V', E', \lambda', D, a')$.

Take a node $i \in V'$ which is not a separator in the original tree, i.e. $i \in V' \cap V$. It sends a message $\mu_{i\to j}$ in the inward propagation phase of Lauritzen-Spiegelhalter and afterwards divides it out of its current content which we abbreviate with η_i. By the construction of (V', E', λ', D), the receiving node $j = ch(i)$ is a separator in (V, E, λ, D), that is $j \in (V' - V)$. Node i contains $\eta_i \otimes (\mu_{i\to j})^{-1}$ and j stores $e \otimes \mu_{i\to j} = \mu_{i\to j}$ after this step. Then node j is ready to send a message towards node $k = ch(j)$. But we clearly have $\mu_{i\to j} = \mu_{j\to k}$. Since every emitted message is divided out of the store, the content of node j becomes

$$\mu_{i\to j} \otimes (\mu_{j\to k})^{-1} \quad = \quad f_{\gamma(\mu_{j\to k})}.$$

We continue with Lauritzen-Spiegelhalter and assume that node k is ready to send the message for node j during the outward propagation phase. This message equals $\phi^{\downarrow\lambda(k)\cap\lambda(j)}$ due to Theorem 4.20 and also becomes the new store of j according to Equation (4.30). The message sent from j towards i is finally $\phi^{\downarrow\lambda(j)\cap\lambda(i)} = \phi^{\downarrow\lambda(k)\cap\lambda(i)}$ so that we get there

$$\eta_i \otimes (\mu_{i\to j})^{-1} \otimes \phi^{\downarrow\lambda(i)\cap\lambda(k)} \quad = \quad \phi^{\downarrow\lambda(i)}.$$

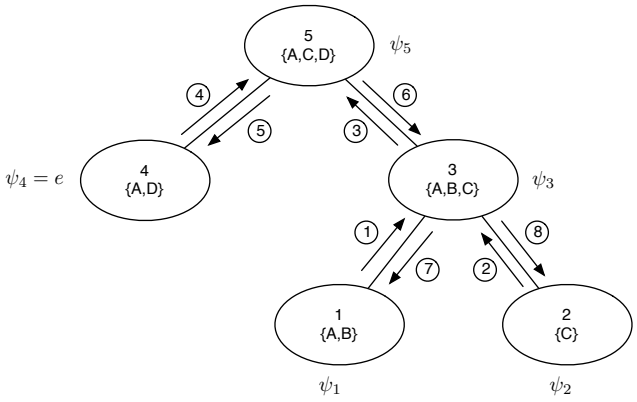
This follows again from the correctness of Lauritzen-Spiegelhalter. But

$$(\mu_{i\to j})^{-1} \otimes \phi^{\downarrow\lambda(i)\cap\lambda(k)}$$

is also the message from k towards i in the HUGIN architecture using (V, E, λ, D, a), which has passed already through the separator j. □

The messages used throughout the proof correspond to the Lauritzen-Spiegelhalter messages. Therefore, we may conclude that the existence of the collect messages is in fact a sufficient condition for Theorem 4.21. If on the other hand the valuation algebra is regular, this condition can again be ignored.

As usual, we close this section with Figure 4.14 showing a complete run of the HUGIN architecture. Since the inward phase corresponds to the collect algorithm, the missing values for d_i are taken from Figure 4.8.



	Message content:	Node content:
$\mu_{1\to3}$	$\psi_1^{\downarrow d_1\cap\lambda(3)}$	$\psi_1^{(5)} = \psi_1$
$\mu_{2\to3}$	$\psi_2^{\downarrow d_2\cap\lambda(3)}$	$\psi_2^{(5)} = \psi_2$
$\mu_{3\to5}$	$(\psi_3 \otimes \mu_{1\to3} \otimes \mu_{2\to3})^{\downarrow d_3\cap\lambda(5)}$	$\psi_3^{(5)} = \psi_3 \otimes \mu_{1\to3} \otimes \mu_{2\to3}$
$\mu_{4\to5}$	e	$\psi_4^{(5)} = e$
-	-	$\phi^{\downarrow\lambda(5)} = \psi_5^{(5)} = \psi_5 \otimes \mu_{3\to5}$
$\mu_{5\to4}$	$\phi^{\downarrow\lambda(5)\cap\lambda(4)} \otimes \mu_{4\to5}^{(-1)}$	$\phi^{\downarrow\lambda(4)} = \psi_4^{(5)} \otimes \mu_{5\to4}$
$\mu_{5\to3}$	$\phi^{\downarrow\lambda(5)\cap\lambda(3)} \otimes \mu_{3\to5}^{(-1)}$	$\phi^{\downarrow\lambda(3)} = \psi_3^{(5)} \otimes \mu_{5\to3}$
$\mu_{3\to1}$	$\phi^{\downarrow\lambda(3)\cap\lambda(1)} \otimes \mu_{1\to3}^{(-1)}$	$\phi^{\downarrow\lambda(1)} = \psi_1^{(5)} \otimes \mu_{3\to1}$
$\mu_{3\to2}$	$\phi^{\downarrow\lambda(3)\cap\lambda(2)} \otimes \mu_{2\to3}^{(-1)}$	$\phi^{\downarrow\lambda(2)} = \psi_2^{(5)} \otimes \mu_{3\to2}$

Figure 4.14: A complete run of the HUGIN architecture.

4.7 Idempotent Architecture

Section 2.6.3 introduced the property of idempotency as a trivial case of a division where every valuation is the inverse of itself. Thus, the Lauritzen-Spiegelhalter as

well as the HUGIN architecture can both be applied to projection problems with factors from an idempotent valuation algebra. We will see that both architectures simplify considerably under this setup.

In case of Lauritzen-Spiegelhalter, the message sent from i to $ch(i)$ during the inward phase is divided out of the node content. Since valuations are their own inverses, this reduces to a combination. Hence, instead of dividing out the emitted message, it is combined to the content of the sending node. By idempotency, this combination has no effect. Consider the inward message

$$\mu_{i\to ch(i)} = \left(\psi_i \otimes \bigotimes_{j\in pa(i)} \mu_{j\to i}\right)^{\downarrow \omega_i \cap \lambda(ch(i))}.$$

This message is divided out of the stored content of node i

$$\begin{aligned}\psi_i \otimes \mu_{i\to ch(i)}^{-1} \otimes \bigotimes_{j\in pa(i)} \mu_{j\to i} &= \psi_i \otimes \mu_{i\to ch(i)} \otimes \bigotimes_{j\in pa(i)} \mu_{j\to i} \\ &= \left(\psi_i \otimes \bigotimes_{j\in pa(i)} \mu_{j\to i}\right) \otimes \left(\psi_i \otimes \bigotimes_{j\in pa(i)} \mu_{j\to i}\right)^{\downarrow \omega_i \cap \lambda(ch(i))} \\ &= \psi_i \otimes \bigotimes_{j\in pa(i)} \mu_{j\to i}.\end{aligned}$$

The last equality follows from idempotency. Consequently, the solution of a multi-query projection problem consists in the execution of the collect algorithm towards a previously chosen root node m and a subsequent outward propagation phase. No division needs to be performed.

A similar argument can be found for the HUGIN architecture. The message sent from node $ch(i)$ to node i during the outward phase is $\phi^{\downarrow \lambda(i)\cap\lambda(ch(i))}$. It then passes the separator where the following computation takes place:

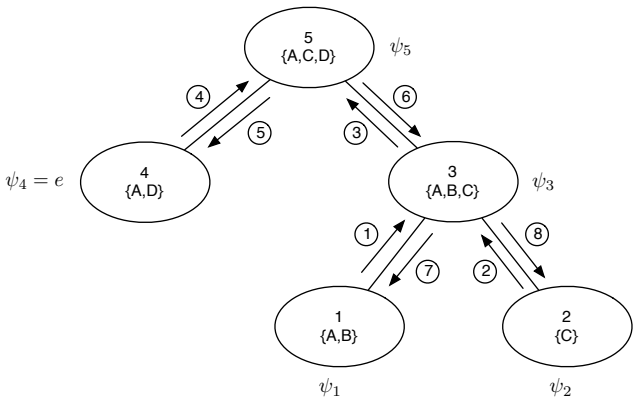
$$\mu_{i\to ch(i)}^{-1} \otimes \phi^{\downarrow \lambda(i)\cap\lambda(ch(i))} = \mu_{i\to ch(i)} \otimes \phi^{\downarrow \lambda(i)\cap\lambda(ch(i))} = \phi^{\downarrow \lambda(i)\cap\lambda(ch(i))}.$$

This follows from Equation (4.30) and idempotency. So, the valuations in the separators have no effect on the passing messages and therefore we may again do without the execution of division.

To sum it up, the architectures of Lauritzen-Spiegelhalter and HUGIN both simplify to the same, new architecture where no division needs to be done. We will subsequently refer to this simplification as the *idempotent architecture*, and a complete run of it is shown in Figure 4.15.

4.8 Architectures with Scaling

Obtaining scaled results from the computation of a projection problem is an essential requirement for many instances. Remember that the existence of scaling presupposes



	Message content:	**Node content:**
$\mu_{1\to 3}$	$\psi_1^{\downarrow d_1 \cap \lambda(3)}$	$\psi_1^{(5)} = \psi_1$
$\mu_{2\to 3}$	$\psi_2^{\downarrow d_2 \cap \lambda(3)}$	$\psi_2^{(5)} = \psi_2$
$\mu_{3\to 5}$	$(\psi_3 \otimes \mu_{1\to 3} \otimes \mu_{2\to 3})^{\downarrow d_3 \cap \lambda(5)}$	$\psi_3^{(5)} = \psi_3 \otimes \mu_{1\to 3} \otimes \mu_{2\to 3}$
$\mu_{4\to 5}$	e	$\psi_4^{(5)} = e$
-	-	$\phi^{\downarrow \lambda(5)} = \psi_5^{(5)} = \psi_5 \otimes \mu_{3\to 5}$
$\mu_{5\to 4}$	$\phi^{\downarrow \lambda(5) \cap \lambda(4)}$	$\phi^{\downarrow \lambda(4)} = \psi_4^{(5)} \otimes \mu_{5\to 4}$
$\mu_{5\to 3}$	$\phi^{\downarrow \lambda(5) \cap \lambda(3)}$	$\phi^{\downarrow \lambda(3)} = \psi_3^{(5)} \otimes \mu_{5\to 3}$
$\mu_{3\to 1}$	$\phi^{\downarrow \lambda(3) \cap \lambda(1)}$	$\phi^{\downarrow \lambda(1)} = \psi_1^{(5)} \otimes \mu_{3\to 1}$
$\mu_{3\to 2}$	$\phi^{\downarrow \lambda(3) \cap \lambda(2)}$	$\phi^{\downarrow \lambda(2)} = \psi_2^{(5)} \otimes \mu_{3\to 2}$

Figure 4.15: A complete run of the idempotent architecture.

a separative valuation algebra (Φ^*, D) with null elements. Then, using the properties derived in Section 2.7, this task could be performed as follows. Given a query x,

$$\begin{aligned}\left(\phi^{\downarrow x}\right)^{\downarrow} = \left(\phi^{\downarrow}\right)^{\downarrow x} &= \left((\psi_1 \otimes \cdots \otimes \psi_m)^{\downarrow}\right)^{\downarrow x} \\ &= \left((\psi_1^{\downarrow} \otimes \cdots \otimes \psi_m^{\downarrow})^{\downarrow}\right)^{\downarrow x} = \left(\psi_1^{\downarrow} \oplus \cdots \oplus \psi_m^{\downarrow}\right)^{\downarrow x}.\end{aligned}$$

Thus, scaled query answers are obtained if we solve the projection problem in the scaled valuation algebra $(\Phi^{\downarrow}, D)$ associated with (Φ^*, D). This on the other hand requires scaling all join tree factors ψ_i. Due to Equation (4.5), they are created by combining knowledgebase factors, and since a combination of scaled valuations does generally not produce a scaled result, we cannot directly assume that join tree factors are scaled – even if the knowledgebase is scaled. Additionally, the execution of $\oplus$ also demands a scaling operation. Altogether, this approach has to be discarded for efficiency reasons. By contrast, performing the embedding of scaling into the local computation architectures is more promising, and this will be done in this Section for all architectures discussed so far. The only exception is the idempotent architecture. Here, scaling has no effect since in an idempotent valuation algebra, all valuations are naturally scaled as shown in Equation (2.54).

4.8.1 Scaled Shenoy-Shafer Architecture

The *scaled Shenoy-Shafer architecture* first executes the collect phase so that $\phi^{\downarrow\lambda(m)}$ can be computed in the root node. Then, the root content ψ_m is replaced by

$$\psi_m \otimes \left(\phi^{\downarrow\emptyset}\right)^{-1} = \psi_m \otimes \left((\phi^{\downarrow\lambda(m)})^{\downarrow\emptyset}\right)^{-1}. \tag{4.32}$$

When the distribute phase starts, the outgoing messages from the root node use this modified node content. The result of these computations is stated in the following theorem.

Theorem 4.22. *At the end of the message passing in the scaled Shenoy-Shafer architecture, we obtain at node i*

$$\left(\phi^{\downarrow}\right)^{\downarrow\lambda(i)} = \psi_i \otimes \bigotimes_{j\in ne(i)} \mu'_{j\to i}, \tag{4.33}$$

where $\mu'_{j\to i}$ are the modified messages from the scaled architecture.

Proof. The messages of the collect phase do not change. We have $\mu_{i\to ch(i)} = \mu'_{i\to ch(i)}$. For the distribute messages, we show that

$$\mu'_{ch(i)\to i} = \mu_{ch(i)\to i} \otimes \left(\phi^{\downarrow\emptyset}\right)^{-1}. \tag{4.34}$$

The distribute message sent by the root node to parent i is

$$\begin{aligned}
\mu'_{m\to i} &= \left(\psi_m \otimes \left(\phi^{\downarrow\emptyset}\right)^{-1} \otimes \bigotimes_{j\in ne(m), j\neq i} \mu_{j\to m}\right)^{\downarrow\omega_{m\to i}\cap\lambda(i)} \\
&= \left(\psi_m \otimes \bigotimes_{j\in ne(m), j\neq i} \mu_{j\to m}\right)^{\downarrow\omega_{m\to i}\cap\lambda(i)} \otimes \left(\phi^{\downarrow\emptyset}\right)^{-1} \\
&= \mu_{m\to i} \otimes \left(\phi^{\downarrow\emptyset}\right)^{-1}.
\end{aligned}$$

This follows from the combination axiom. Hence, Equation (4.34) holds for all messages that are emitted by the root node. We proceed by induction and assume

that the proposition holds for $\mu_{ch(i)\to i}$. Then, by application of the combination axiom, we have for all $j \in pa(i)$

$$\begin{aligned}
\mu'_{i\to j} &= \left(\psi_i \otimes \bigotimes_{k\in ne(i),k\neq j} \mu'_{k\to i}\right)^{\downarrow\omega_{i\to j}\cap\lambda(j)} \\
&= \left(\psi_i \otimes \mu'_{ch(i)\to i} \otimes \bigotimes_{k\in pa(i),k\neq j} \mu_{k\to i}\right)^{\downarrow\omega_{i\to j}\cap\lambda(j)} \\
&= \left(\psi_i \otimes \mu_{ch(i)\to i} \otimes \left(\phi^{\downarrow\emptyset}\right)^{-1} \otimes \bigotimes_{k\in pa(i),k\neq j} \mu_{k\to i}\right)^{\downarrow\omega_{i\to j}\cap\lambda(j)} \\
&= \left(\psi_i \otimes \left(\phi^{\downarrow\emptyset}\right)^{-1} \otimes \bigotimes_{k\in ne(i),k\neq j} \mu_{k\to i}\right)^{\downarrow\omega_{i\to j}\cap\lambda(j)} \\
&= \left(\psi_i \otimes \bigotimes_{k\in ne(i),k\neq j} \mu_{k\to i}\right)^{\downarrow\omega_{i\to j}\cap\lambda(j)} \otimes \left(\phi^{\downarrow\emptyset}\right)^{-1} = \mu_{i\to j} \otimes \left(\phi^{\downarrow\emptyset}\right)^{-1}.
\end{aligned}$$

By Theorem 4.17 we finally conclude that

$$\begin{aligned}
\psi_i \otimes \bigotimes_{j\in ne(i)} \mu'_{j\to i} &= \psi_i \otimes \mu'_{ch(i)\to i} \bigotimes_{j\in pa(i)} \mu_{j\to i} \\
&= \psi_i \otimes \mu_{ch(i)\to i} \otimes \left(\phi^{\downarrow\emptyset}\right)^{-1} \otimes \bigotimes_{j\in pa(i)} \mu_{j\to i} \\
&= \psi_i \otimes \bigotimes_{j\in ne(i)} \mu_{j\to i} \otimes \left(\phi^{\downarrow\emptyset}\right)^{-1} \\
&= \phi^{\downarrow\lambda(i)} \otimes \left(\phi^{\downarrow\emptyset}\right)^{-1} = \left(\phi^{\downarrow\lambda(i)}\right)^{\downarrow} = \left(\phi^{\downarrow}\right)^{\downarrow\lambda(i)}.
\end{aligned}$$

□

To sum it up, the execution of one single scaling operation in the root node scales all marginals in the Shenoy-Shafer architecture.

4.8.2 Scaled Lauritzen-Spiegelhalter Architecture

The Lauritzen-Spiegelhalter architecture first executes the collect algorithm with the extension that emitted messages are divided out of the node content. At the end of this collect phase, the root node contains $\phi^{\downarrow\lambda(m)}$. We pursue a similar strategy as in the scaled Shenoy-Shafer architecture and replace the root content by

$$(\phi^{\downarrow})^{\downarrow\lambda(m)} = \phi^{\downarrow\lambda(m)} \otimes \left(\phi^{\downarrow\emptyset}\right)^{-1}. \tag{4.35}$$

Then, the outward propagation proceeds as usual.

Theorem 4.23. *At the end of the scaled Lauritzen-Spiegelhalter architecture, node i contains $(\phi^{\downarrow})^{\downarrow\lambda(i)}$, provided that all messages during an execution of the Shenoy-Shafer architecture can be computed.*

Proof. By Equation (4.35), the theorem is satisfied for the root node. We proceed by induction and assume that the proposition holds for node $ch(i)$. Thus, node $ch(i)$ contains $(\phi^{\downarrow})^{\downarrow\lambda(ch(i))}$. The content of node i in turn is given by Equation (4.28). Applying Theorem 4.20, we obtain

$$\begin{aligned}\psi_i' \otimes \mu_{i\to ch(i)}^{-1} \otimes (\phi^{\downarrow})^{\downarrow\lambda(i)\cap\lambda(ch(i))} &= \psi_i' \otimes \mu_{i\to ch(i)}^{-1} \otimes \phi^{\downarrow\lambda(i)\cap\lambda(ch(i))} \otimes \left(\phi^{\downarrow\emptyset}\right)^{-1} \\ &= \phi^{\downarrow\lambda(i)} \otimes \left(\phi^{\downarrow\emptyset}\right)^{-1} \\ &= \left(\phi^{\downarrow\lambda(i)}\right)^{\downarrow} = \left(\phi^{\downarrow}\right)^{\downarrow\lambda(i)}.\end{aligned}$$

□

Again, a single execution of scaling in the root node scales all marginals in the Lauritzen-Spiegelhalter architecture.

4.8.3 Scaled HUGIN Architecture

Scaling in the HUGIN architecture is equal to Lauritzen-Spiegelhalter. We again execute the collect phase and modify the root content according to Equation (4.35). Then, the outward phase starts in the usual way.

Theorem 4.24. *At the end of the computations in the HUGIN architecture, each node $i \in V$ stores $(\phi^{\downarrow})^{\downarrow\lambda(i)}$ and every separator $j \in S$ the marginal $(\phi^{\downarrow})^{\downarrow\lambda(j)}$, provided that all messages during an execution of the Shenoy-Shafer architecture can be computed.*

Proof. By Equation (4.35), the theorem is satisfied for the root node. We proceed by induction and assume that the proposition holds for node $ch(i)$. Then, the separator lying between i and $ch(i)$ will update its stored content to

$$(\phi^{\downarrow})^{\downarrow\lambda(i)\cap\lambda(ch(i))} \otimes \mu_{i\to ch(i)}^{-1} = \phi^{\downarrow\lambda(i)\cap\lambda(ch(i))} \otimes \left(\phi^{\downarrow\emptyset}\right)^{-1} \otimes \mu_{i\to ch(i)}^{-1}.$$

Node i contains ψ_i' when receiving the forwarded separator content and computes

$$\psi_i' \otimes \phi^{\downarrow\lambda(i)\cap\lambda(ch(i))} \otimes \left(\phi^{\downarrow\emptyset}\right)^{-1} \otimes \mu_{i\to ch(i)}^{-1} = \phi^{\downarrow\lambda(i)} \otimes \left(\phi^{\downarrow\emptyset}\right)^{-1} = \left(\phi^{\downarrow}\right)^{\downarrow\lambda(i)}.$$

□

Once more, a single execution of scaling in the root node scales all marginals in the HUGIN architecture.

These adaptions for scaling bring the discussion of local computation to a first close. The last section of this chapter is dedicated to a rather cursory complexity analysis of local computation. In addition, we discuss some approaches to the efficient construction of covering join trees and also touch upon a couple of important efforts to achieve better performance, especially in the Shenoy-Shafer architecture.

4.9 Complexity & Improvements

In Section 4.2.4 we identified the domain size as a crucial factor for complexity. This holds, on one hand, for the memory used to store a valuation and, on the other hand, also for the complexity of the valuation algebra operations. In a covering join tree, (combinations of) knowledgebase factors are stored in join tree nodes and we always have $d(\psi_i) \subseteq \lambda(i)$, ψ_i denoting the content of node i. Thus, we conclude that memory complexity as well as the complexity of combination and marginalization during local computation are bounded by the cardinality of the largest join tree node label. This measure is captured by the *width* of the join tree.

4.9.1 Treewidth Complexity

We learned in Section 4.1.2 how knowledgebases are represented by hypergraphs which can be extended to model projection problems by simply adding the set of queries. Then, Definition 4.9 of a covering join tree for a projection problem corresponds to a *hypertree decomposition* of the hypergraph associated with this projection problem (Gottlob *et al.*, 1999). Consequently, we may carry over the important notion of *treewidth* (Robertson & Seymour, 1986) to covering join trees.

Definition 4.25. *The* width *of a covering join tree* $\mathcal{T} = (V, E, \lambda, D)$ *is defined by*

$$width(\mathcal{T}) \quad = \quad \max_{i \in V} |\lambda(i)| - 1. \tag{4.36}$$

In the literature, another measure named *hypertree decomposition width* is often used in this context. It corresponds to the maximum number of knowledgebase factors that are assigned to the same node of the covering join tree (Zabiyaka & Darwiche, 2007).

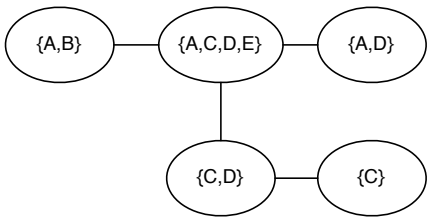


Figure 4.16: A join tree whose width equals 3.

The notion of width therefore reflects the efficiency of local computation on a particular join tree, and the solution of a projection problem by local computation is therefore most efficient if the covering join tree with the smallest width is used. This measure is called the *treewidth* of the projection problem.

4.9.2 Efficient Join Tree Construction

The bad news first – finding join trees with minimum width is known to be NP-complete (Arnborg *et al.*, 1987). Thus, if we want to achieve reasonable local computation complexity, we are generally forced to fall back on heuristics (Rose, 1970; Bertele & Brioschi, 1972; Yannakakis, 1981; Kong, 1986; Almond & Kong, 1991; Haenni & Lehmann, 1999). An overview of recommended heuristics can be found in (Lehmann, 2001) and a comparison is drawn by (Cano & Moral, 1995). From the bird's eye view, most heuristics are based on the choice of some appropriate variable elimination sequence that includes all variables in the knowledgebase. A covering join tree can then be constructed using Shenoy's fusion algorithm (Shenoy, 1992b), but without executing valuation algebra operations. This results in a local computation base which provides all components for local computation. However, this sketched join tree construction process requires a lot of knowledgebase rearrangement operations which in turn can lead to performance problems. A particularly efficient way for join tree construction starting with a knowledgebase and a predefined variable elimination sequence has been proposed by (Lehmann, 2001) and uses a so-called *variable-valuation-linked-list*. This data structure was decisive for the efficient implementation of the NENOK framework.

4.9.3 Binary Shenoy-Shafer Architecture

In a *binary join tree*, each node has at most three neighbors, and (Shenoy, 1997) remarked that they generally allow better performance for the Shenoy-Shafer architecture. This has essentially two reasons:

- First, nodes with more than three neighbors compute a lot of redundant combinations. For illustration, consider the left hand join tree in Figure 4.17 where the messages sent by node 4 are:

$$\begin{aligned}
\mu_{4\to1} &= (\psi_4 \otimes \mu_{2\to4} \otimes \mu_{3\to4} \otimes \mu_{5\to4})^{\downarrow\omega_{4\to1}} \\
\mu_{4\to2} &= (\psi_4 \otimes \mu_{1\to4} \otimes \mu_{3\to4} \otimes \mu_{5\to4})^{\downarrow\omega_{4\to2}} \\
\mu_{4\to3} &= (\psi_4 \otimes \mu_{1\to4} \otimes \mu_{2\to4} \otimes \mu_{5\to4})^{\downarrow\omega_{4\to3}} \\
\mu_{4\to5} &= (\psi_4 \otimes \mu_{1\to4} \otimes \mu_{2\to4} \otimes \mu_{3\to4})^{\downarrow\omega_{4\to5}}.
\end{aligned}$$

  We remark immediately that some combinations of messages are computed more than once. Let us turn to the right hand join tree in Figure 4.17 which provides an alternative local computation base for the same projection problem under the assumption that no knowledgebase factor with domain $\{A, B, C, D\}$ exists. Here, no combination is executed more than once since this join tree is binary. Consider for example node 3:

$$\begin{aligned}
\mu_{3\to1} &= (\psi_3 \otimes \mu_{2\to3} \otimes \mu_{6\to3})^{\downarrow\omega_{3\to1}} \\
\mu_{3\to2} &= (\psi_3 \otimes \mu_{1\to3} \otimes \mu_{6\to3})^{\downarrow\omega_{3\to2}} \\
\mu_{3\to6} &= (\psi_3 \otimes \mu_{1\to3} \otimes \mu_{2\to3})^{\downarrow\omega_{3\to6}}.
\end{aligned}$$

- The second reason for preferring binary join trees is that some computations may take place on larger domains than actually possible. Comparing the two join trees in Figure 4.17, we observe that the width of the binary join tree is smaller compared to the left hand join tree. Even though the binary tree contains more nodes, the computations are more efficient in case of smaller width.

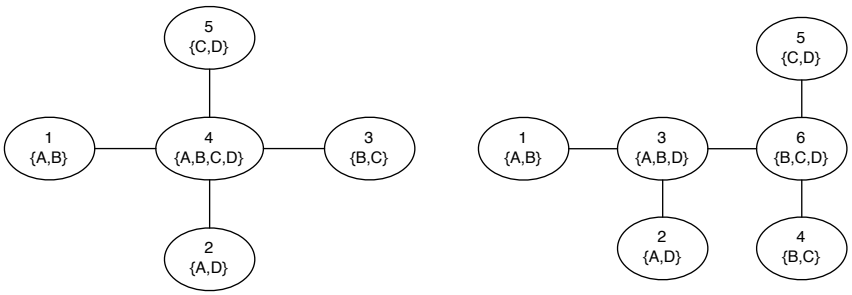


Figure 4.17: The benefit of using binary join trees.

Finally, we state that any join tree can be transformed into a binary join tree by adding a sufficient number of new nodes. A corresponding join tree binarization algorithm is given in (Lehmann, 2001). Further ways to improve the performance of local computation architectures by aiming to cut down on messages during the propagation are proposed by (Schmidt & Shenoy, 1998) and (Haenni, 2004).

4.10 Conclusion

Writing down the computational problem of inference in the generic language of a valuation algebra leads very naturally to the notion of projection problems, whose straightforward solution becomes intractable in most cases. The reason is that the memory usage as well as the complexity of combination and marginalization tend to increase exponentially with the domain size of the involved factors. It is therefore sensible to search for algorithms that solve projection problems by confining the domain of all intermediate results. This exactly is the promise of local computation architectures. All architectures described in this chapter presuppose a covering join tree whose nodes accommodate the knowledgebase factors. Such a join tree can always be found, and the concept of covering join trees convinces by its flexibility and an increase of efficiency with respect to the traditional join tree factorization. Four different local computation architectures have been studied in this chapter. The Shenoy-Shafer architecture is the most general algorithm and applies to all formalisms that fulfill the valuation algebra axioms. This is in opposition to the Lauritzen-Spiegelhalter and Hugin architectures which both exploit the presence of a division operator to improve performance. The fourth algorithm called idempotent architecture is even more restrictive by presuming an idempotent valuation algebra.

Its key advantage is that local computation becomes outstandingly simple under this setting. All non-idempotent architectures have further been extended to compute scaled marginals directly, and it turned out that only one single execution of the scaling operator in the root node is required for this task.

5
Distributed Knowledgebases

The definition of a knowledgebase as fundamental component of a local computation base does not provide any evidence about the (geographic) origin of its factors. But this is admittedly a very interesting question since the interpretation of valuations as pieces of knowledge or information suggests that these factors may come from different sources. Let us investigate this thought by reconsidering the notion of virtual processor introduced in Section 4.3.1. This time however, we put it into reality and say that every knowledgebase factor is stored on such a processor. Note that we use the term processor for an independent processing unit equipped with a private memory space. This leads to the concept of a *distributed knowledgebase*.

Definition 5.1. *A* distributed knowledgebase *is a knowledgebase* $\{\phi_1, \ldots, \phi_n\}$ *together with an assignment mapping* $\chi : \{\phi_1, \ldots, \phi_n\} \to P$ *determining the host processor of each valuation* ϕ_i *with respect to a given processor set* P*.*

Next, i we adopt the simplifying assumption that at most one knowledgebase factor is assigned to every join tree node by the factor assignment of Definition 4.5, we obtain in a very natural manner a (partial) assignment of processors to join tree nodes. In other words, the join tree turns into an overlay for the processor network. If messages are exchanged between join tree nodes during the local computation run, they are also transmitted within the processor network if different processors are assigned to the sending and receiving join tree nodes. The awareness that the size of valuations tends to increase exponentially with growing domains clearly affects the efficiency of communication in this system. It is therefore reasonable to reduce inter-processor communication as much as possible. This, in a few words, is the challenge of this chapter.

Section 5.1 raises the basic question of how the communication costs of transmitting valuations between processors can be measured. Further, we also set up the assignment of processors to join tree nodes in this section. Then, Section 5.2 formulates the task of minimizing communication costs as a decision problem and analyses its complexity and solution.

5.1 Measuring Communication Costs

Efficient communication is naturally a very important topic when dealing with projection problems that involve distributed knowledgebases. To make things easier, we first inspect the costs of transmitting single valuations between remote processors. In the complexity considerations of Section 4.2.4, $\omega(\phi)$ denoted the amount of memory that is used to store a valuation ϕ, and it is very natural to reapply this measure for estimating communication costs. Further, the efficiency of local computation has been shown by the fact that the domains of all intermediate factors are bounded by the join tree width. We may therefore assume that a valuation's weight shrinks under projection since otherwise the application of local computation would hardly be a runtime improvement. This motivates the following definition:

Definition 5.2. *Let (Φ^*, D) be a valuation algebra. A function $\omega : \Phi^* \to \mathbb{N} \cup \{0\}$ is called* weight function *if, for all $\phi \in \Phi^*$ and $x \subseteq d(\phi)$, we have $\omega(\phi) \geq \omega(\phi^{\downarrow x})$. Without loss of generality, we define $\omega(e) = 0$.*

Thus, we propose to model the communication costs $c_\phi(i, j)$ caused by sending some valuation ϕ from processor i to processor j by

$$c_\phi(i, j) \quad = \quad \omega(\phi) \cdot d_{i,j}, \tag{5.1}$$

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

If we then focus on solving projection problems over distributed knowledgebases, we quickly notice that sending all factors to a selected processor is beyond all question for larger knowledgebases. Moreover, the above definition guarantees more efficient communication if only projections of knowledgebase factors are transmitted. We therefore assign processors to join tree nodes so that only the local computation messages with bounded domains are exchanged. For that purpose, let us first make the simplifying assumption that the assignment mapping of Definition 4.5 allocates at most one knowledgebase factor to every join tree node. Such a distribution can always be enforced artificially as illustrated in Figure 5.1.

Consequently, every join tree factor ψ_i corresponds either to some knowledgebase factor or to the identity element. This naturally gives raise to a new processor assignment mapping $\xi : V \to P$ which is defined for all $i \in V$ as:

$$\xi(i) \quad = \quad \begin{cases} \chi(\phi_j) & \text{if } \psi_i = \phi_j, \\ p \in P \text{ arbitrary} & \text{otherwise.} \end{cases} \tag{5.2}$$

To each node, we assign either the processor where its knowledgebase factor is located or an arbitrary processor in case of the identity element. This goes perfectly

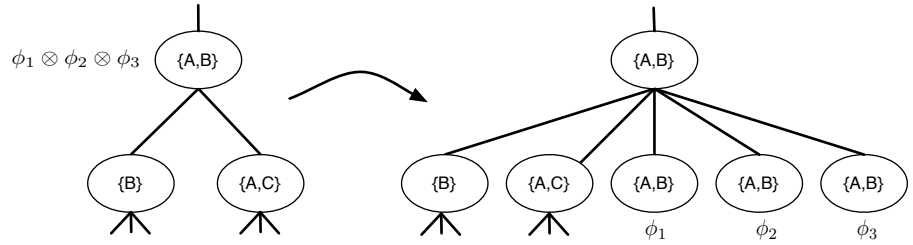


Figure 5.1: Modified knowledgebase factor allocation.

with the idea of join tree nodes as virtual processors, brought to reality by the assignment of a real-world processor.

The total communication costs of a complete run of the Shenoy-Shafer architecture are now obtained by adding together all costs that are caused by the exchanged messages. Due to Equation (5.1), the communication costs of transmitting a single local computation message $\mu_{i\to j}$ are $c(\mu_{i\to j})$. Observe that we use an abbreviated notation to avoid redundancy. Thus, for a given processor assignment mapping ξ, the total communication costs of a Shenoy-Shafer run are:

$$T_\xi \;=\; \sum_{i=1}^{m} \sum_{j\in ne(i)} c(\mu_{i\to j}) \;=\; \sum_{i=1}^{m} \sum_{j\in ne(i)} \omega(\mu_{i\to j}) \cdot d_{\xi(i),\xi(j)}. \tag{5.3}$$

Regrettably, all messages $\mu_{i\to j}$ must be known in order to compute their weight $\omega(\mu_{i\to j})$. Therefore, this formula can only be used to compute communication costs after the local computation algorithm has been executed. A prediction of communication costs before starting the actual run is generally impossible. However, this possibility of making predictions is indispensable for ensuring efficient communication. We are therefore looking for some additional requirement of the underlying valuation algebra which will allow the prediction of communication costs.

Definition 5.3. *Let (Φ^*, D) be a valuation algebra and ω its weight function. ω is called* weight predictor *if there exists a function $f : D \to \mathbb{N} \cup \{0\}$ such that for all $\phi \in \Phi^*$*

$$\omega(\phi) \;=\; f(d(\phi)).$$

Valuation algebras possessing a weight predictor are called weight predictable*.*

Thus, a valuation algebra is said to be weight predictable if the weight of a valuation can be computed using only its domain. This is in fact a very strong requirement as pointed out in the following example:

Example 5.1. *A possible weight function for probability potentials and connatural formalisms (which will be called semiring valuations in Chapter 6) is*

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

This is clearly a weight predictor since $\omega(\phi)$ can be computed from $d(\phi)$. A reasonable and often used weight function for relations is

$$\omega(\phi) \quad = \quad |d(\phi)| \cdot card(\phi),$$

where $card(\phi)$ denotes the number of tuples in ϕ. Regrettably, this is clearly not a weight predictor. $\ominus$

In view of this example, we indicate that a lot of research has been done into heuristical definitions of weight predictors for relations in order to optimize query answering in distributed databases (Tamer Özsu & Valduriez, 1999).

The property of weight predictability does indeed allow to compute an upper bound for the communication costs before the Shenoy-Shafer architecture has actually started:

$$\begin{aligned} T_\xi &= \sum_{i=1}^{m} \sum_{j \in ne(i)} f(\omega_{i \to j} \cap \lambda(j)) \cdot d_{\xi(i),\xi(j)} \\ &\leq \sum_{i=1}^{m} \sum_{j \in ne(i)} f(\lambda(i) \cap \lambda(j)) \cdot d_{\xi(i),\xi(j)} \\ &= 2 \sum_{i=1}^{m-1} f(\lambda(i) \cap \lambda(ch(i))) \cdot d_{\xi(i),\xi(ch(i))}. \end{aligned}$$

Here, $\omega_{i \to j} \cap \lambda(j)$ refers to the domain of the message sent from node i to node j, formally defined in Equation (4.21). This formula applies only to the join tree node labels and is independent of the actual node content.

5.2 Minimizing Communication Costs

With the ability to evaluate communication costs before starting local computation, we can now address the task of optimizing communication. Remember that Equation (5.2) assigns an arbitrary processor to all join tree nodes that initially do not hold a knowledgebase factor. Thus, if $\Psi = \{i \in V : \psi_i = e\}$ denotes the set of all nodes with identity element, there are $|P|^{|\Psi|}$ different assignment mappings ξ which all generate a different amount of communication costs. Clearly, a brute force determination of the best processor assignment becomes infeasible with increasing cardinality of Ψ. Following (Garey & Johnson, 1990), we transform this optimization task into a decision problem with ξ being the decision variable. Hence, we need to find some processor assignment ξ such that

$$\sum_{i=1}^{m-1} f_i \cdot d_{\xi(i),\xi(ch(i))} \quad \leq \quad B, \tag{5.4}$$

for a given upper bound $B \in \mathbb{N}$ and $f_i = f(\lambda(i) \cap \lambda(ch(i)))$. Subsequently, we will refer to this decision problem as the *partial distribution problem* (*PDP*) because of

its task of completing a partial processor distribution over join tree nodes.

The remaining part of this section is dedicated to a further analysis of the partial distribution problem. We will first introduce a very famous and well-studied decision problem called *multiway cut* problem, which has a variety of applications especially in the field of parallel computing systems. Afterwards, we show how the two problems are related and how insights on the multiway cut problem can be used to solve the partial distribution problem efficiently.

5.2.1 Analysis of the Partial Distribution Problem

(Dahlhaus *et al.*, 1994) originally initiated the research on the so-called *multiterminal cut* or *multiway cut* problem. Meanwhile, this problem has been studied extensively because of its extensive application area. Instead of the original description of the multiway cut problem, we will use the more general and illustrative definition given in (Erdös & Szekely, 1994):

> **Instance:** Given a graph $G = (V, E)$, a finite set of colors C, a positive number $B \in \mathbb{N}$ and a weight function $w : E \to \mathbb{N}$ assigning a weight $w(i, j)$ to each edge $(i, j) \in E$. Furthermore, we assume for $N \subseteq V$ a given partial coloration $\nu : N \to C$.
>
> **Question:** Is there a completed mapping $\overline{\nu} : V \to C$, such that $\overline{\nu}(i) = \nu(i)$ for all $i \in N$ and
>
> $$\sum_{(i,j)\in E,\ \overline{\nu}(i)\neq\overline{\nu}(j)} w(i,j) \quad \leq \quad B.$$

A partial coloration ν defines a partition of N by $N_i = \{n \in N, \nu(n) = i\}$. A given edge $(i, j) \in E$ is called *color-changing* in the coloration $\overline{\nu}$, if $\overline{\nu}(i) \neq \overline{\nu}(j)$. The set of color-changing edges in the coloration $\overline{\nu}$ separates every N_i from all other N_j and is therefore called *multiterminal cut*. This has given the decision problem its name. Figure 5.2 shows an instance of the multiway cut problem with three colors and a possible coloration of total weight 28.

The above definition of the multiway cut problem includes the so-called *color-independent* version of a weight function, which has also been used in (Dahlhaus *et al.*, 1994). A more general form is proposed by (Erdös & Szekely, 1994) and defined as $w : E \times C \times C \to \mathbb{N}$. In this case, the weight function is called *color-dependent* and the number $w(i, j, p, q)$ specifies the weight of the edge $(i, j) \in E$ if $\overline{\nu}(i) = p$ and $\overline{\nu}(j) = q$. Clearly, color-independence is reached if for any $(i, j) \in E$, $p_1 \neq q_1$ and $p_2 \neq q_2$, we have $w(i, j, p_1, q_1) = w(i, j, p_2, q_2)$. Finally, if $w(i, j) = c$ for all $(i, j) \in E$, the weight function is said to be *constant*. Note that without loss of generality, we can assume in this case that $c = 1$.

(Dahlhaus *et al.*, 1994) pointed out that the multiway cut problem is NP-complete even for $|N| = 3$, $|N_i| = 1$ and constant weight functions. For the special

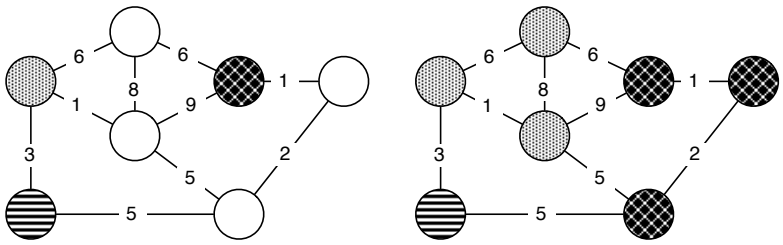


Figure 5.2: A multiway cut instance with three colors $C = \{dotted, striped, checked\}$ and a possible coloration with total weight 28. The numbers labeling the edges represent the weight function w.

case of a tree however, (Erdös & Szekely, 1994) proved that the multiway cut problem can be solved in polynomial time even for color-dependent weight functions. The corresponding algorithm has time complexity $O(|V| \cdot |C|^2)$.

The following theorem finally determines the complexity of the partial distribution problem, associated with the minimization of communication costs in local computation with weight predictable valuation algebras.

Theorem 5.4. *The partial distribution problem is solvable in polynomial time.*

Proof. We will show how a given PDP instance can be interpreted as a color-dependent multiway cut instance on the same tree, and conclude that each solution of the multiway cut problem also provides a solution for the original PDP instance. Thus, assume a given PDP instance on a join tree $G = (V, E)$. We define $N = \{i \in V : \psi_i \neq e\}$ and $C = \{c_1, \ldots, c_{|P|}\}$. The weights of the multiway cut instance are given by $w(i, j, p, q) = f_i \cdot d_{p,q}$ for $j = ch(i)$ and the initial coloration of the multiway cut instance $\nu : N \to C$ is defined as: $\nu(i) = c_j$ if $\chi(\psi_i) = p_j$ for $i \in N$, χ being the processor assignment mapping of the distributed projection problem. The upper bound B is equal for both instances.

Let $\overline{\nu}$ be a solution of the constructed multiway cut problem. Clearly, this induces a solution ξ of the corresponding PDP instance by defining: $\xi(i) = p_j$ if $\overline{\nu}(i) = c_j$. Therefore, and because G is a tree, we know that PDP adopts polynomial time complexity. □

This result shows that local computation is not only an efficient way to compute projection problems, but the underlying join tree structure also ensures efficient communication in case of projection problems over distributed and weight predictable knowledgebases. A corresponding algorithm to solve the partial distribution problem efficiently is the objective of the following subsection.

5.2.2 An Algorithm to Solve Partial Distribution Problems

In the proof of Theorem 5.4, we have seen that PDP is an application of the color-dependent multiway cut problem. Because of this tight relationship, we may reformulate the polynomial algorithm for the multiway cut problem given in (Erdös & Szekely, 1994) in terms of the PDP, such that it can be applied directly to the problem at hand. To start with, we consider a simplified version of the partial distribution problem: Let $le(V) = \{i \in V : pa(i) = \emptyset\}$ denote the set of leaf nodes of the join tree $G = (V, E)$. We assume that initially every leaf node hosts a knowledgebase factor and that identity elements are assigned to all other nodes, i.e. $\{i \in V : \psi_i \neq e\} = le(V)$. This simplified version will be called *autumnal PDP* since only leaf nodes are colored. A possible instance of such a join tree is shown in Figure 5.3.

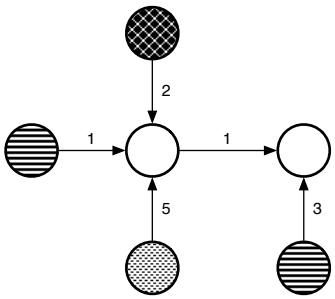


Figure 5.3: A PDP instance with three processors $P = \{dotted,\ striped,\ checked\}$ satisfying the imposed restriction that only leaves initially have an assigned processor. For simplicity, we assume that all distances in the processor network are constant, i.e. $d_{i,j} = 1$ for $i \neq j$ and $i, j \in P$. The numbers express the weight of the messages sent along the corresponding edge.

Definition 5.5. *Let $G = (V, E)$ be a tree. A penalty function is a map*

$$pen : P \times V \to (\mathbb{N} \cup \{0, \infty\})^{|P|}$$

such that $pen_i(v)$ constitutes the total weight of the sub-tree of node v, on condition that processor i has been assigned to node v.

The processor assignment algorithm that minimizes communication costs can now be formulated as a two-phase process: The first phase corresponds to an inward tree propagation which assigns penalty values to each node. Ensuing, the second phase propagates outwards and assigns a processor to all interior nodes such that the penalty values are minimized.

Phase I: For each leaf $v \in le(V)$, we define:

$$pen_i(v) \quad = \quad \begin{cases} 0 & \text{if } \chi(\psi_v) = i, \\ \infty & \text{otherwise.} \end{cases}$$

Then, we compute recursively for each node $v \in V - le(V)$:

$$pen_i(v) \quad = \quad \sum_{u \in pa(v)} \min_{j \in P} \{f_u \cdot d_{j,i} + pen_j(u)\}.$$

Figure 5.4 shows the penalty values obtained from applying the inward phase to the instance given in Figure 5.3.

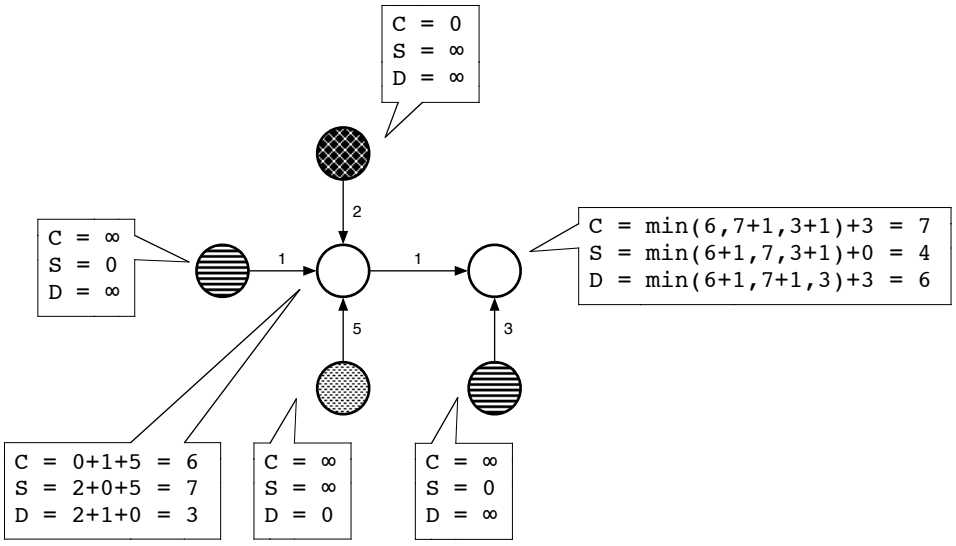


Figure 5.4: The penalty values computed during the inward phase. The capitals stand for: C = *checked*, S = *striped* and D = *dotted*.

Phase II: We now determine a complete processor assignment mapping $\xi : V \to P$ such that the total communication costs in the join tree are minimized. For the root node $r \in V$, we define $\xi(r) = i$ such that $pen_i(r) \leq pen_j(r)$ for all $i, j \in P$. Then we assign recursively to each other node $v \in V$, $\xi(v) = i$ if $f_v \cdot d_{i,\xi(ch(v))} + pen_i(v) \leq f_v \cdot d_{j,\xi(ch(v))} + pen_j(v)$ for all $i, j \in P$.

Note that whenever a new node is considered during the outward phase, it is ensured that a processor has already been assigned to its descendants. Therefore, the recursive minimization step of phase two is well-defined. Figure 5.5 finally shows how processors are assigned to the join tree nodes regarding the penalty values computed during the inward phase.

It is easy to see that at the end of the algorithm, we have $\xi(v) = \chi(\psi_v)$ for all $v \in le(V)$. Since G is a tree, there are $|V| - 1$ edges and therefore the algorithm consists of roughly $2 \cdot |V|$ steps. At each step, we compute $|P|^2$ sums and take the minimum, which results in a time complexity of $O(|V| \cdot |P|^2)$.

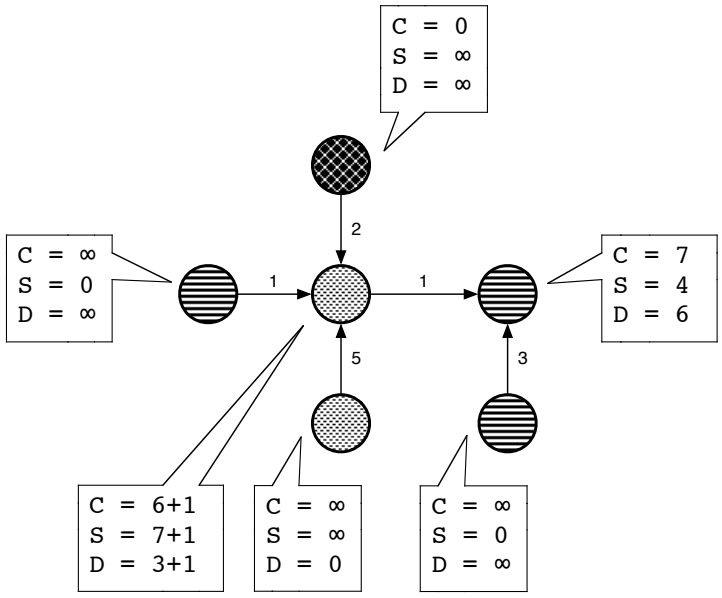


Figure 5.5: The completed processor assignment based on the penalty values from Figure 5.4.

This algorithm for the minimization of communication costs is based on the assumption that initially only leaf nodes possess an assigned processor. As the initial processor assignment arises from the knowledgebase factor distribution, we should now stress a more general version of this algorithm in order to solve arbitrary PDP instances. To do so, we only need to change the penalty values computed during the inward phase, which leads to the following reformulation of Phase I:

Phase I*: For each leaf $v \in le(V)$, we define:

$$pen_i(v) \quad = \quad \begin{cases} 0 & \text{if } \psi_v = e, \\ 0 & \text{if } \psi_v \neq e \wedge \chi(\psi_v) = i, \\ \infty & \text{otherwise.} \end{cases}$$

Then, we compute recursively for each node $v \in V - le(V)$:

$$pen_i(v) \quad = \quad \begin{cases} \sum_{u \in pa(v)} \min_{j \in P} \{f_u \cdot d_{j,i} + pen_j(u)\} & \text{if } \psi_v = e, \\ 0 & \text{if } \psi_v \neq e \wedge \chi(\psi_v) = i, \\ \infty & \text{otherwise.} \end{cases}$$

The modification for leaf nodes ensures that if the node contains an identity element, the assignment is postponed to the outward phase, and since the penalty value is set to zero, it will not influence the decisions in the outward phase but

accept the same processor as assigned to its child. The second formula realizes an implicit tree splitting when inner nodes already possess a processor. For such a node i, we cut every edge that connects i with its neighbors such that $|ne(i)|$ new PDP instances are generated. Then, a copy of node i is again added to every former neighbor of i. We accent that every copy of node i is now either a leaf or the root node in the corresponding tree. In the latter case, a slightly modified tree numbering transforms root nodes to leaf nodes again. To sum it up, we find $|ne(i)|$ autumnal PDP instances which can now be treated independently of one another. Clearly, the sum of minimum costs of the created instances equals the minimum costs of the original instance. These transformations are wrapped into the second formula.

5.2.3 Optimizations

Finally, we would like to point out that this more general algorithm computes the optimum processor assignment only in accordance with the given initial distribution. Consider for example the general PDP instance shown in the left hand part of Figure 5.6. The key point is that independently of the minimization algorithm's result, the total costs are built up from two weights w_{11} and $\min\{w_{12}, w_{13}\}$. Furthermore, if the minimization algorithm assigns the *striped* processor to the root node, the message coming from the deepest node is sent to the *checked* remote processor, treated and sent back to the *striped* processor when the root node is reached. Clearly, this overhead has emerged because of the imprudent initial distribution. A possible way to avoid such inefficiencies is given by the transformation shown in the right hand part of Figure 5.6, which extends the decision scope of the minimization algorithm. Now, the total costs are $\min\{w_{10}, w_{11} + \min\{w_{12}, w_{13}\}\}$, in any case at most as high as before. This is the statement of the following lemma.

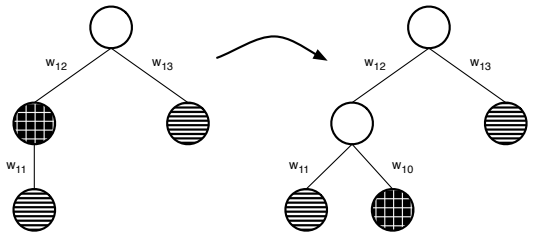


Figure 5.6: Delegation of assignment decisions to the minimization algorithm.

Lemma 5.6. *Every PDP instance can be transformed into an equivalent instance where processors are only assigned to leaf nodes. The communication costs of the new PDP instance are at least as small as in the original instance.*

The equivalence notion in this lemma means that the new join tree is again a covering join tree for the same projection problem. However, the drawback of this transformation clearly is that the new join tree demands more computational effort

during the distribute phase of local computation. It is therefore a balance between communication costs and computational effort.

5.3 Conclusion

Knowledge and information are naturally distributed resources, which also suggests their distributed processing. Local computation architectures work as message passing algorithms and are therefore well suited for an implementation as distributed algorithms. However, efficient communication is an important aspect in every such system, and this requires the possibility of estimating the weight of exchanged messages. In case of local computation, messages are valuations themselves, and we have seen that minimizing communication costs presupposes some way to estimate a valuation's weight even before this object actually exists. A sufficient condition for that purpose is weight predictability, and under this setting the problem reduces to the well-known multiway cut problem on trees, for which a low polynomial algorithm exists. In this perspective, the join tree can be viewed as an overlay of the processor network that ensures efficient communication. This is schematically illustrated in Figure 5.7.

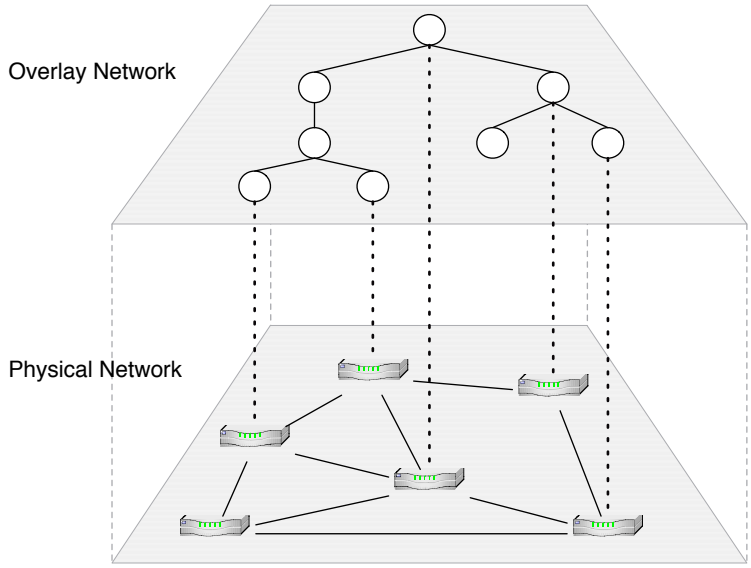


Figure 5.7: Interpretation of join trees as overlay network.

Part II

Semiring Valuation Algebras

6
Semiring Valuation Algebras

The mathematical framework of valuation algebras introduced in Chapter 2 provides sufficient structure for the application of local computation techniques, which took center stage in Chapter 4. In the process, numerous advantages of such a general framework have become clear, and we have seen for ourselves in Chapter 3 which different formalisms are unified under this perspective. It is fundamental for this generality that we did not accept any knowledge about the structure of valuations. Loosely spoken, they have only been considered as mathematical objects which possess a domain and which can further be combined and marginalized. For certain applications however, it is preferable to have additional knowledge about the buildup of valuations. This amounts to the identification of some families of valuation algebra instances which share a common internal structure.

The first family of valuation algebras, identified in this chapter, grows out of the obvious common structure of indicator functions and probability potentials introduced in Sections 3.2 and 3.4. Reduced to a common denominator, both instances are obtained by assigning values to configurations out of a finite configuration set. On the other hand, they seem at first glance to differ in the kind of values that are assigned and in the way combination and marginalization are computed. But this impression is deceptive. We will learn that both examples belong to a very large class of instances which are called semiring valuation algebras. A semiring is by itself a mathematical structure that comprises a set of values and two operations called addition and multiplication. Then, the values assigned to configurations are those of a semiring, and the operations of combination and marginalization can both be expressed using the two semiring operations. Following the course of Chapter 2, we will also study which semiring properties are required to induce the valuation algebra properties summarized in Figure 2.3.

The reasons why we are interested in semiring valuation algebras are manifold. On the one hand, semiring examples are very large in number and, because each semiring gives raise to a valuation algebra, we naturally obtain as many new valuation algebras as semirings exist. Moreover, if we interpret valuation algebras as formalisms for knowledge representation, we are not even able to explain for some

of these instances what kind of knowledge they model. Nevertheless, we have efficient algorithms for their processing – a quite exhilarating thought. On the other hand, some new applications for semiring valuation algebras, which will be studied in Chapter 8, do not directly correspond to the definition of projection problems given in Section 4.2. Nonetheless, we will see that they are solvable by local computation techniques.

This chapter starts with an introduction to semiring theory, restricted to the concepts that are needed to understand how semirings produce valuation algebras. To convince the reader of the richness of semiring examples, a large catalogue of instances will be given in Section 6.2. Then, based on (Kohlas & Wilson, 2006), it is shown how semirings produce valuation algebras, followed by an extensive study of how additional valuation algebra properties are obtained from the underlying semiring. This is the content of Sections 6.3 to 6.6. Finally, we close this chapter by examining what kind of valuation algebras grow out of the semiring examples from Section 6.2.

6.1 Semirings

We start this section by a formal definition of a semiring with the remark that the definition given here normally corresponds to a *commutative semiring*.

Definition 6.1. *A tuple $\mathcal{A} = \langle A, +, \times \rangle$ with binary operations $+$ and $\times$ is called* semiring *if $+$ and $\times$ are associative, commutative and if $\times$ distributes over $+$, i.e. for $a, b, c \in A$ we have*

$$a \times (b + c) \quad = \quad (a \times b) + (a \times c). \tag{6.1}$$

If there is an element $\mathbf{0} \in A$ such that $\mathbf{0} + a = a + \mathbf{0} = a$ and $\mathbf{0} \times a = a \times \mathbf{0} = \mathbf{0}$ for all $a \in A$, then $\mathcal{A}$ is called a semiring with *zero element*. A zero element is always unique and if it does not exist, it can be adjoined to any semiring by adding an extra element 0 to A and extending $+$ and $\times$ to $A \cup \{0\}$ by $a + 0 = 0 + a = a$ and $a \times 0 = 0 \times a = 0$ for all $a \in A \cup \{0\}$. Then, it is easy to verify that $\langle A \cup \{0\}, +, \times \rangle$ is a semiring, and we may assume for our further studies that every semiring contains a zero element. If furthermore $a + b = \mathbf{0}$ implies that $a = b = \mathbf{0}$ for all $a, b \in A$, then the semiring is called *positive*.

A semiring element $\mathbf{1} \in A$ is said to be a *unit element* if $\mathbf{1} \times a = a \times \mathbf{1} = a$ for all $a \in A$. Again, there can be at most one unit element. Furthermore, $\mathcal{A}$ is called *idempotent* if $a + a = a$ for all $a \in A$. In this case, the semiring $\mathcal{A} = \langle A, +, \times \rangle$ can be extended to include a unit element as follows: For each $a \in A$ define a new element a_1 such that $a \neq b$ implies $a_1 \neq b_1$. Let then $A' = A \cup A_1$, where $A_1 = \{a_1 : a \in A\}$. If a and b are arbitrary elements of A, we define $a +' b = a + b$. Otherwise, $a +' b_1$, $a_1 +' b$ and $a_1 +' b_1$ are all defined to be $(a + b)_1$. Additionally, we set $a \times' b = a \times b$, $a \times' b_1 = a_1 \times' b = (a \times b) + a$ and $a_1 \times' b_1 = (a_1 \times' b) +' a_1$. The system $\mathcal{A}' = \langle A', +', \times' \rangle$ is then a semiring with unit element 0_1, which is again idempotent. Without loss

of generality, it is assumed subsequently that all idempotent semirings possess a unit element, and we next show how a partial order can be defined on idempotent semirings.

6.1.1 Partially Ordered Semirings

On every idempotent semiring $\mathcal{A}$, we introduce a relation $\leq_{id}$ by:

$$a \leq_{id} b \quad \Leftrightarrow \quad a + b = b. \tag{6.2}$$

This is also called the *natural order* of an idempotent semiring. Some important properties of this relation are listed in the following lemma.

Lemma 6.2.

1. *$\leq_{id}$ is a partial order, i.e. reflexive, transitive and antisymmetric.*
2. *$a \leq_{id} b$ and $a' \leq_{id} b'$ imply $a + a' \leq_{id} b + b'$ and $a \times a' \leq_{id} b \times b'$.*
3. *$\leq_{id}$ is monotonic, i.e. $a \leq_{id} b$ implies $a + c \leq_{id} b + c$ and $a \times c \leq_{id} b \times c$.*
4. *$\forall a, b \in A$ we have $\mathbf{0} \leq_{id} a \leq_{id} a + b$.*
5. *A is positive.*
6. *$a + b = \sup\{a, b\}$*

Proof.

1. The relation $\leq_{id}$ is clearly reflexive, because for all $a \in A$ we have $a + a = a$ and therefore $a \leq_{id} a$. Next, suppose that $a \leq_{id} b$ and $b \leq_{id} c$. We have $c = b + c = a + b + c = a + c$ and therefore $a \leq_{id} c$. Finally, antisymmetry follows immediately since $a \leq_{id} b$ and $b \leq_{id} a$ imply that $b = a + b = a$, thus $a = b$.

2. Suppose $a \leq_{id} b$ and $a' \leq_{id} b'$ and therefore $a + b = b$ and $a' + b' = b'$. Then, $(a + a') + (b + b') = (a + b) + (a' + b') = (b + b')$, hence $a + a' \leq_{id} b + b'$. For the second part, we first show that $a \times a' \leq_{id} b \times a'$. This is the case because $(a \times a') + (b \times a') = (a + b) \times a' = b \times a'$. Similarly, we can conclude that $a' \times b \leq_{id} b' \times b$. Hence, we have $a \times a' \leq_{id} b \times a' \leq_{id} b' \times b$ and from transitivity, we obtain $a \times a' \leq_{id} b \times b'$.

3. Apply Property 2 for $a' = b' = c$.

4. We have $\mathbf{0} + a = a$ as well as $a + (a + b) = a + b$, hence $\mathbf{0} \leq_{id} a \leq_{id} a + b$.

5. Suppose that $a + b = \mathbf{0}$. Applying Property 4 we obtain $\mathbf{0} \leq_{id} a \leq_{id} a + b = \mathbf{0}$ and by transitivity and antisymmetry we conclude that $a = \mathbf{0}$. Similarly, we can derive $b = \mathbf{0}$.

6. Due to Property 4 we have $a, b \leq_{id} a + b$. Let c be another upper bound of a and b, $a \leq_{id} c$ and $b \leq_{id} c$. Then, by Property 2, $a + b \leq_{id} c + c = c$. Thus $a + b$ is the least upper bound.

□

Property 3 states that $\leq_{id}$ behaves monotonically under both semiring operations. However, for later use, we introduce another more restrictive version of monotonicity and write for this purpose

$$a <_{id} b \quad \Leftrightarrow \quad a \leq_{id} b \text{ and } a \neq b. \tag{6.3}$$

Definition 6.3. *An idempotent semiring is called* strictly monotonic *if for $c \neq \mathbf{0}$, $a <_{id} b$ implies that $a \times c <_{id} b \times c$.*

If $\mathcal{A} = \langle A, +, \times \rangle$ has a unit element with the property that

$$a + \mathbf{1} \quad = \quad \mathbf{1}, \tag{6.4}$$

then $\mathcal{A}$ is called *c-semiring* (*constraint semiring*). This definition corresponds to (Bistarelli *et al.*, 2002) and characterizes c-semirings by the fact that the unit element behaves absorbingly with respect to addition. We remark first that c-semirings are idempotent, since $\mathbf{1} + \mathbf{1} = \mathbf{1}$ implies that $a + a = a$ for all $a \in A$. Consequently, the above relation $\leq_{id}$ is well-defined in c-semirings, which leads to the following refinements of Lemma 6.2.

Lemma 6.4. *Let $\mathcal{A} = \langle A, +, \times \rangle$ be a c-semiring.*

1. *$\forall a, b \in A$ we have $\mathbf{0} \leq_{id} a \times b \leq_{id} a \leq_{id} a + b \leq_{id} \mathbf{1}$.*
2. *$a \times b = a$ implies that $a \leq_{id} b$.*
3. *If $\times$ is idempotent, $a \times b = \inf\{a, b\}$.*

Proof.

1. From Equation (6.4) follows that $a \leq_{id} \mathbf{1}$ for all $a \in A$. Because the properties from Lemma 6.2 still hold, it remains to be proved that $a \times b \leq_{id} a$ for all $a, b \in A$. This claim results from the distributive law since $a + (a \times b) = (a \times \mathbf{1}) + (a \times b) = a \times (\mathbf{1} + b) = a \times \mathbf{1} = a$.
2. We have $a + b = (a \times b) + b = (a + \mathbf{1}) \times b = \mathbf{1} \times b = b$.
3. Due to Property 1 we have $a \times b \leq_{id} a, b$. Let c be another lower bound of a and b, $c \leq_{id} a$ and $c \leq_{id} b$. Then, by Lemma 6.2 Property 2 and idempotency of $\times$, $c = c \times c \leq_{id} a \times b$. Thus, $a \times b$ is the greatest lower bound.

□

With supremum and infimum according to Lemmas 6.2 and 6.4, c-semirings with idempotent multiplication adopt the structure of a *lattice*. Moreover, the following theorem states that it is even distributive. This has been remarked by (Bistarelli *et al.*, 1997). In contrast however, the lattice in our case is not necessarily complete since we do not assume infinite summation in the definition of a c-semiring.

Theorem 6.5. *If $\mathcal{A}$ is a c-semiring with idempotent $\times$, then $\mathcal{A}$ is a* distributive lattice *with $a + b = \sup\{a, b\}$ and $a \times b = \inf\{a, b\}$.*

Proof. It remains to be proved that the two operations $+$ and $\times$ distribute over each other in case of an idempotent c-semiring. According to Equation (6.1), $\times$ distributes over $+$. On the other hand, we have for $a, b, c \in A$

$$\begin{aligned}(a+b)\times(a+c) &= a\times(a+b)+c\times(a+b)\\ &= (a\times a)+(a\times b)+(a\times c)+(b\times c)\\ &= a+a\times(b+c)+(b\times c)\\ &= a\times(\mathbf{1}+(b+c))+(b\times c)\\ &= (a\times\mathbf{1})+(b\times c) \;=\; a+(b\times c).\end{aligned}$$

□

6.1.2 Totally Ordered Semirings

Property 6 of Lemma 6.2 will play an important role in Chapter 8, where it is shown that idempotent semirings give rise to optimization problems if the relation $\leq_{id}$ is actually a *total order*. Due to the following result, such semirings are also called *addition-is-max semirings* (Wilson, 2004).

Lemma 6.6. *We have $a + b = \max\{a, b\}$ if, and only if, $\leq_{id}$ is total.*

Proof. If $\leq_{id}$ is total, we either have $a \leq_{id} b$ or $b \leq_{id} a$ for all $a, b \in A$. Assume that $a \leq_{id} b$. Then, $a + b = b$ and therefore $a + b = \max\{a, b\}$. The converse implication holds trivially. □

An important consequence of a total order concerns Definition 6.3:

Lemma 6.7. *In a totally ordered, strictly monotonic, idempotent semiring we have for $a \neq \mathbf{0}$ that $a \times b = a \times c$ if, and only if, $b = c$.*

Proof. Clearly, $b = c$ implies that $a \times b = a \times c$. On the other hand, assume $a \times b = a \times c$. Since $\mathcal{A}$ is totally ordered, either $b <_{id} c$, $c <_{id} b$ or $b = c$ must hold. Suppose $b <_{id} c$. Then, because $\times$ behaves strictly monotonic, we have $a \times b <_{id} a \times c$ which contradicts the assumption. A similar contradiction is obtained if we assume $c <_{id} b$. Therefore, $b = c$ must hold. □

So far, we studied the properties of the relation $\leq_{id}$, which has been introduced artificially for an idempotent semiring. It seems therefore natural to compare this relation with other total orders that may already exist in a particular semiring. This is pointed out in the following theorem.

Theorem 6.8. *Let $\mathcal{A} = \langle A, +, \times\rangle$ be an idempotent semiring with an arbitrary total, monotonic order $\leq$ defined over A. Then, we have for all $a, b \in A$:*

1. $\mathbf{0} \leq \mathbf{1} \;\Rightarrow\; [a \leq b \;\Leftrightarrow\; a \leq_{id} b]$.
2. $\mathbf{1} \leq \mathbf{0} \;\Rightarrow\; [a \leq b \;\Leftrightarrow\; b \leq_{id} a]$.

Proof. We remark first that $\mathbf{0} \leq \mathbf{1}$ implies $\mathbf{0} = \mathbf{0} \times b \leq \mathbf{1} \times b = b$ and therefore $a = a + \mathbf{0} \leq a + b$ by monotonicity of the assumed total order. Hence, $a, b \leq a + b$ for all $a, b \in A$. Similarly, we derive from $\mathbf{1} \leq \mathbf{0}$ that $a + b \leq a, b$.

1. Assume that $a \leq b$ and hence by monotonicity and idempotency $a+b \leq b+b = b$. Since $\mathbf{0} \leq \mathbf{1}$ we have $b \leq a+b$ and therefore $b \leq a+b \leq b$. Thus, we conclude by antisymmetry that $a + b = b$, i.e. $a \leq_{id} b$. On the other hand, if $a \leq_{id} b$, we obtain from $\mathbf{0} \leq \mathbf{1}$ that $a \leq a + b = b$ and therefore $a \leq b$ holds.

2. Assume that $a \leq b$ and hence by idempotency and monotonicity $a = a + a \leq a + b$. Since $\mathbf{1} \leq \mathbf{0}$ we have $a + b \leq a$ and therefore $a \leq a + b \leq a$. Thus, we conclude by antisymmetry that $a + b = a$, i.e. $b \leq_{id} a$. On the other hand, if $b \leq_{id} a$, we obtain from $\mathbf{1} \leq \mathbf{0}$ that $a = a + b \leq b$ and therefore $a \leq b$ holds.

□

Note that for semirings where $\mathbf{0} = \mathbf{1}$, both consequences of Theorem 6.8 are trivially satisfied because these semirings consist of only one single element. Further, this theorem states that in the case of an idempotent semiring, we can identify every total monotonic order with either the natural semiring order $\leq_{id}$ or its inverse. This will be especially important in Chapter 8, where it is shown that the projection problem turns into an optimization task when dealing with such semirings. It is then sufficient to maximize with respect to the natural semiring order, even if the real optimization problem demands a minimization task.

To close this section, we refer to Figure 6.4 that summarizes the most important semiring variations and their relationships, extended with the properties for inverse semiring elements that will be introduced in Section 6.6.

6.2 Semiring Examples

Before we actually develop the theory of semiring valuation algebras, we give a listing of some famous semiring instances in order to exemplify the introduced concepts. Furthermore, we will fall back on these semirings in Section 6.7 and show which kind of valuation algebras they induce.

6.2.1 Arithmetic Semirings

Let A be the set of non-negative real numbers $\mathbb{R}_{\geq 0}$ with $+$ and $\times$ designating the usual addition and multiplication. This is clearly a positive semiring with the number 0 as zero element and the number 1 as unit element. In the same way, we could also take the fields of real or rational numbers, or alternatively only non-negative integers. In the former two cases, the semiring would not be positive anymore, whereas ordinary addition and multiplication on non-negative integers $\mathbb{N} \cup \{0\}$ yield again a positive semiring. Arithmetic semirings are clearly not idempotent.

6.2.2 Bottleneck Semiring

Another semiring is obtained if we take $a + b = \max\{a, b\}$ and $a \times b = \min\{a, b\}$ over the set of real numbers $\mathbb{R} \cup \{+\infty, -\infty\}$. Then, $-\infty$ is the zero element, $+\infty$ the unity, and both operations are clearly idempotent. Since the unit element $+\infty$ behaves absorbing under maximization, it is a c-semiring and due to Theorem 6.5 a distributive lattice. The natural semiring order $\leq_{id}$ is total and coincides with the usual order between real numbers according to Theorem 6.8. Finally, it is very important to remark that $\leq_{id}$ is not strictly monotonic. If $c <_{id} a <_{id} b$, we have $a \times c = \min\{a, c\} = c = \min\{b, c\} = b \times c$.

6.2.3 Tropical Semiring

A very popular semiring is defined over the set of non-negative integers $\mathbb{N} \cup \{0, \infty\}$ with $a + b = \min\{a, b\}$ and the usual addition $+_{\mathbb{N}}$ for $\times$ with the convention that $a +_{\mathbb{N}} \infty = \infty$. This semiring is idempotent, ∞ is the zero element and the integer 0 is the unit element. The semiring order $\leq_{id}$ behaves strictly monotonic over $\times$ and conforms to the inverse order of non-negative integers as a consequence of Theorem 6.8. Alternatively, we could also take max for $+$ over the real numbers $\mathbb{R} \cup \{-\infty\}$. Multiplication is represented by $+_{\mathbb{R}}$ with $a +_{\mathbb{R}} (-\infty) = -\infty$. This semiring is again idempotent, has $-\infty$ as zero element and 0 as unit. $\leq_{id}$ behaves strictly monotonic over $\times$ but corresponds directly to the usual order of non-negative integers.

6.2.4 Truncation Semiring

An interesting variation of the tropical semiring is obtained if we take $A = \{0, \ldots, k\}$ for some integer k. Addition corresponds again to minimization but this time, we take the *truncated integer addition* for $\times$, i.e. $a \times b = \min\{a +_{\mathbb{N}} b, k\}$. This is an idempotent c-semiring, k is the zero element and 0 the unit. The semiring order corresponds again to the inverse integer order but strict monotonicity does not hold anymore.

6.2.5 Triangular Norm Semirings

Triangular norms (*t-norms*) were originally introduced in the context of probabilistic metric spaces (Menger, 1942; Schweizer & Sklar, 1960). They represent binary operations on the unit interval $A = [0, 1]$ which are commutative, associative, non-decreasing in both arguments, and have the number 1 as unit and 0 as zero element:

1. $\forall a, b, c \in [0, 1]$ we have $T(a, b) = T(b, a)$ and $T(a, T(b, c)) = T(T(a, b), c)$,
2. $a \leq a'$ and $b \leq b'$ imply $T(a, b) \leq T(a', b')$,
3. $\forall a \in [0, 1]$ we have $T(a, 1) = T(1, a) = a$ and $T(a, 0) = T(0, a) = 0$.

In order to obtain a semiring, we define the operation $\times$ on the unit interval by a t-norm and $+$ as max. This is a c-semiring with the number 0 as zero element and 1 as unit, where $\leq_{id}$ accords with the usual order of real numbers due to Theorem 6.8. Here are some typical t-norms:

- *Minimum T-Norm:* $T(a, b) = \min\{a, b\}$.
- *Product T-Norm:* $T(a, b) = a \cdot b$.
- *Lukasiewicz T-Norm:* $T(a, b) = \max\{a + b - 1, 0\}$.
- *Drastic Product:* $T(a, 1) = T(1, a) = a$ whereas $T(a, b) = 0$ in all other cases.

Instead of maximization, we may also take min for addition. Then, the number 1 becomes the zero element and 0 the unit. We reencounter a c-semiring where $\leq_{id}$ corresponds to the inverse order of real numbers. Strict monotonicity depends on the choice of the t-norm.

6.2.6 Semiring of Boolean Functions

Consider a set of r propositional variables. Then, the set of all *Boolean functions*

$$f : \{0, 1\}^r \to \{0, 1\} \tag{6.5}$$

forms a semiring with addition $f + g = \max\{f, g\}$ and multiplication $f \times g = \min\{f, g\}$ which are both evaluated pointwise. This semiring is idempotent with the constant mappings $f_{\mathbf{0}}$ being the zero element and $f_{\mathbf{1}}$ the unit element,

$$f_{\mathbf{0}} : \{0, 1\}^r \to 0 \qquad f_{\mathbf{1}} : \{0, 1\}^r \to 1.$$

It is furthermore a c-semiring and the natural semiring order $\leq_{id}$ is only partial. Of particular interest is the case where $r = 0$. This semiring is called *Boolean semiring* and consists of exactly two elements $f_{\mathbf{0}}$ and $f_{\mathbf{1}}$. It is this case convenient to identify the two functions with their corresponding value such that we may regard this semiring as the set $\{0, 1\}$ with maximization for addition and minimization for multiplication. It is more concretely a totally ordered, strictly monotonic c-semiring and $\leq_{id}$ coincides with the standard order in $\{0, 1\}$. The Boolean semiring is also a special case of the bottleneck semiring.

6.2.7 Distributive Lattices

Generalizing the forgoing example, we can say that every distributive lattice is an idempotent semiring with *join* for $+$ and *meet* for $\times$. The bottom element $\bot$ of the lattice becomes the zero element and, if the lattice has a top element $\top$, it becomes the unit element. In this case, the semiring is even a c-semiring with idempotent $\times$. This converse direction of Theorem 6.5 covers the semiring of Boolean functions as well as the bottleneck semiring. For example, the power set $\mathcal{P}(r)$ of a set r generates a distributive lattice with union $\cup$ for $+$ and intersection $\cap$ for $\times$. The empty set $\emptyset$ becomes the zero element, r is the unit element, and the partial ordering $\leq_{id}$ equals $\subseteq$. We will refer to this semiring as the *subset lattice*. Another interesting example of a distributive lattice is given by the natural numbers (or all positive divisors of a natural number n) with the greatest common divisor as meet and the least common multiple as join. We refer to (Davey & Priestley, 1990) for a broad listing of further examples of distributive lattices.

6.2.8 Multidimensional Semirings

Consider n possibly different semirings $\mathcal{A}_i = \langle A_i, +_i, \times_i \rangle$ for $i = 1, \ldots, n$. We define

$$A \quad = \quad A_1 \times \cdots \times A_n$$

with corresponding operations

$$\begin{aligned}(a_1, \ldots, a_n) + (b_1, \ldots, b_n) &= (a_1 +_1 b_1, \ldots, a_n +_n b_n) \\ (a_1, \ldots, a_n) \times (b_1, \ldots, b_n) &= (a_1 \times_1 b_1, \ldots, a_n \times_n b_n).\end{aligned}$$

Since both operations are executed component-wise, they clearly inherit associativity, commutativity and distributivity from the operations in $\mathcal{A}_i$. Hence, $\mathcal{A} = \langle A, +, \times \rangle$ becomes itself a semiring with zero element $\mathbf{0} = (\mathbf{0}_1, \ldots, \mathbf{0}_n)$ and unit $\mathbf{1} = (\mathbf{1}_1, \ldots, \mathbf{1}_n)$, provided that all $\mathcal{A}_i$ possess a unit element. If $+_i$ is idempotent for all $\mathcal{A}_i$, then so is $+$ in $\mathcal{A}$ and if all $\mathcal{A}_i$ are c-semirings, then $\mathcal{A}$ is also a c-semiring. Observe also that multidimensional semirings are in most cases only partially ordered.

There are naturally many other semiring examples that do not appear in this listing. Nevertheless, we named most semirings that are of practical importance for the theory of semiring valuation algebras. However, we also mention that every *ring* is a semiring with the additional property that inverse additive elements exist. In the same breath, *fields* are rings with multiplicative inverses. These remarks lead to further semiring examples such as the *ring of polynomials* or the *Galois field*.

6.3 Semiring Valuation Algebras

Equipped with this catalogue of semiring examples, we will now come to the highlight of this chapter and show how semirings induce valuation algebras by a very natural mapping from configurations to semiring values. This delightful and natural theory was developed in (Kohlas, 2004; Kohlas & Wilson, 2006), who also substantiated for the first time the relationship between semiring properties and the attributes of their induced valuation algebras. We will discover in Section 6.7 that formalisms for constraint modelling are an important subgroup of semiring valuation algebras. In this context, a similar framework to abstract different constraint satisfaction problems was introduced by (Bistarelli *et al.*, 1997; Bistarelli *et al.*, 2002), and this triggered the study of semiring valuation algebras. The importance of such formalisms outside the field of constraint satisfaction was furthermore explored by (Aji & McEliece, 2000), who also proved the applicability of the Shenoy-Shafer architecture. However, this is one of many conclusions to which we come by showing that semiring-based formalisms satisfy the valuation algebra axioms.

To start with, consider an arbitrary semiring $\mathcal{A} = \langle A, +, \times \rangle$ and a set r of variables with finite frames. A *semiring valuation* ϕ with domain $s \subseteq r$ is defined to be a function that associates a value from A with each configuration $\mathbf{x} \in \Omega_s$,

$$\phi : \Omega_s \to A. \tag{6.6}$$

We will subsequently denote the set of all semiring valuations with domain s by Φ_s and use Φ for all semiring valuations whose domain belongs to the subset lattice $D = \mathcal{P}(r)$. Next, the following operations in (Φ, D) are introduced:

1. *Labeling:* $\Phi \to D$: $d(\phi) = s$ if $\phi \in \Phi_s$.
2. *Combination:* $\Phi \times \Phi \to \Phi$: for $\phi, \psi \in \Phi$ and $\mathbf{x} \in \Omega_{d(\phi)\cup d(\psi)}$ we define

$$(\phi \otimes \psi)(\mathbf{x}) \quad = \quad \phi(\mathbf{x}^{\downarrow d(\phi)}) \times \psi(\mathbf{x}^{\downarrow d(\psi)}). \tag{6.7}$$

3. *Marginalization:* $\Phi \times D \to \Phi$: for $\phi \in \Phi$, $t \subseteq d(\phi)$ and $\mathbf{x} \in \Omega_t$ we define

$$\phi^{\downarrow t}(\mathbf{x}) \quad = \quad \sum_{\mathbf{y}\in\Omega_{s-t}} \phi(\mathbf{x}, \mathbf{y}). \tag{6.8}$$

Note that the definition of marginalization is well defined due to the associativity and commutativity of semiring addition. We now arrive at the central theorem of this chapter:

Theorem 6.9. *A system of semiring valuations (Φ, D) with labeling, combination and marginalization as defined above, satisfies the axioms of a valuation algebra.*

Proof. We verify the Axioms (A1) to (A6) of a valuation algebra given in Section 2.1. Observe that the labeling (A2), marginalization (A3) and domain (A6) properties are immediate consequences from the above definitions.

(A1) *Commutative Semigroup:* The commutativity of combination follows directly from the commutativity of the $\times$ operation in the semiring A and the definition of combination. Associativity is proved as follows. Assume that ϕ, ψ and η are valuations with domains $d(\phi) = s$, $d(\psi) = t$ and $d(\eta) = u$, then

$$\begin{aligned}
(\phi \otimes (\psi \otimes \eta))(\mathbf{x}) &= \phi(\mathbf{x}^{\downarrow s}) \times (\psi \otimes \eta)(\mathbf{x}^{\downarrow t\cup u}) \\
&= \phi(\mathbf{x}^{\downarrow s}) \times \Big(\psi((\mathbf{x}^{\downarrow t\cup u})^{\downarrow t}) \times \eta((\mathbf{x}^{\downarrow t\cup u})^{\downarrow u})\Big) \\
&= \phi(\mathbf{x}^{\downarrow s}) \times \Big(\psi(\mathbf{x}^{\downarrow t}) \times \eta(\mathbf{x}^{\downarrow u})\Big) \\
&= \phi(\mathbf{x}^{\downarrow s}) \times \psi(\mathbf{x}^{\downarrow t}) \times \eta(\mathbf{x}^{\downarrow u}).
\end{aligned}$$

The same result is obtained in exactly the same way for $((\phi\otimes\psi)\otimes\eta)(\mathbf{x})$. Thus associativity holds.

(A4) *Transitivity:* Transitivity of projection means simply that we can sum out variables in two steps. That is, if $t \subseteq s \subseteq d(\phi) = u$, then, for all $\mathbf{x} \in \Omega_t$,

$$\begin{aligned}
(\phi^{\downarrow s})^{\downarrow t}(\mathbf{x}) \quad &= \quad \sum_{\mathbf{y}\in\Omega_{s-t}} \phi^{\downarrow s}(\mathbf{x}, \mathbf{y}) \quad = \quad \sum_{\mathbf{y}\in\Omega_{s-t}} \sum_{\mathbf{z}\in\Omega_{u-s}} \phi(\mathbf{x}, \mathbf{y}, \mathbf{z}) \\
&= \quad \sum_{(\mathbf{y},\mathbf{z})\in\Omega_{u-t}} \phi(\mathbf{x}, \mathbf{y}, \mathbf{z}) \quad = \quad \phi^{\downarrow t}(\mathbf{x}).
\end{aligned}$$

(A5) *Combination:* Suppose that ϕ has domain t and ψ domain u and $\mathbf{x} \in \Omega_s$, where $t \subseteq s \subseteq t \cup u$. Then we have for $\mathbf{x} \in \Omega_s$,

$$\begin{aligned}(\phi \otimes \psi)^{\downarrow s}(\mathbf{x}) &= \sum_{\mathbf{y} \in \Omega_{(t \cup u)-s}} (\phi \otimes \psi)(\mathbf{x}, \mathbf{y}) = \sum_{\mathbf{y} \in \Omega_{u-s}} \Big(\phi(\mathbf{x}^{\downarrow t}) \times \psi(\mathbf{x}^{\downarrow s \cap u}, \mathbf{y})\Big) \\ &= \phi(\mathbf{x}^{\downarrow t}) \times \sum_{\mathbf{y} \in \Omega_{u-s}} \psi(\mathbf{x}^{\downarrow s \cap u}, \mathbf{y}) = \phi(\mathbf{x}^{\downarrow t}) \times \psi^{\downarrow s \cap u}(\mathbf{x}^{\downarrow s \cap u}) \\ &= (\phi \otimes \psi^{\downarrow s \cap u})(\mathbf{x}).\end{aligned}$$

□

To sum it up, a simple mapping from configurations to semiring values provides sufficient structure to give rise to a valuation algebra. Chapter 2 embraced also a comprehensive study of further properties that may occur in a valuation algebra. Following this guideline, we are now going to investigate which mathematical attributes are needed in the underlying semiring to guarantee the specific properties in the induced valuation algebra.

6.4 Semiring Valuation Algebras with Neutral Elements

If the semiring $\mathcal{A}$ which gives rise to the valuation algebra (Φ, D) has a unit element, then we have for every domain s a valuation $e_s(\mathbf{x}) = \mathbf{1}$ for all $\mathbf{x} \in \Omega_s$. This is the neutral valuation in the semigroup Φ_s with respect to combination, i.e. for all $\phi \in \Phi_s$ we have $e_s \otimes \phi = \phi \otimes e_s = \phi$. These neutral elements satisfy Property (A7):

(A7) *Neutrality:* We have by definition for all $\mathbf{x} \in \Omega_{s \cup t}$:

$$(e_s \otimes e_t)(\mathbf{x}) = e_s(\mathbf{x}^{\downarrow s}) \times e_t(\mathbf{x}^{\downarrow t}) = \mathbf{1} \times \mathbf{1} = \mathbf{1}. \tag{6.9}$$

Even though the presence of a unit element is sufficient for the existence of neutral valuations, they generally do not project to neutral valuations. This property was called stability and it turns out that idempotent addition is supplementary required to obtain a stable valuation algebra.

(A8) *Stability:* If the semiring is idempotent, we have $e_s^{\downarrow t} = e_t$ for $t \subseteq s$ by

$$e_s^{\downarrow t}(\mathbf{x}) = \sum_{\mathbf{y} \in \Omega_{s-t}} e_s(\mathbf{x}, \mathbf{y}) = \sum_{\mathbf{y} \in \Omega_{s-t}} \mathbf{1} = \mathbf{1}. \tag{6.10}$$

6.5 Semiring Valuation Algebras with Null Elements

Since every semiring contains a zero element, we can directly advise the candidate for a null semiring valuation in Φ_s. This is $z_s(\mathbf{x}) = \mathbf{0}$ for all $\mathbf{x} \in \Omega_s$. We clearly have for all $\phi \in \Phi_s$

$$\phi \otimes z_s(\mathbf{x}) = \phi(\mathbf{x}) \times z_s(\mathbf{x}) = \mathbf{0}$$

and therefore $\phi \otimes z_s = z_s$. But this is not yet sufficient because the nullity axiom must also be satisfied, and this comprises two requirements. First, remark that null elements project to null elements. More involved is the second requirement which claims that only null elements project to null elements. Positivity of the semiring is a sufficient condition for this axiom.

(A9) *Nullity:* Let $\mathbf{x} \in \Omega_t$ with $t \subseteq s = d(\phi)$. In a positive semiring, $\phi^{\downarrow t} = z_t$ implies that $\phi = z_s$. Indeed,

$$z_t(\mathbf{x}) \;=\; \mathbf{0} \;=\; \phi^{\downarrow t}(\mathbf{x}) \;=\; \sum_{\mathbf{y} \in \Omega_{s-t}} \phi(\mathbf{x}, \mathbf{y}) \tag{6.11}$$

implies that $\phi(\mathbf{x}, \mathbf{y}) = \mathbf{0}$ for all $\mathbf{y} \in \Omega_{s-t}$. Consequently, $\phi = z_s$.

Remember that due to Lemma 6.2 Property 5, idempotent semirings are positive and therefore induce valuation algebras with null elements.

6.6 Semiring Valuation Algebras with Inverse Elements

The presence of inverse valuations is of particular interest because they allow the application of specialized local computation architectures. In Section 2.6 we identified three different conditions for the existence of inverse valuations. Namely, these conditions are separativity, regularity and idempotency. Following (Kohlas & Wilson, 2006), we renew these considerations and investigate the requirements for a semiring to induce a valuation algebra with inverse elements. More concretely, it is effectual to identify the semiring properties that induce either separative, regular or idempotent valuation algebras. Then, the theory developed in Section 2.6 can be applied to identify the concrete inverse valuations. The commencement marks again the most general requirement called separativity.

6.6.1 Separative Semiring Valuation Algebras

According to (Hewitt & Zuckerman, 1956), a commutative semigroup A with operation $\times$ can be embedded into a semigroup which is a union of disjoint groups if, and only if, it is *separative*. This means that for all $a, b \in A$,

$$a \times b \;=\; a \times a \;=\; b \times b \tag{6.12}$$

implies $a = b$. Thus, let $\{G_\alpha : \alpha \in Y\}$ be such a family of disjoint groups with index set Y, whose union

$$G \;=\; \bigcup_{\alpha \in Y} G_\alpha \tag{6.13}$$

is a semigroup into which the commutative semigroup A is embedded. Hence, there exists an injective mapping $h : A \to G$ such that

$$h(a \times b) \;=\; h(a) \times h(b). \tag{6.14}$$

Note that the left hand multiplication refers to the semigroup operation in A whereas the right hand operation stands for the semigroup operation in G. If we affiliate every semigroup element $a \in A$ with its image $h(a) \in G$, we may assume without loss of generality that $A \subseteq G$.

Every group G_α contains a unit element, which we denote by f_α. These units are idempotent since $f_\alpha \times f_\alpha = f_\alpha$. Let $f \in G$ be an arbitrary idempotent element. Then, f must belong to some group G_α. Consequently, $f \times f = f \times f_\alpha$, which implies that $f = f_\alpha$ due to Equation (6.12). Thus, the group units are the only idempotent elements in G.

Next, it is clear that $f_\alpha \times f_\beta$ is also an idempotent element and consequently the unit of some group G_γ, i.e. $f_\alpha \times f_\beta = f_\gamma$. We define $\alpha \leq \beta$ if

$$f_\alpha \times f_\beta \quad = \quad f_\alpha. \tag{6.15}$$

This relation is clearly reflexive, antisymmetric and transitive, i.e. a partial order between the elements of Y. Now, if $f_\alpha \times f_\beta = f_\gamma$, then it follows that $\gamma \leq \alpha, \beta$. Let $\delta \in Y$ be another lower bound of α and β. We have $f_\alpha \times f_\delta = f_\delta$ and $f_\beta \times f_\delta = f_\delta$. Then, $f_\gamma \times f_\delta = f_\alpha \times f_\beta \times f_\delta = f_\alpha \times f_\delta = f_\delta$. So $\delta \leq \gamma$ and γ is therefore the greatest lower bound of α and β. We write $\gamma = \alpha \wedge \beta$ and hence

$$f_\alpha \times f_\beta \quad = \quad f_{\alpha \wedge \beta}.$$

To sum it up, Y forms a *semilattice*, a partially ordered set where the infimum exists between any pair of elements.

Subsequently, we denote the inverse of a group element $a \in G_\alpha$ by a^{-1}. Then, for $a, a^{-1} \in G_\alpha$ and $b, b^{-1} \in G_\beta$ it follows that $a \times b$, $a^{-1} \times b^{-1}$ and $f_{\alpha \wedge \beta}$ belong to the same group $G_{\alpha \wedge \beta}$. Because

$$(a \times b) \times (a^{-1} \times b^{-1}) \times (a \times b) \;=\; a \times f_\alpha \times b \times f_\beta \;=\; a \times b,$$

and

$$(a^{-1} \times b^{-1}) \times (a \times b) \times (a^{-1} \times b^{-1}) \;=\; a^{-1} \times f_\alpha \times b^{-1} \times f_\beta \;=\; a^{-1} \times b^{-1},$$

we conclude that $a \times b$ and $a^{-1} \times b^{-1}$ are inverses and are therefore contained in the same group. This holds also for $f_{\alpha \wedge \beta}$ since

$$(a \times b) \times (a^{-1} \times b^{-1}) \;=\; (a \times a^{-1}) \times (b \times b^{-1}) \;=\; f_\alpha \times f_\beta \;=\; f_{\alpha \wedge \beta}.$$

On this account,

$$(a \times b)^{-1} \quad = \quad a^{-1} \times b^{-1}.$$

We next introduce an equivalence relation between semigroup elements and say that $a \equiv b$ if a and b belong to the same group G_α. This is a congruence relation with respect to $\times$, since $a \equiv a'$ and $b \equiv b'$ imply that $a \times b \equiv a' \times b'$, which in turn

implies that the congruence classes are semigroups. Consequently, A decomposes into a family of disjoint semigroups,

$$A = \bigcup_{a \in A} [a].$$

Also, the partial order of Y carries over to equivalence classes by defining $[a] \leq [b]$ if, and only if, $[a \times b] = [a]$. Further, we have $[a \times b] = [a] \wedge [b]$ for all $a, b \in A$. Hence, the semigroups $[a]$ form a semilattice, isomorph to Y. We call the equivalence class $[a]$ the *support* of a. Observe also that $[\mathbf{0}] = \{\mathbf{0}\}$. This holds because for all $a \in [\mathbf{0}] = G_\gamma$ we have $a \times \mathbf{0} = \mathbf{0}$, which implies that $f_\gamma = \mathbf{0}$. Consequently, for $a \in [\mathbf{0}]$, $a \times a^{-1} = \mathbf{0}$ and therefore $a = a \times a \times a^{-1} = a \times \mathbf{0} = \mathbf{0}$.

The applied techniques that lead to the support decomposition of A are summarized in Figure 6.1. The following definition given in (Kohlas & Wilson, 2006) is crucial since it summarizes all requirements for a semiring to give rise to a separative valuation algebra. Together with the claim for a separative semigroup, it demands a decomposition which is monotonic under addition. This is needed to make allowance for the marginalization in Equation (2.28), as shown beneath.

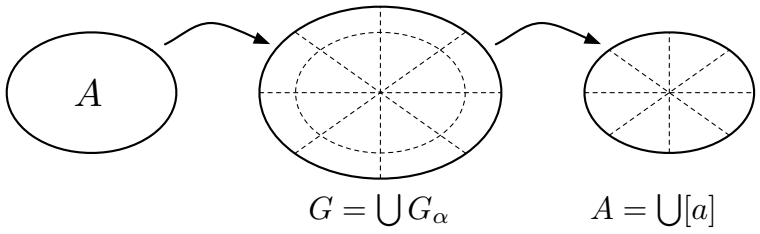


Figure 6.1: A separative semigroup A is embedded into a semigroup G that consists of disjoint groups G_α. The support decomposition of A is then derived by defining two semigroup elements as equivalent if they belong to the same group G_α.

Definition 6.10. *A semiring $\mathcal{A} = \langle A, +, \times \rangle$ is called* separative *if*

- *the multiplicative semigroup of $\mathcal{A}$ is separative,*
- *there is an embedding into a union of groups such that for all $a, b \in A$*

$$[a] \leq [a + b]. \tag{6.16}$$

The first condition expresses a kind of strengthening of positivity. This is the statement of the following lemma.

Lemma 6.11. *A separative semiring is positive.*

Proof. From Equation (6.16) we conclude $[\mathbf{0}] \leq [a]$ for all $a \in A$. Then, assume $a + b = \mathbf{0}$. Hence, $[\mathbf{0}] \leq [a] \leq [a + b] = [\mathbf{0}]$. Consequently, $a = b = \mathbf{0}$. □

The following theorem states that a separative semiring is sufficient to guarantee that the induced valuation algebra is also separative in terms of Definition 2.11.

Theorem 6.12. *Let (Φ, D) be a valuation algebra induced by a separative semiring. Then (Φ, D) is separative.*

Proof. Because $\mathcal{A}$ is separative, the same holds for the combination semigroup of Φ,

$$\phi \otimes \psi \;=\; \phi \otimes \phi \;=\; \psi \otimes \psi$$

implies $\phi = \psi$. Consequently, this semigroup can also be embedded into a semigroup which is a union of disjoint groups. In fact, the decomposition which is interesting for our purposes is the one induced by the decomposition of the underlying semiring. We say that $\phi \equiv \psi$, if

- $d(\phi) = d(\psi) = s$, and
- $\phi(\mathbf{x}) \equiv \psi(\mathbf{x})$ for all $\mathbf{x} \in \Omega_s$.

This is clearly an equivalence relation in Φ. Let $\phi, \psi, \eta \in \Phi$ and $\psi \equiv \eta$ with $d(\phi) = s$ and $d(\psi) = d(\eta) = t$. Then, it follows that $d(\phi \otimes \psi) = d(\phi \otimes \eta) = s \cup t$ and for all $\mathbf{x} \in \Omega_{s \cup t}$ we have

$$\phi(\mathbf{x}^{\downarrow s}) \times \psi(\mathbf{x}^{\downarrow t}) \quad \equiv \quad \phi(\mathbf{x}^{\downarrow s}) \times \eta(\mathbf{x}^{\downarrow t}).$$

We therefore have a combination congruence in Φ. It follows then that the equivalence classes $[\phi]$ are sub-semigroups of the combination semigroup of Φ.

Next, we define for a valuation $\phi \in \Phi_s$ the mapping $sp_{[\phi]} : \Omega_s \to Y$ by

$$sp_{[\phi]}(\mathbf{x}) = \alpha \quad \text{if} \quad \phi(\mathbf{x}) \in G_\alpha, \tag{6.17}$$

where Y is the semilattice of the group decomposition of the separative semiring. This mapping is well defined since $sp_{[\phi]} = sp_{[\psi]}$ if $[\phi] = [\psi]$. We define for a valuation ϕ with $d(\phi) = s$

$$G_{[\phi]} \quad = \quad \{g : \Omega_s \to G : \forall \mathbf{x} \in \Omega_s,\ g(\mathbf{x}) \in G_{sp_{[\phi]}(\mathbf{x})}\}.$$

It follows that $G_{[\phi]}$ is a group, and the semigroup $[\phi]$ is embedded in it. The unit element $f_{[\phi]}$ of $G_{[\phi]}$ is given by $f_{[\phi]}(\mathbf{x}) = f_{sp_{[\phi]}(\mathbf{x})}$ and the inverse of ϕ is defined by $\phi^{-1}(\mathbf{x}) = (\phi(\mathbf{x}))^{-1}$. This induces again a partial order $[\phi] \leq [\psi]$ if $f_{[\phi]}(\mathbf{x}) \leq f_{[\psi]}(\mathbf{x})$ for all $\mathbf{x} \in \Omega_s$, or $[\phi \otimes \psi] = [\phi]$. It is even a semilattice with $f_{[\phi \otimes \psi]} = f_{[\phi]} \wedge f_{[\psi]}$.

The union of these groups

$$G^* \quad = \quad \bigcup_{\phi \in \Phi} G_{[\phi]}$$

is a semigroup because, if $g_1 \in G_{[\phi]}$ and $g_2 \in G_{[\psi]}$, then $g_1 \otimes g_2$ is defined for $\mathbf{x} \in \Omega_{s \cup t}$, $d(\phi) = s$ and $d(\psi) = t$ by

$$g_1 \otimes g_2(\mathbf{x}) \quad = \quad g_1(\mathbf{x}^{\downarrow s}) \times g_2(\mathbf{x}^{\downarrow t}) \tag{6.18}$$

and belongs to $G_{[\phi\otimes\psi]}$ and is commutative as well as associative.

We have the equivalence $\phi \otimes \phi \equiv \phi$ because $[\phi]$ is closed under combination. From Equation (6.16) it follows that for $t \subseteq d(\phi)$ and all $\mathbf{x} \in \Omega_s$,

$$[\phi(\mathbf{x})] \quad \leq \quad [\phi^{\downarrow t}(\mathbf{x}^{\downarrow t})].$$

This means that $[\phi] \leq [\phi^{\downarrow t}]$ or also

$$\phi^{\downarrow t} \otimes \phi \quad \equiv \quad \phi. \tag{6.19}$$

We derived the second requirement for a separative valuation algebra given in Definition 2.11. It remains to show that the cancellativity property holds in every equivalence class $[\phi]$. Thus, assume $\psi, \psi' \in [\phi]$, $d(\phi) = s$ and for all $\mathbf{x} \in \Omega_s$

$$\phi(\mathbf{x}) \times \psi(\mathbf{x}) \quad = \quad \phi(\mathbf{x}) \times \psi'(\mathbf{x}).$$

Since all elements $\phi(\mathbf{x}), \psi(\mathbf{x})$ and $\psi'(\mathbf{x})$ are contained in the same group, it follows that $\psi(\mathbf{x}) = \psi'(\mathbf{x})$ by multiplication with $\phi(\mathbf{x})^{-1}$. Therefore, $\psi = \psi'$ which proves cancellativity of $[\phi]$. □

6.6.2 Regular Semiring Valuation Algebras

In the previous case, we exploited the fact that the multiplicative semigroup of a separative semiring can be embedded into a semigroup consisting of a union of disjoint groups. This allowed to introduce a particular equivalence relation between semiring elements which in turn leads to a decomposition of the induced valuation algebra into cancellative semigroups with the corresponding congruence relation satisfying the requirement of a separative valuation algebra. In this section, we start with a semiring whose semigroup decomposes directly into a union of groups. The mathematical requirement is captured by the following definition.

A semigroup A with an operation $\times$ is called *regular* if for all $a \in A$ there is an element $b \in A$ such that

$$a \times b \times a \quad = \quad a. \tag{6.20}$$

Section 2.6.2 introduced the Green relation in the context of a regular valuation algebra and we learned that the corresponding congruence classes are directly groups. This technique can be generalized to regular semigroups. We define

$$a \equiv b \quad \text{if, and only if,} \quad a \times A = b \times A, \tag{6.21}$$

and obtain a decomposition of A into disjoint congruence classes,

$$A \quad = \quad \bigcup_{a\in A} [a]. \tag{6.22}$$

These classes are commutative groups under $\times$, where every element a has a unique inverse denoted by a^{-1}. Further, we write $f_{[a]}$ for the identity element in the group

$[a]$ and take up the partial order defined by $f_{[a]} \leq f_{[b]}$ if, and only if, $f_{[a]} \times f_{[b]} = f_{[a]}$. This is again a semilattice as we know from Section 6.6.1. Finally, we obtain an induced partial order between the decomposition groups by $[a] \leq [b]$ if, and only if, $f_{[a]} \leq f_{[b]}$. From the bird's eye view, this is summarized in Figure 6.2.

Definition 6.13. *A semiring $\mathcal{A} = \langle A, +, \times \rangle$ is called* regular *if*

- *its multiplicative semigroup is regular,*
- *for all $a, b \in A$, $[a] \leq [a + b]$.*

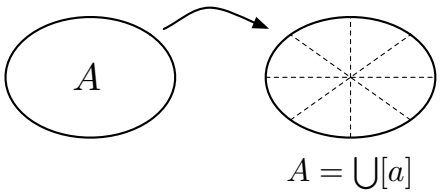


Figure 6.2: A regular semigroup A decomposes directly into a union of disjoint groups using the Green relation.

Theorem 6.14. *Let (Φ, D) be a valuation algebra induced by a regular semiring. Then (Φ, D) is regular.*

Proof. Suppose $\phi \in \Phi$ with $t \subseteq s = d(\phi)$ and $\mathbf{y} \in \Omega_t$. We define

$$\chi(\mathbf{y}) = (\phi^{\downarrow t}(\mathbf{y}))^{-1}. \tag{6.23}$$

Then, it follows that for any $\mathbf{x} \in \Omega_s$

$$\begin{aligned}(\phi \otimes \phi^{\downarrow t} \otimes \chi)(\mathbf{x}) &= \phi(\mathbf{x}) \times \phi^{\downarrow t}(\mathbf{x}^{\downarrow t}) \times \chi(\mathbf{x}^{\downarrow t}) \\ &= \phi(\mathbf{x}) \times \phi^{\downarrow t}(\mathbf{x}^{\downarrow t}) \times (\phi^{\downarrow t}(\mathbf{x}^{\downarrow t}))^{-1} \\ &= \phi(\mathbf{x}) \times f_{\phi^{\downarrow t}}.\end{aligned}$$

Here, the abbreviations f_ϕ and $f_{\phi^{\downarrow t}}$ are used for $f_{[\phi(\mathbf{x})]}$ and $f_{[\phi^{\downarrow t}(\mathbf{x}^{\downarrow t})]}$ respectively. Since $\mathcal{A}$ is regular, we have $[\phi(\mathbf{x})] \leq [\phi^{\downarrow t}(\mathbf{x}^{\downarrow t})]$ and $f_\phi \leq f_{\phi^{\downarrow t}}$. Thus

$$\begin{aligned}\phi(\mathbf{x}) \times f_{\phi^{\downarrow t}} &= (\phi(\mathbf{x}) \times f_\phi) \times f_{\phi^{\downarrow t}} \\ &= \phi(\mathbf{x}) \times (f_\phi \times f_{\phi^{\downarrow t}}) \\ &= \phi(\mathbf{x}) \times f_\phi \;=\; \phi(\mathbf{x}).\end{aligned}$$

This proves the requirement of Definition 2.19. □

We conclude this discussion by the remark that regular semirings are also separative. Regularity of the multiplicative semigroup implies that every element in A has an inverse and we derive from $a \times a = a \times b = b \times b$ that a and b are contained in the same group of the semiring decomposition, i.e. $[a] = [b]$. Multiplying with the inverse of a gives then $a = f_{[a]} \times b = b$. This proves that the multiplicative semigroup of a regular semiring is separative. Requirement (6.16) follows from the strengthening of positivity in Definition 6.13.

6.6.3 Cancellative Semiring Valuation Algebras

In Section 6.6.1 we started from a separative semigroup, which can be embedded into a semigroup consisting of a union of disjoint groups. Here, we discuss another special case in which the semigroup is embedded into a single group. The required property for the application of this technique is cancellativity. Remember that a semigroup A with operation $\times$ is called cancellative, if for $a, b, c \in A$,

$$a \times b \quad = \quad a \times c \tag{6.24}$$

implies $b = c$. Such a semigroup can be embedded into a group G by application of essentially the same technique as in Section 2.6.1. We consider pairs (a, b) of elements $a, b \in A$ and define

$$(a, b) = (c, d) \quad \text{if} \quad a \times d = b \times c.$$

Multiplication between pairs of semigroup elements is defined component-wise,

$$(a, b) \times (c, d) \quad = \quad (a \times c, b \times d).$$

This operation is well defined. It is furthermore associative, commutative and the multiplicative unit e is given by the pairs (a, a) for all $a \in A$. Note that all these pairs are equal with respect to the above definition. We then have

$$(a, b) \times (b, a) \;=\; (a \times b, a \times b),$$

which shows that (a, b) and (b, a) are inverses. Consequently, the set G of pairs (a, b) is a group into which A is embedded by the mapping $a \mapsto (a \times a, a)$. If A itself has a unit, then $1 \mapsto (1, 1) = e$. Without loss of generality, we may therefore consider A as a subset of G. This is summarized in Figure 6.3.

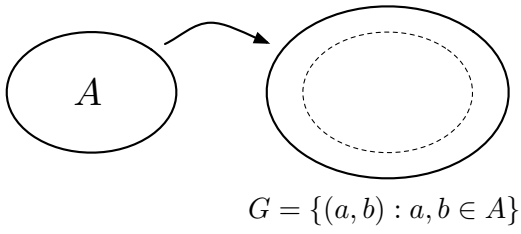


Figure 6.3: A cancellative semigroup A is embedded into a group G consisting of pairs of elements from A.

Let us return to semirings and show how cancellative semirings induce valuation algebras with inverse elements.

Definition 6.15. *A semiring $\mathcal{A} = \langle A, +, \times \rangle$ is called* cancellative *if its multiplicative semigroup is cancellative.*

A cancellative semiring is also separative since from $a \times a = a \times b$ it follows that $a = b$. We know that cancellative semirings can be embedded into a single group consisting of pairs of semiring elements, and since we have only one group, (6.16) is trivially fulfilled. Thus, cancellative semirings are particular simple cases of separative semirings and consequently induce separative valuation algebras according to Section 6.6.1. We also point out that no direct relationship between cancellative and regular semirings exist. Cancellative semirings are embedded into a single group whereas regular semirings decompose into a union of groups.

6.6.4 Idempotent Semiring Valuation Algebras

Section 2.6.3 introduced idempotency as a very strong condition which allows to turn a blind eye on division since every valuation is its own inverse. On the other hand, such valuation algebras are of particular interest since they allow for a very simple and efficient local computation architecture. Thus, we go about studying which semirings give rise to idempotent valuation algebras.

Theorem 6.16. *Let (Φ, D) be a valuation algebra induced by a c-semiring with idempotent $\times$. Then (Φ, D) is idempotent.*

Proof. From Lemma 6.4 Property 1 we conclude that

$$\phi(\mathbf{x}) \times \phi^{\downarrow t}(\mathbf{x}^{\downarrow t}) \quad \leq_{id} \quad \phi(\mathbf{x}).$$

On the other hand, we have by Lemma 6.4 Property 1 and idempotency

$$\phi(\mathbf{x}) \times \phi^{\downarrow t}(\mathbf{x}^{\downarrow t}) \;=\; \phi(\mathbf{x}) \times \sum_{\mathbf{y} \in \Omega_{s-t}} \phi(\mathbf{x}^{\downarrow t}, \mathbf{y}) \;\geq_{id}\; \phi(\mathbf{x}) \times \phi(\mathbf{x}) \;=\; \phi(\mathbf{x}).$$

Thus, $\phi(\mathbf{x}) \times \phi^{\downarrow t}(\mathbf{x}^{\downarrow t}) = \phi(\mathbf{x})$ which proves idempotency. □

Idempotency directly implies the regularity of its multiplicative semigroup. Additionally, all groups $[a]$ consist of a unique element as we learned in Section 2.6.3. Since for all $a, b \in A$ we have

$$\begin{aligned} a \times (a + b) \;&=\; (a \times a) + (a \times b) \;=\; a + (a \times b) \\ &=\; a \times (1 + b) \;=\; a \times 1 \;=\; a, \end{aligned}$$

$[a] \leq [a + b]$ must hold. Consequently, idempotent semirings are regular. Furthermore, addition is also idempotent in a c-semiring with idempotent multiplication, since

$$a = a \times 1 = a \times (a + 1) = (a \times a) + (a \times 1) = a + a.$$

Thus, such semirings are positive and posses a unit element, which leads to the following important complement.

Lemma 6.17. *Let (Φ, D) be a valuation algebra induced by a c-semiring with idempotent multiplication. Then (Φ, D) is an information algebra.*

This closes the algebraic survey of semiring valuation algebras.

6.7 Semiring Valuation Algebra Instances

The listing of semirings given in Section 6.2 served to exemplify the semiring concepts introduced beforehand. We are next going to reconsider these semirings in order to show which valuation algebras they concretely induce. We will meet familiar faces such as probability potentials or indicator functions, which motivated the study of semiring valuation algebras. In addition, a couple of new instances will be identified and, by application of the theoretical results developed in this chapter, we may directly say which properties these instances admit. This extends the valuation algebra example catalogue of Chapter 3.

6.7.1 Probability Potentials

If we consider the arithmetic semiring of non-negative real numbers as introduced in Section 6.2.1, we come across the valuation algebra of probability potentials. Moreover, the properties of this semiring confirm what we found out in Section 3.4: Probability potentials possess neutral and null elements, but the algebra is not stable because its underlying semiring is not idempotent. In addition, we remark that the arithmetic semiring of non-negative real numbers is regular. It decomposes into a union of disjoint groups $\mathbb{R}_{>0}$ and $\{0\}$, and we clearly have $[0] \leq [a]$ for all semiring elements a. Applying Theorem 6.14 we conclude that probability potentials form a regular valuation algebra.

It is worth pointing out that these considerations are very specific. If we restrict the arithmetic semiring on positive reals $\mathbb{R}_{>0}$, no null elements exist anymore. Additionally, the semiring satisfies the cancellativity property in the particular simple case that the semiring is itself a group. Another interesting situation concerning division unfolds if we take the non-negative integers $\mathbb{N} \cup \{0\}$. This semiring is neither cancellative nor regular, but it is separative. It decomposes into the semigroups $\{0\}$ and $\mathbb{N}$. The first is already a trivial group, whereas $\mathbb{N}$ is embedded into the group of positive rational numbers. On this note, the semiring view affords a whole family of valuation algebra instances which are closely related to probability potentials.

6.7.2 Constraint Systems

The Boolean semiring of Section 6.2.6 induces the valuation algebras of indicator functions or relations. Indeed, since the Boolean semiring is a c-semiring with idempotent multiplication, we confirm using Lemma 6.17 that the induced valuation algebra is an information algebra. It is fairly common to interpret such valuations as *crisp constraints*. We say that a configuration $\mathbf{x} \in \Omega_s$ satisfies the constraint $\phi \in \Phi_s$ if $\phi(\mathbf{x}) = 1$. This is also accordant with the formalism of indicator functions discussed in Section 3.2.

Alternative constraint systems may be derived by examining other semirings. The tropical semiring of Section 6.2.3 induces the valuation algebra of *weighted constraints*. This valuation algebra is stable and possesses null elements which

follows from the properties of the tropical semiring. Since semiring multiplication corresponds to addition, we always have that $a \times b = a \times c$ implies $b = c$. In other words, tropical semirings are cancellative. To any pair of numbers $a, b \in A$ we assign the difference $a - b$ which is not necessarily in the semiring anymore. Thus, if we consider a tropical semiring on non-negative integers, it is embedded into the additive group of all integers. If on the other hand the semiring includes all integers or reals, it is already itself a group under addition. Finally, weighted constraints are also scalable. Following (2.44), the scale of a constraint $c \in \Phi_s$ is given by

$$c^{\downarrow}(\mathbf{x}) \;=\; c(\mathbf{x}) + \left(c^{\downarrow\emptyset}\right)^{-1}(\diamond) \;=\; c(\mathbf{x}) - \min_{\mathbf{y}\in\Omega_s} c(\mathbf{y}). \tag{6.25}$$

In a similar way, we can use the truncation semiring from Section 6.2.4 for the derivation of weighted constraints. Another important constraint system is formed by *set constraints* which are obtained from the subset lattice of Section 6.2.7. As we have seen, this is a distributive c-semiring such that its induced valuation algebra admits the structure of an information algebra.

6.7.3 Possibility Potentials

The very popular valuation algebra of *possibility potentials* is induced by the triangular norm semirings of Section 6.2.5. These are c-semirings and consequently, the valuation algebra of possibility potentials is stable and includes null elements. Division on the other hand depends strongly on the actual t-norm. If we use the minimum t-norm for multiplication, it is clearly idempotent and we obtain an information algebra according to Lemma 6.17. The product t-norm in turn induces a regular valuation algebra as we have seen in Section 6.7.1. Finally, the Lukasiewicz t-norm and the drastic product are not even separative. Scaling is again an important point but since the product t-norm is the only one that induces a valuation algebra with a (non-trivial) division, we restrict our attention to this particular case. Then, the scale of the possibility potential $p \in \Phi_s$ is

$$p^{\downarrow}(\mathbf{x}) \;=\; p(\mathbf{x}) \cdot \left(p^{\downarrow\emptyset}\right)^{-1}(\diamond) \;=\; \frac{p(\mathbf{x})}{\max_{\mathbf{y}\in\Omega_s} p(\mathbf{y})}. \tag{6.26}$$

Historically, possibility theory was proposed by (Zadeh, 1978) as an alternative approach to probability theory, and (Shenoy, 1992a) furnished the explicit proof that this formalism indeed satisfies the valuation algebra axioms. This was limited to only some specific t-norms and (Kohlas, 2003) generalized the proof to arbitrary t-norms. We also refer to (Schiex, 1992) which unified this and the foregoing subsection to *possibilistic constraints*.

6.8 Projection Problems with Semiring Valuations

Beyond the capability to identify new valuation algebra instances, the semiring view also permits to detect concealed projection problems in many different fields of application. To outline this ability, we first write the definition of the projection problem

in terms of semiring valuation algebras. Assume a knowledgebase $\{\phi_1, \ldots, \phi_n\}$ of semiring valuations and a query $x \subseteq s_1 \cup \ldots \cup s_n = s$ with $s_i = d(\phi_i)$. Then, the projection problem of Definition 4.1 consists in computing

$$\phi^{\downarrow x}(\mathbf{x}) \quad = \quad \sum_{\mathbf{z} \in \Omega_{s-x}} \phi(\mathbf{x}, \mathbf{z}), \tag{6.27}$$

where ϕ is given for all $\mathbf{y} \in \Omega_s$ by

$$\phi(\mathbf{y}) \quad = \quad \phi_1(\mathbf{y}^{\downarrow s_1}) \times \cdots \times \phi_n(\mathbf{y}^{\downarrow s_n}). \tag{6.28}$$

This generic formula appears in at first glance very different contexts. Two examples from *signal processing* will be given right here and we will resume this discussion in Section 8.1.

6.8.1 Hadamard Transform

The *Hadamard transform* is an essential part of many applications that incorporate error correcting codes, random number generation, simulation of quantum computers or image and video compression. The (unnormalized) Hadamard transform uses a $2^m \times 2^m$ Hadamard matrix H_m that is defined recursively by the so-called *Sylvester construction*:

$$H_1 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \qquad H_m = \begin{bmatrix} H_{m-1} & H_{m-1} \\ H_{m-1} & -H_{m-1} \end{bmatrix}.$$

For instance, we obtain for $m = 3$,

$$H_3 \quad = \quad \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\ 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \end{pmatrix}.$$

Hadamard matrices have many interesting properties. They are symmetric, orthogonal, self–inverting up to a constant and any two rows differ in exactly 2^{m-1} positions. The columns of a Hadamard matrix are called *Walsh functions*. Thus, if we multiply a real-valued vector with a Hadamard matrix, we obtain a decomposition of this vector into a superposition of Walsh functions. This constitutes the Hadamard transform. For illustration, assume three binary variables X_1 to X_3 and a function $f : \{X_1, X_2, X_3\} \to \mathbb{R}$. This function is completely determined by its values v_i for $i = 1, \ldots, 8$. Then, we obtain for its Hadamard transform,

$$H_3 \cdot \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \\ v_8 \end{pmatrix} = \begin{pmatrix} v_1 + v_2 + v_3 + v_4 + v_5 + v_6 + v_7 + v_8 \\ v_1 - v_2 + v_3 - v_4 + v_5 - v_6 + v_7 - v_8 \\ v_1 + v_2 - v_3 - v_4 + v_5 + v_6 - v_7 - v_8 \\ v_1 - v_2 - v_3 + v_4 + v_5 - v_6 - v_7 + v_8 \\ v_1 + v_2 + v_3 + v_4 - v_5 - v_6 - v_7 - v_8 \\ v_1 - v_2 + v_3 - v_4 - v_5 + v_6 - v_7 + v_8 \\ v_1 + v_2 - v_3 - v_4 - v_5 - v_6 + v_7 + v_8 \\ v_1 - v_2 - v_3 + v_4 - v_5 + v_6 + v_7 - v_8 \end{pmatrix}.$$

An alternative way to compute this Hadamard transform bears on a duplication of variables. We introduce Y_1 to Y_3 which may intuitively be seen as a binary representation of the line number in the Hadamard matrix. Then, the same computational task is expressed by the following formula where $x_i, y_i \in \{0, 1\}$,

$$\sum_{x_1,x_2,x_3} f(x_1, x_2, x_3) \cdot (-1)^{x_1 y_1} \cdot (-1)^{x_2 y_2} \cdot (-1)^{x_3 y_3}.$$

Comparing this formula with Equation (6.27) makes apparent that it corresponds indeed to a projection problem over an arithmetic semiring with query $\{Y_1, Y_2, Y_3\}$ and factor domains $\{X_1, X_2, X_3\}$, $\{X_1, Y_1\}$, $\{X_2, Y_2\}$ and $\{X_3, Y_3\}$. Consequently, Hadamard transforms can be computed by application of the collect algorithm on a join tree with root node $\{Y_1, Y_2, Y_3\}$. This leads to the *fast Hadamard transform* as explained in (Aji & McEliece, 2000).

6.8.2 Discrete Fourier Transform

Very similar to the Hadamard transform, the *discrete Fourier transform* also shapes up as a projection problem over semiring valuations. The following conversion is taken from (Aji, 1999). Assume a positive integer N and a function $f : \mathbb{Z}_N \to \mathbb{C}$. The discrete Fourier transform F of f is then given as:

$$F(y) = \sum_{x=0}^{N-1} f(x) e^{-\frac{2i\pi xy}{N}}. \tag{6.29}$$

Now, take $N = 2^m$ and write x and y in binary representation:

$$x = \sum_{k=0}^{m-1} x_k 2^k \quad \text{and} \quad y = \sum_{l=0}^{m-1} y_l 2^l$$

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

$$xy = \sum_{0 \le k,l \le m-1} x_k y_l 2^{k+l}.$$

Using this encoding in the above Fourier transform gives then

$$\begin{aligned} F(y_0, \ldots, y_{m-1}) &= \sum_{x_0,\ldots,x_{m-1}} f(x_0 \ldots, x_{m-1}) \prod_{0 \le k,l \le m-1} e^{-\frac{2i\pi x_k y_l}{2^{m-k-l}}} \\ &= \sum_{x_0,\ldots,x_{m-1}} f(x_0 \ldots, x_{m-1}) \prod_{0 \le k,l \le m-1} e^{-\frac{i\pi x_k y_l}{2^{m-k-l-1}}} \end{aligned}$$

We can easily convince ourselves that this formula corresponds to a projection problem over the complex, arithmetic semiring.

6.9 Conclusion

This chapter identified a large family of valuation algebras that share a very specific common structure relying on a mapping from configurations to semiring values. The developed theory connects the properties of the induced valuation algebras with the attributes of the underlying semiring. In a nutshell, Figure 6.4 summarizes the most important semiring attributes with the special focus on their inheritance structure. Then, the following relationships between semiring attributes and valuation algebra properties exist:

- Semirings with unit element induce valuation algebras with neutral elements.
- Idempotent semirings induce stable valuation algebras.
- Positive semirings induce valuation algebras with null elements.
- Separative or cancellative semirings induce separative valuation algebras.
- Regular semirings induce regular valuation algebras.
- C-semirings with idempotent multiplication induce information algebras.

The profit of this theory is substantial as we obtain a possibly new valuation algebra instance from every semiring, and due to the above relationships, no further effort for the algebraic analysis of these formalisms is required. This was perfectly illustrated in Section 6.7, where it was shown how already known as well as completely new valuation algebras are derived. Furthermore, the particular shape of projection problems with semiring-based knowledgebase factors may also be used to track down possibly unaware fields of applicability for local computation. To complete this chapter, we reconsider the instance property map drawn in Section 3.9 and extend it by the new formalisms identified with the aid of the semiring framework. This leads to Figure 6.5.

The semiring valuation algebra framework is simultaneously very general because it covers a large number of instances but also very restrictive concerning their buildup. All valuations are simple mappings of configurations to semiring values, but it is exactly this restrictive form which also entails a new field of study that looks

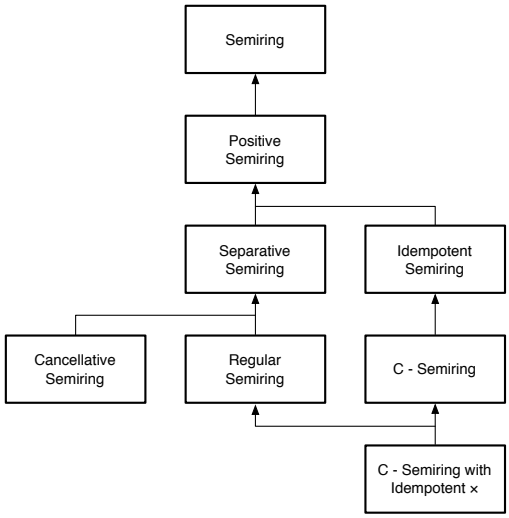


Figure 6.4: The semiring framework.

for efficient ways of representing or storing generic semiring valuations. Clearly, the most proximate representation is the tabular form we used for indicator functions or probability potentials in Chapter 3. A considerably better way to deal with the exponential size of semiring valuations is proposed by (Wachter *et al.*, 2007) and exploits a property that is similar to *context-specific independence* from Bayesian network theory. Thereby, configurations which are mapped to the same semiring value are grouped using a Boolean function which in turn is represented in a suitable way to ensure its fast evaluation.

	Neutral Elements	**Stability**	**Null Elements**	**Separativity**	**Regularity**	**Idempotency**	**Scalability**
Indicator Functions	√	√	√	√	√	√	√
Relations	√	√	√	√	√	√	√
Probability Potentials	√	○	√	√	√	○	√
Belief Functions	√	√	√	√	○	○	√
Distance Functions	√	√	○	√	√	√	√
Densities	○	○	√	√	○	○	√
Gaussian Potentials	○	○	○	√	○	○	√
Clause Sets	○	○	○	√	√	√	√
Weighted Constraints	√	√	√	√	○	○	√
Set Constraints	√	√	√	√	√	√	√
Possibility Potentials	√	√	√	depends on t-norm			

Figure 6.5: Extended instance property map from Section 3.9.

7

Set-Based Semiring Valuation Algebras

The family of semiring valuation algebras is certainly extensive, but there are nevertheless important formalisms which do not admit this particular structure. Some of them have already been mentioned in Chapter 3 such as distance functions, densities or set potentials. The last are of particular interest in this context. Set potentials map configuration sets on non-negative real numbers, whereas both valuation algebra operations reduce to addition and multiplication. This is remarkably close to the buildup of semiring valuation algebras, if we envisage a generalization from configurations to configuration sets. In doing so, we hit upon a second family of valuation algebra instances which covers such important formalisms as set potentials or mass functions. As far as we know, this generalization has never been considered before in the literature and there may be good reasons for this. Set potentials are in fact the only known formalism of practical importance that adopts this structure. On the other hand, this theory again produces a multiplicity of new valuation algebra instances, although we are not yet acquainted with their exact field of application. Consequently, we confine ourselves to a short treatment of set-based semiring valuations, leaving out a proper inspection of division as performed in previous chapters.

7.1 Set-Based Semiring Valuations

Let us again consider an arbitrary semiring $\mathcal{A} = \langle A, +, \times \rangle$ and a set r of finite variables. A *set-based semiring valuation* ϕ with domain $s \subseteq r$ is defined to be a function that associates a value from A with each configuration subset of Ω_s,

$$\phi : \mathcal{P}(\Omega_s) \to A. \tag{7.1}$$

The set of all set-based semiring valuations with domain s will subsequently be denoted by Φ_s and we use Φ for all set-based semiring valuations whose domain belongs to the lattice $D = \mathcal{P}(r)$. Next, the following operations are introduced:

1. *Labeling:* $\Phi \to D$: $d(\phi) = s$ if $\phi \in \Phi_s$.

2. *Combination:* $\Phi \times \Phi \to \Phi$: for $\phi, \psi \in \Phi$ and $A \subseteq \Omega_{d(\phi)\cup d(\psi)}$ we define

$$(\phi \otimes \psi)(A) \;=\; \sum_{B^{\uparrow s\cup t}\cap C^{\uparrow s\cup t}=A} \phi(B) \times \psi(C). \tag{7.2}$$

3. *Marginalization:* $\Phi \times D \to \Phi$: for $\phi \in \Phi$, $t \subseteq d(\phi)$ and $A \subseteq \Omega_t$ we define

$$\phi^{\downarrow t}(A) \;=\; \sum_{B^{\downarrow t}=A} \phi(B). \tag{7.3}$$

For our purposes, it is a lot more convenient to rewrite these definitions in the language of relations introduced in Sections 3.1 and 3.3. We observe that $B^{\uparrow s\cup t} \cap C^{\uparrow s\cup t} = B \bowtie C$ and $B^{\downarrow t} = \pi_t(B)$. Then, since we already know that relations form a valuation algebra, their properties can be used to prove the following main theorem:

Theorem 7.1. *A system of set-based semiring valuations (Φ, D) with labeling, combination and marginalization as defined above satisfies the valuation algebra axioms.*

Proof. We verify the valuation algebra axioms given in Section 2.1. At first glance, we see that the labeling axiom (A2) and the marginalization axiom (A3) are immediate consequences of the above definitions.

(A1) *Commutative Semigroup:* Commutativity of combination follows directly from the commutativity of semiring multiplication and natural join. For $\phi \in \Phi_s, \psi \in \Phi_t$ and $A \subseteq \Omega_{s\cup t}$ we have

$$(\phi \otimes \psi)(A) \;=\; \sum_{B\bowtie C=A} \phi(B) \times \psi(C) \;=\; \sum_{C\bowtie B=A} \phi(C) \times \psi(B) \;=\; (\psi \otimes \phi)(A).$$

Next, we derive associativity assuming that $\nu \in \Phi_u$ and $A \subseteq \Omega_{s\cup t\cup u}$

$$\begin{aligned}
(\phi \otimes (\psi \otimes \nu))(A) &= \sum_{B\bowtie E=A} \phi(B) \times (\psi \otimes \nu)(E) \\
&= \sum_{B\bowtie E=A} \phi(B) \times \sum_{C\bowtie D=E} \psi(C) \times \nu(D) \\
&= \sum_{B\bowtie E=A} \sum_{C\bowtie D=E} \phi(B) \times \psi(C) \times \nu(D) \\
&= \sum_{B\bowtie C\bowtie D=A} \phi(B) \times \psi(C) \times \nu(D)
\end{aligned}$$

The same result is obtained in exactly the same way for $(\phi\otimes(\psi\otimes\nu))(A)$ which proves associativity of combination.

(A4) *Transitivity:* For $\phi \in \Phi$ with $s \subseteq t \subseteq d(\phi)$ we have

$$\begin{aligned}
(\phi^{\downarrow t})^{\downarrow s}(A) &= \sum_{\pi_s(B)=A} \phi^{\downarrow t}(B) \;=\; \sum_{\pi_s(B)=A} \sum_{\pi_t(C)=B} \phi(C) \;=\; \sum_{\pi_s(\pi_t(C))=A} \phi(C) \\
&= \sum_{\pi_s(C)=A} \phi(C) \;=\; \phi^{\downarrow s}(A).
\end{aligned}$$

Observe that we used the transitivity of projection with holds since relations form a valuation algebra themselves.

(A5) *Combination:* Suppose that $\phi \in \Phi_s$, $\psi \in \Phi_t$ and $A \subseteq \Omega_z$, where $s \subseteq z \subseteq s \cup t$. Then, using the combination property of the valuation algebra of relations we obtain

$$\begin{aligned}(\phi \otimes \psi)^{\downarrow z}(A) &= \sum_{\pi_z(B)=A} \phi \otimes \psi(B) = \sum_{\pi_z(B)=A} \sum_{C \bowtie D = B} \phi(C) \times \psi(D) \\ &= \sum_{\pi_z(C \bowtie D)=A} \phi(C) \times \psi(D) = \sum_{C \bowtie \pi_{t \cap z}(D)=A} \phi(C) \times \psi(D) \\ &= \sum_{C \bowtie E = A} \phi(C) \times \sum_{\pi_{t\cap z}(D)=E} \psi(D) = \sum_{C \bowtie E = A} \phi(C) \times \psi^{\downarrow t \cap z}(E) \\ &= \phi \otimes \psi^{\downarrow t \cap z}(A).\end{aligned}$$

(A6) *Domain:* For $\phi \in \Phi$ and $x = d(\phi)$ we have

$$\phi^{\downarrow x}(A) = \sum_{B=A} \phi(B) = \phi(A).$$

□

Giving a first summary, semirings possess enough structure to afford this second family of valuation algebra instances. We are next going to investigate the necessary requirements for the underlying semiring to guarantee neutral and null elements in the induced valuation algebra.

7.2 Neutral Set-Based Semiring Valuations

If the semiring $\mathcal{A}$ has a unit element, then we have for every domain s a valuation e_s such that $\phi \otimes e_s = e_s \otimes \phi = \phi$ for all $\phi \in \Phi_s$. This element e_s is defined as follows:

$$e_s(A) = \begin{cases} \mathbf{1}, & \text{if } A = \Omega_s, \\ \mathbf{0}, & \text{otherwise.} \end{cases} \tag{7.4}$$

Indeed, it holds for $\phi \in \Phi_s$ that

$$\phi \otimes e_s(A) = \sum_{B \bowtie C = A} \phi(B) \times e_s(C) = \phi(A) \times e(\Omega_s) = \phi(A) \times \mathbf{1} = \phi(A).$$

These neutral elements satisfy property (A7). For this proof we gain certainty that $A \bowtie B = \Omega_{s\cup t}$ if, and only if, $A = \Omega_s$ and $B = \Omega_t$.

(A7) *Neutrality:* On one hand we have

$$e_s \otimes e_t(\Omega_{s\cup t}) = \sum_{A \bowtie B = \Omega_{s\cup t}} e_s(A) \times e_t(B) = e_s(\Omega_s) \times e_t(\Omega_t) = \mathbf{1}.$$

On the other hand, if $A \bowtie B \subset \Omega_{s\cup t}$, then either $A \subset \Omega_s$ or $B \subset \Omega_t$. So, at least one factor corresponds to the zero element of the semiring and therefore $e_s \otimes e_t(C) = \mathbf{0}$ for all $C \subset \Omega_{s\cup t}$.

Set-based semiring valuation algebras with neutral elements are always stable.

(A8) *Stability:* On the one hand we have

$$e_s^{\downarrow t}(\Omega_t) \;=\; \sum_{\pi_t(A)=\Omega_t} e_s(A) \;=\; e_s(\Omega_s) \;=\; \mathbf{1}.$$

The second equality holds because $e_s(\Omega_s) = \mathbf{1}$ is the only non-zero term within this sum. On the other hand, $e_s^{\downarrow t}(A) = \mathbf{0}$ for all $A \subset \Omega_t$ because $e_s(\Omega_s)$ does not occur.

7.3 Null Set-Based Semiring Valuations

Because all semirings are assumed to contain a zero element, we have for every domain $s \in D$ a valuation z_s such that $\phi \otimes z_s = z_s \otimes \phi = z_s$. This element is defined as $z_s(A) = \mathbf{0}$ for all $A \subseteq \Omega_s$. Indeed, we have for $\phi \in \Phi_s$,

$$\phi \otimes z_s(A) \;=\; \sum_{B \bowtie C = A} \phi(B) \times z_s(C) \;=\; \mathbf{0}.$$

These candidates z_s must satisfy the nullity axiom which requires two properties. The first condition that null elements project to null elements is clearly satisfied. More involved is the second condition that only null elements project to null elements. Positivity is again a sufficient condition.

(A9) *Nullity:* For $\phi \in \Phi_s$ and $t \subseteq s$, we have

$$\mathbf{0} \;=\; \phi^{\downarrow t}(A) \;=\; \sum_{\pi_t(B)=A} \phi(B)$$

implies that $\phi(B) = \mathbf{0}$ for all B with $\pi_t(B) = A$. Hence, $\phi = z_s$.

7.4 Set-Based Semiring Valuation Algebra Instances

It was pointed out in the introduction of this chapter that our knowledge about practically meaningful set-based semiring valuations is very modest. However, regardless of this estimation, two examples have to be mentioned.

7.4.1 Set Potentials & Mass Functions

Remember that the formalism of set potentials was the guiding theme for the development of set-based semiring valuations, and our investigations in this chapter confirm what we have already found out about this formalism in Section 3.5. Namely, set potentials (and their normalized version called mass functions) have neutral elements, are stable and possess null elements.

7.4.2 Possibility Functions

(Zadeh, 1979) originally introduced *possibility functions* compatible to our framework of set-based semiring valuations over a (particular) triangular norm semiring. But it turned out that possibility functions are completely specified by their values assigned to singleton configuration sets. Based on this insight, (Shenoy, 1992a) derived the valuation algebra of possibility potentials discussed in Section 6.7.3. Working with possibility functions is therefore unusual but we can deduce from this chapter that they nevertheless form a valuation algebra themselves.

7.5 Conclusion

A question unanswered up to now concerns the relationship between traditional semiring valuations and set-based semiring valuations. On the one hand, semiring valuations might be seen as special cases of set-based semiring valuations where only singleton configuration sets are allowed to have non-zero values. However, we nevertheless shrink away from saying that set-based semiring valuations include the family of traditional semiring valuations. The main reason for this is the inconsistency of the definition of neutral elements in both formalisms. This is also shown by the fact that a semiring valuation algebra requires an idempotent semiring for stability, whereas all set-based semiring valuation algebras with neutral elements are naturally stable. A possible way to avoid this problem is to propose an alternative definition of neutral elements for set-based semiring valuations, but then we loose conformity with neutral elements in Dempster-Shafer theory (see Sections 3.5 and 7.4.1). Altogether, we prefer to consider the two semiring-related formalisms as disjoint subfamilies of valuation algebra instances.

Part III

Dynamic Programming

8
Optimization

After this short intermezzo discussing set-based semiring valuation algebras, we return to ordinary semiring valuations with a special focus on those instances that are based on totally ordered semirings with idempotent addition. Then, the very particular projection problem with empty query turns into an optimization task under these requirements. To be more concrete, this amounts to the computation of either the maximum or minimum semiring value of the objective function, which is given as knowledgebase factorization. The best example of such an application is constraint satisfaction, where a crisp constraint ϕ is satisfiable exactly if its maximum value is $\phi(\mathbf{x}) = 1$ for some configuration $\mathbf{x}$. Evidently, with the collect algorithm developed in Section 4.4.1, we have already the method needed for the efficient solution of such optimization tasks. Moreover, if we consider variable elimination in place of marginalization, the collect algorithm performs a step-wise and consecutive optimization for every set of variables eliminated between two join tree nodes. This particular procedure to tackle optimization problems is generally known as *dynamic programming* (Bertele & Brioschi, 1972) and can be found in almost every handbook about algorithms and programming techniques. In a pioneering work, (Shenoy, 1996) established the close relationship between valuation algebras and dynamic programming, and he furthermore proved that the axioms for discrete dynamic programming given in (Mitten, 1964) entail those of the valuation algebra. Hence, the axiomatic system of a valuation algebra that enables local computation also constitutes the mathematical permission to apply dynamic programming.

However, the interest in optimization problems raises another question that is of prime importance. In most cases, we are not only looking for some optimum value, but we then ask for either one single or all configurations that adopt the computed value. Such configurations will be referred to as *solution configurations*. In terms of crisp constraints, these are the configurations that fulfill the given constraint, but the problem also covers other important applications such as the identification of most and least probable configurations in a Bayesian network. Remember that the local computation algorithms we have mentioned do not aim at the search for configurations and a simple *table lookup procedure* in the objective function ϕ is out of question due to the computational intractability of building ϕ. Therefore, (Shenoy,

1996) proposed a new local computation technique that allows to find single solution configurations as an additional process following the fusion algorithm (Shenoy, 1992b). In this chapter, we introduce a generalization of this technique to covering join trees so that collect instead of fusion can be used. Then, by imposing a further restriction on the underlying semiring, we extend this algorithm so that all solution configurations are found by essentially the same procedure. This is the first class of algorithms studied in this chapter.

A different approach for the same task leads to a second class of algorithms which aim at a more compact representation of the solution configuration set. Substantially, these methods can be summarized as the construction of a Boolean function (respectively a representation thereof) whose models correspond to the solution configurations we are looking for. This will be done in parallel to the collect algorithm such that the latter will act as a *compilation procedure* for solution configurations. Naturally, this approach is only interesting if the Boolean function allows to extract its models efficiently. Therefore, the construction rules for this compilation are defined in such a way that the resulting Boolean function becomes representable by a very particular *propositional directed acyclic graph* (PDAG) called d-DNNF (Darwiche & Marquis, 2002; Wachter & Haenni, 2006), which provably has the property of efficient model enumeration. But d-DNNF structures are also highly suitable for other important queries than model enumeration, and this leads to a multiplicity of new applications. To mention just a few examples, we may perform probabilistic equivalence checks of objective functions or compute the probabilities of solution configurations given a prior distribution for every involved variable. Altogether, we point out many new fields of application where this alliance of local computation and knowledge representation becomes a very helpful tool.

This chapter is organized in the following way: We first give a formal definition of an optimization problem and illustrate the importance of this concept by a collection of typical applications. Then, we introduce the notion of solution configurations in Section 8.2, whose identification will take center stage in the rest of this chapter. The first class of algorithms for this purpose aim for a direct computation of solution configurations using local computation techniques. This is the content of Section 8.3. The second class of algorithms that provide a compilation of solution configurations into a Boolean function is developed in Section 8.4 and brings this chapter to a close.

8.1 Optimization Problems

We learned in Section 4.2 that projection problems precisely state the computational interest in valuation algebras, and their efficient solution was the ambition for the development of all local computation algorithms in Chapter 4. We will next see that projection problems acquire a very particular meaning when dealing with valuation

algebras which are induced by semirings with idempotent addition. Then, we have for a projection problem with query $t \subseteq d(\phi) = s$ and $\mathbf{x} \in \Omega_t$,

$$\phi^{\downarrow t}(\mathbf{x}) \;=\; \sum_{\mathbf{y}\in\Omega_{s-t}} \phi(\mathbf{x},\mathbf{y}) \;=\; \sup\{\phi(\mathbf{x},\mathbf{y}),\ \mathbf{y}\in\Omega_{s-t}\}, \tag{8.1}$$

if $\phi = \phi_1 \otimes \ldots \otimes \phi_n$ denotes the objective function. This corresponds to the computation of the lowest upper bound of all semiring values that are assigned to those configurations of ϕ that project to $\mathbf{x}$. This is a consequence of Lemma 6.2, Property 6. In particular, if we consider the query to be empty, we obtain

$$\phi^{\downarrow\emptyset}(\diamond) \;=\; \sup\{\phi(\mathbf{x}),\ \mathbf{x}\in\Omega_s\},$$

which amounts to the computation of the lowest upper bound of all semiring values within ϕ. If we furthermore assume that the underlying semiring is totally ordered, we obtain according to Lemma 6.6

$$\phi^{\downarrow t}(\mathbf{x}) \;=\; \sum_{\mathbf{y}\in\Omega_{s-t}} \phi(\mathbf{x},\mathbf{y}) \;=\; \max\{\phi(\mathbf{x},\mathbf{y}),\ \mathbf{y}\in\Omega_{s-t}\}, \tag{8.2}$$

and

$$\phi^{\downarrow\emptyset}(\diamond) \;=\; \max\{\phi(\mathbf{x}),\ \mathbf{x}\in\Omega_s\} \tag{8.3}$$

respectively. This motivates the following definition:

Definition 8.1. *A projection problem with empty query over a totally ordered, idempotent semiring is called an optimization problem.*

We subsequently survey some typical examples of optimization problems in order to convince the reader of the far reaching application field of this theory. Since optimization problems are likewise projection problems, this listing also extends the collection of projection problems given in Section 4.2.

8.1.1 Classical Optimization

Perhaps the most obvious optimization problem is obtained from the tropical semiring of Section 6.2.3, where $\times$ corresponds to the usual addition of non-negative integers and $+$ to maximization (or minimization). Accordingly, $\phi^{\downarrow\emptyset}(\diamond)$ represents the maximum (minimum) value over all configurations of the objective function with respect to the natural semiring order, i.e.

$$\phi^{\downarrow\emptyset}(\diamond) \;=\; \max_{\mathbf{x}\in\Omega_s}\left(\phi_1(\mathbf{x}^{\downarrow d(\phi_1)}) \times \ldots \times \phi_n(\mathbf{x}^{\downarrow d(\phi_n)})\right). \tag{8.4}$$

8.1.2 Satisfiability Problems

We have already seen in Section 6.7.2 that crisp and weighted constraints fit into the framework of semiring valuation algebras. Since addition corresponds to either minimization or maximization in both cases, these formalisms clearly induce

optimization problems according to Definition 8.1. To be more concrete, a crisp constraint is satisfiable if $R_\phi = \{\mathbf{x} \in \Omega_s : \phi(\mathbf{x}) = 1\}$ contains at least one configuration, and this is the case if $\phi^{\downarrow\emptyset}(\diamond) = 1$, because addition is maximization. Otherwise, if $\phi^{\downarrow\emptyset}(\diamond) = 0 = z_\emptyset(\diamond)$, the set of constraints is contradictory due to the nullity axiom. Remember that these systems also comprise the SAT problem of propositional logic.

8.1.3 Maximum Satisfiability Problems

MAX SAT (Garey & Johnson, 1990) can be regarded as an approximation task for the SAT problem of propositional logic. Here, we are interested in the maximum number of clauses that can be made true, even though the total clause set may be unsatisfied under the corresponding truth value assignment. To do so, we start as usually by representing the models of each clause of the SAT instance as semiring valuation. This time however, we consider these valuations to be induced by the tropical semiring with maximization. Then, $\phi(\mathbf{x})$ represents the number of true clauses under the truth assignment $\mathbf{x}$ and consequently, $\phi^{\downarrow\emptyset}(\diamond)$ designates the maximum number of true clauses over all possible truth value assignments.

8.1.4 Most & Least Probable Configuration

Section 6.7.1 has shown that the arithmetic semiring induces the valuation algebra of probability potentials. However, for some applications one is not directly interested in a concrete probability value but instead, it is sufficient to identify either a minimum or maximum value of all probabilities in ϕ. The maximum can for example be determined using the product t-norm semiring. Here, addition is maximization and the value $\phi^{\downarrow\emptyset}(\diamond)$ corresponds to the probability of the most probable configuration. Similarly, if we replace maximization by minimization, the marginal identifies the value of the least probable configuration.

8.1.5 Channel Decoding

The semiring consisting of the unit interval with minimization for addition and the product t-norm for multiplication can also be used for decoding purposes. Assume an unreliable, memoryless communication channel with $\mathbf{x} = (x_1, \ldots, x_n)$ denoting the unknown input of the channel and $\mathbf{y} = (y_1, \ldots, y_n)$ the observed codeword after transmission. The possible values for x_i and y_i are specified by the coding alphabet. Now, the decoding process asks to deduce the input from the received output and this can for example be done by the maximization of the probability $P(\mathbf{y}|\mathbf{x})$. We have

$$\max_{\mathbf{x}} p(\mathbf{y}|\mathbf{x}) \quad = \quad \max_{\mathbf{x}} \left(\prod_{i=1}^{n} p(y_i|x_i) \cdot p(x_i) \right),$$

where the transmission probabilities $p(y_i|x_i)$ are known from the channel specification. This approach is generally called *Bayes decoding* and looks for the input that leads most probably to the observed output. An important simplification of this

decoding scheme assumes a uniform probability distributions for the input symbols. Then, it remains to compute

$$\max_{\mathbf{x}} p(\mathbf{y}|\mathbf{x}) \quad = \quad \max_{\mathbf{x}} \left(\prod_{i=1}^{n} p(y_i|x_i) \right),$$

which is known as *maximum likelihood decoding*. Alternatively, it is often convenient for computational purposes to replace the maximization of $P(\mathbf{y}|\mathbf{x})$ by the minimization of its negated logarithm. We then obtain for the case of maximum likelihood decoding

$$\min_{\mathbf{x}} \left(\log p(\mathbf{y}|\mathbf{x}) \right) \quad = \quad \min_{\mathbf{x}} \left(- \sum_{i=1}^{n} \log p(y_i|x_i) \right). \tag{8.5}$$

This is again an optimization problem but this time induced by the tropical semiring.

Linear codes go a step further and take into consideration that not all arrangements of input symbols may be valid code words. Such systems therefore provide a so-called *(low density) parity check matrix* H which assures $H \cdot \mathbf{x}^T = \mathbf{0}$ exactly if $\mathbf{x}$ is a valid code word. For illustration purposes, assume the following matrix taken from (Wiberg, 1996):

$$H \quad = \quad \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}.$$

It clearly follows that $\mathbf{x} = (x_1, \ldots, x_7)$ is a valid code word if

$$(x_1 + x_2 + x_4, x_3 + x_4 + x_6, x_4 + x_5 + x_7) \quad = \quad (0, 0, 0).$$

These additional constraints can be blent in the maximum likelihood decoding scheme using the following auxiliary functions:

$$\chi_1(x_1, x_2, x_4) \quad = \quad \begin{cases} 0, & \text{if } x_1 + x_2 + x_4 = 0 \\ \infty, & \text{otherwise,} \end{cases}$$

$$\chi_2(x_3, x_4, x_6) \quad = \quad \begin{cases} 0, & \text{if } x_3 + x_4 + x_6 = 0 \\ \infty, & \text{otherwise,} \end{cases}$$

$$\chi_3(x_4, x_5, x_7) \quad = \quad \begin{cases} 0, & \text{if } x_4 + x_5 + x_7 = 0 \\ \infty, & \text{otherwise.} \end{cases}$$

We then obtain for Equation (8.5) and $n = 7$,

$$\min_{\mathbf{x}} \left(\sum_{i=1}^{7} - \log p(y_i|x_i) + \chi_1(x_1, x_2, x_4) + \chi_2(x_3, x_4, x_6) + \chi_3(x_4, x_5, x_7) \right).$$

It remains an optimization problem with knowledgebase factors induced by the tropical semiring. Furthermore, applying the collect algorithm from Section 4.4.1 to this setting yields the *Gallager-Tanner-Wiberg algorithm* (Gallager, 1963) as remarked by (Aji & McEliece, 2000). A recommendable survey of these and related decoding schemes is given in (MacKay, 2003) where the author presents the corresponding algorithms as message-passing systems. This suggests that even more sophisticated and state of the art decoding systems such as *convolutional* or *turbo codes* are covered by optimization problems over semiring valuation algebras.

8.2 Solution Configurations & Solution Extensions

Optimization problems are special cases of single-query projection problems and can therefore be solved efficiently using the collect algorithm introduced in Section 4.4.1. Doing so, the explicit and often intractable computation of the objective function ϕ is nifty omitted. However, as motivated in the introduction of this chapter, we are in many cases not only interested in some optimum value, but it is in fact required to identify the configurations that adopt the computed value. Following (Shenoy, 1996), we refer to such configurations as *solution configurations* for the objective function ϕ.

Definition 8.2. *Let (Φ, D) be a valuation algebra over a totally ordered, idempotent semiring. For $\phi \in \Phi_s$, we call $\mathbf{x} \in \Omega_s$ a solution configuration if $\phi(\mathbf{x}) = \phi^{\downarrow\emptyset}(\diamond)$.*

The set of all solution configurations for a given valuation ϕ is called *solution configuration set* and defined as:

$$c_\phi \quad = \quad \{\mathbf{x} \in \Omega_s : \phi(\mathbf{x}) = \phi^{\downarrow\emptyset}(\diamond)\}. \tag{8.6}$$

In the further process, we will often deal with *partial solution configuration sets* $c_\phi^{\downarrow t}$ for some $t \subseteq s$. We therefore remark that

$$\begin{aligned} c_\phi^{\downarrow t} &= \{\mathbf{x} \in \Omega_s : \phi(\mathbf{x}) = \phi^{\downarrow\emptyset}(\diamond)\}^{\downarrow t} \\ &= \{\mathbf{y} \in \Omega_t : \phi^{\downarrow t}(\mathbf{y}) = \phi^{\downarrow\emptyset}(\diamond)\} \;=\; c_{\phi^{\downarrow t}}. \end{aligned} \tag{8.7}$$

Further, because $\mathbf{0} \leq_{id} \phi(\mathbf{x})$ for all $\mathbf{x} \in \Omega_s$ due to Lemma 6.2, the following important property holds.

Lemma 8.3. *$\phi^{\downarrow\emptyset}(\diamond) = \mathbf{0}$ implies that $c_\phi = \Omega_s$.*

The existence of at least one solution configuration is an immediate consequence of the total order and the finiteness of Ω_s. It therefore holds that $c_\phi \neq \emptyset$ for every $\phi \in \Phi$. This in turn implies that we can always find a *configuration extension* $\mathbf{c} \in \Omega_{s-t}$ such that $\phi(\mathbf{x}, \mathbf{c}) = \phi^{\downarrow t}(\mathbf{x})$ for all $t \subseteq s$ and $\mathbf{x} \in \Omega_t$.

Definition 8.4. *Let (Φ, D) be a valuation algebra over a totally ordered, idempotent semiring. For $\phi \in \Phi_s$, $t \subseteq s$ and $\mathbf{x} \in \Omega_t$ we define*

$$W_\phi^t(\mathbf{x}) \quad = \quad \{\mathbf{c} \in \Omega_{s-t} : \phi(\mathbf{x}, \mathbf{c}) = \phi^{\downarrow t}(\mathbf{x})\}. \tag{8.8}$$

It is very important to remark the close relationship between this definition and the solution configuration set c_ϕ. For the particular case where $t = \emptyset$ and therefore $\mathbf{x} = \diamond$, we have

$$W_\phi^\emptyset(\diamond) \;=\; c_\phi. \tag{8.9}$$

Similarly, if $t = d(\phi)$,

$$W_\phi^t(\mathbf{x}) \;=\; \{\diamond\} \tag{8.10}$$

for all $\mathbf{x} \in \Omega_t$. Note that we consider here the configuration extension of solution configurations which will henceforth be called *solution extension*. These results are generalized for any $t \subseteq d(\phi)$ in the following lemma, which follows directly from Definition 8.4.

Lemma 8.5. *For $\phi \in \Phi$ and $t \subseteq u \subseteq d(\phi)$ we have*

$$c_\phi^{\downarrow u} \;=\; \left\{(\mathbf{x},\mathbf{y}) \;:\; \mathbf{x} \in c_\phi^{\downarrow t} \text{ and } \mathbf{y} \in W_{\phi^{\downarrow u}}^t(\mathbf{x})\right\}.$$

Note that Equation (8.9) follows for $t = \emptyset$ and $u = d(\phi)$.

Lemma 8.6. *For $s, t \subseteq d(\phi)$ and $\mathbf{c} \in c_\phi^{\downarrow s}$ we have*

$$W_{\phi^{\downarrow s\cup t}}^s(\mathbf{c}) \;=\; W_{\phi^{\downarrow t}}^{s\cap t}(\mathbf{c}^{\downarrow s\cap t}).$$

Proof. Since $\mathbf{c}$ is a solution configuration for $\phi^{\downarrow s}$, it follows that for $\mathbf{x} \in W_{\phi^{\downarrow s\cup t}}^s(\mathbf{c})$

$$\phi^{\downarrow\emptyset}(\diamond) \;=\; \phi^{\downarrow s}(\mathbf{c}) \;=\; \phi^{\downarrow s\cup t}(\mathbf{c},\mathbf{x}) \;=\; \phi^{\downarrow t}(\mathbf{c}^{\downarrow s\cap t},\mathbf{x}) \;=\; \phi^{\downarrow s\cap t}(\mathbf{c}^{\downarrow s\cap t}).$$

Thus, by Definition 8.4 we have

$$\begin{aligned}
W_{\phi^{\downarrow s\cup t}}^s(\mathbf{c}) &= \left\{\mathbf{x} \in \Omega_{(s\cup t)-s} \;:\; \phi^{\downarrow s\cup t}(\mathbf{c},\mathbf{x}) = \phi^{\downarrow s}(\mathbf{c})\right\} \\
&= \left\{\mathbf{x} \in \Omega_{(s\cup t)-s} \;:\; \phi^{\downarrow t}(\mathbf{c}^{\downarrow s\cap t},\mathbf{x}) = \phi^{\downarrow s\cap t}(\mathbf{c}^{\downarrow s\cap t})\right\} \\
&= \left\{\mathbf{x} \in \Omega_{t-(s\cap t)} \;:\; \phi^{\downarrow t}(\mathbf{c}^{\downarrow s\cap t},\mathbf{x}) = \phi^{\downarrow s\cap t}(\mathbf{c}^{\downarrow s\cap t})\right\} \\
&= W_{\phi^{\downarrow t}}^{s\cap t}(\mathbf{c}^{\downarrow s\cap t}).
\end{aligned}$$

□

Finally, we derive the following theorem from Lemma 8.6 which will allow in the following section to rebuild solution configurations from partial solution configuration sets.

Theorem 8.7. *For $s, t \subseteq d(\phi)$ we have*

$$c_\phi^{\downarrow s\cup t} \;=\; \left\{(\mathbf{x},\mathbf{y}) \;:\; \mathbf{x} \in c_\phi^{\downarrow s} \text{ and } \mathbf{y} \in W_{\phi^{\downarrow t}}^{s\cap t}(\mathbf{x}^{\downarrow s\cap t})\right\}.$$

8.3 Identifying Solution Configurations

We now go about the task of identifying the solution configurations of some objective function ϕ that is given as factorization $\phi = \phi_1 \otimes \cdots \otimes \phi_n$. It was pointed out in many places that every reasonable approach for this task must dispense with the explicit computation of ϕ. Since local computation is a suitable way to avoid building the objective function, it is sensible to embark on a similar strategy. This section presents a first class of methods that start computing partial solution configuration sets whose elements are afterwards agglutinated to obtain c_ϕ.

8.3.1 Identifying all Solution Configurations with Distribute

The most obvious approach to identify all solution configurations of ϕ follows from the Shenoy-Shafer architecture presented in Section 4.4. After a complete run of the collect and distribute algorithms, every join tree node i contains the marginal of ϕ relative to its node label $\lambda(i)$. This is the statement of Theorem 4.17. From these marginals, we can build non-deterministically all solution configurations by the following procedure. Due to Equations (8.7) and (8.9), we have for the root node m

$$c_\phi^{\downarrow\lambda(m)} \quad = \quad W^{\emptyset}_{\phi^{\downarrow\lambda(m)}}(\diamond). \tag{8.11}$$

Then, we use the following lemma for a stepwise extension of $c_\phi^{\downarrow\lambda(m)}$ to c_ϕ.

Lemma 8.8. *For $i = m-1, \ldots, 1$ we have*

$$c_\phi^{\downarrow\lambda(m)\cup\ldots\cup\lambda(i)} \quad = \quad \Big\{(\mathbf{x},\mathbf{y}) : \mathbf{x} \in c_\phi^{\downarrow\lambda(m)\cup\ldots\cup\lambda(i+1)}, \mathbf{y} \in W^{\lambda(i)\cap\lambda(ch(i))}_{\phi^{\downarrow\lambda(i)}}(\mathbf{x}^{\downarrow\lambda(i)\cap\lambda(ch(i))})\Big\}.$$

Proof. This lemma follows from Theorem 8.7 if we take $s = \lambda(m)\cup\ldots\cup\lambda(i+1)$ and $t = \lambda(i)$. From the running intersection property given in Definition 4.3 we conclude

$$\Big(\lambda(m)\cup\ldots\cup\lambda(i+1)\Big)\cap\lambda(i) \quad = \quad \lambda(i)\cap\lambda(ch(i)).$$

□

To sum it up, we compute c_ϕ by the following procedure.

Algorithm 1:

1. Execute collect and distribute on the join tree factor set $\{\psi_1, \ldots, \psi_m\}$.
2. Identify $c_\phi^{\downarrow\lambda(m)}$ by use of Equation (8.11).
3. Build c_ϕ by repeated application of Lemma 8.8.

This simple algorithm enumerates all solution configurations of ϕ and operates exclusively on factors whose size is bounded by the width of the underlying join tree. Consequently, the algorithm adopts the complexity of a local computation scheme. However, this approach is devalued from the fact, that it is always necessary to perform a complete run of the distribute algorithm. We shall therefore study in the following subsection, under which restrictions one can omit the execution of distribute.

8.3.2 Identifying some Solution Configurations without Distribute

At the end of the collect algorithm, the root node m contains the marginal $\phi^{\downarrow\lambda(m)}$. This ensures that Step 2 of Algorithm 1 can be performed even if we omit the execution of distribute. More challenging is the third step that constructs the solution configuration set c_ϕ using Lemma 8.8. Here, the computation of the solution extension set requires $\phi^{\downarrow\lambda(i)}$ for $i = 1, \ldots, m-1$, and these marginals are not available at the end of collect. Instead, we rather need some way to compute solution extensions based on the node contents $\psi_i^{(m)}$ after collect. The following lemma will be useful for this undertaking. We write ω_i for the domain of node i at completion of collect and refer to Section 4.4.1 for a formal definition of this notation.

Lemma 8.9. *For* $\mathbf{c} \in c_\phi^{\downarrow\lambda(i)\cap\lambda(ch(i))}$ *it holds that*

$$W_{\phi^{\downarrow\lambda(i)}}^{\lambda(i)\cap\lambda(ch(i))}(\mathbf{c}) \quad = \quad W_{\phi^{\downarrow\omega_i}}^{\omega_i\cap\lambda(ch(i))}(\mathbf{c}^{\downarrow\omega_i\cap\lambda(ch(i))}).$$

Proof. For $\mathbf{c} \in c_\phi^{\downarrow\lambda(i)\cap\lambda(ch(i))}$ we have $\mathbf{x} \in W_{\phi^{\downarrow\lambda(i)}}^{\lambda(i)\cap\lambda(ch(i))}(\mathbf{c})$ exactly if

$$\phi^{\downarrow\lambda(i)\cap\lambda(i)}(\mathbf{c}) \quad = \quad \phi^{\downarrow\lambda(i)}(\mathbf{c}, \mathbf{x}).$$

Since $\omega_i \subseteq \lambda(i)$, it follows that $\omega_i \cap \lambda(ch(i)) \subseteq \lambda(i) \cap \lambda(ch(i))$. Then, because $\mathbf{c}$ is a solution configuration, we have

$$\phi^{\downarrow\lambda(i)\cap\lambda(i)}(\mathbf{c}) \quad = \quad \phi^{\downarrow\omega_i\cap\lambda(i)}(\mathbf{c}^{\downarrow\omega_i\cap\lambda(ch(i))}).$$

From Lemma 4.19 we conclude that $\mathbf{x} \in \Omega_{\lambda(i)-(\lambda(i)\cap\lambda(i))} = \Omega_{\omega_i-(\omega_i\cap\lambda(i))}$ and therefore

$$\phi^{\downarrow\lambda(i)}(\mathbf{c}, \mathbf{x}) \quad = \quad \phi^{\downarrow\omega_i}(\mathbf{c}^{\downarrow\omega_i\cap\lambda(ch(i))},\ \mathbf{x}).$$

Thus, we have shown that

$$\phi^{\downarrow\omega_i\cap\lambda(i)}(\mathbf{c}^{\downarrow\omega_i\cap\lambda(ch(i))}) \quad = \quad \phi^{\downarrow\omega_i}(\mathbf{c}^{\downarrow\omega_i\cap\lambda(ch(i))},\ \mathbf{x}),$$

which is the case if, and only if, $\mathbf{x} \in W_{\phi^{\downarrow\omega_i}}^{\omega_i\cap\lambda(ch(i))}(\mathbf{c}^{\downarrow\omega_i\cap\lambda(ch(i))})$. □

Applying this result, we can replace the node labels $\lambda(i)$ in Lemma 8.8 by the corresponding node domains ω_i just after collect. Let us next focus on the following theorem:

Theorem 8.10. *Let $\phi \in \Phi_s$, $\psi \in \Phi_t$ and $s \subseteq u \subseteq s \cup t$. For $\mathbf{x} \in \Omega_u$ we have*

$$W_\psi^{u \cap t}(\mathbf{x}^{\downarrow u \cap t}) \;\subseteq\; W_{\phi \otimes \psi}^u(\mathbf{x}).$$

Proof. Assume $\mathbf{x} \in \Omega_u$ and $\mathbf{c} \in W_\psi^{u \cap t}(\mathbf{x}^{\downarrow t \cap u})$. By Definition 8.4, we have

$$\psi(\mathbf{x}^{\downarrow u \cap t}, \mathbf{c}) \;=\; \psi^{\downarrow u \cap t}(\mathbf{x}^{\downarrow u \cap t}).$$

Hence, we also have

$$\phi(\mathbf{x}^{\downarrow t}) \times \psi(\mathbf{x}^{\downarrow u \cap t}, \mathbf{c}) \;=\; \phi(\mathbf{x}^{\downarrow t}) \times \psi^{\downarrow u \cap t}(\mathbf{x}^{\downarrow u \cap t}).$$

Then, by application of the definition of combination and the combination axiom:

$$\begin{aligned}(\phi \otimes \psi)(\mathbf{x}, \mathbf{c}) \;=\; \phi(\mathbf{x}^{\downarrow t}) \times \psi(\mathbf{x}^{\downarrow u \cap t}, \mathbf{c}) \;&=\; \phi(\mathbf{x}^{\downarrow t}) \times \psi^{\downarrow u \cap t}(\mathbf{x}^{\downarrow u \cap t}) \\ &=\; \left(\phi \otimes \psi^{\downarrow u \cap t}\right)(\mathbf{x}) \;=\; (\phi \otimes \psi)^{\downarrow u}(\mathbf{x}).\end{aligned}$$

We conclude from $(\phi \otimes \psi)(\mathbf{x}, \mathbf{c}) = (\phi \otimes \psi)^{\downarrow u}(\mathbf{x})$ that $\mathbf{c} \in W_{\phi \otimes \psi}^u(\mathbf{x})$. □

The following example illustrates that indeed only inclusion holds between the two solution extension sets in Theorem 8.10.

Example 8.1. *We take the bottleneck semiring from Section 6.2.2 with* max *for* $+$ *and* min *for* $\times$*. Then, assume the two semiring valuations ϕ and ψ with domains $d(\phi) = \{A\}$ and $d(\psi) = \{A, B\}$ given below. The variable frames are $\Omega_A = \{a, \overline{a}\}$ and $\Omega_B = \{b, \overline{b}\}$.*

ϕ =

A	
a	1
$\overline{a}$	1

ψ =

A	B	
a	b	6
a	$\overline{b}$	7
$\overline{a}$	b	8
$\overline{a}$	$\overline{b}$	9

$\phi \otimes \psi$ =

A	B	
a	b	1
a	$\overline{b}$	1
$\overline{a}$	b	1
$\overline{a}$	$\overline{b}$	1

For $u = \{A\} = u \cap t$, we have

$\psi^{\downarrow u \cap t}$ =

A	
a	7
$\overline{a}$	9

$(\phi \otimes \psi)^{\downarrow u}$ =

A	
a	1
$\overline{a}$	1

Finally, we remark that

$$W_\psi^{u \cap t}(a) = W_\psi^{u \cap t}(\overline{a}) = \{(\overline{b})\} \subset W_{\phi \otimes \psi}^u(a) = W_{\phi \otimes \psi}^u(\overline{a}) = \{(b), (\overline{b})\}.$$

⊖

As we will now see, Theorem 8.10 allows indeed to compute solution configurations based on the node contents at completion of collect. For that purpose, we conclude from the distribute algorithm and by application of the transitivity and combination axioms that

$$\phi^{\downarrow \omega_i} \;=\; \left(\phi^{\downarrow \lambda(i)}\right)^{\downarrow \omega_i} \;=\; \left(\psi_i^{(m)} \otimes \mu_{ch(i) \to i}\right)^{\downarrow \omega_i} \;=\; \psi_i^{(m)} \otimes \mu_{ch(i) \to i}^{\downarrow \omega_i \cap \lambda(ch(i))}.$$

Hence, by Theorem 8.10, we find

$$\begin{aligned} W_{\phi^{\downarrow\omega_i}}^{\omega_i\cap\lambda(ch(i))}(\mathbf{c}^{\downarrow\omega_i\cap\lambda(ch(i))}) &= W_{\mu_{ch(i)\to i}^{\downarrow\omega_i\cap\lambda(ch(i))}\otimes\psi_i^{(m)}}^{\omega_i\cap\lambda(ch(i))}(\mathbf{c}^{\downarrow\omega_i\cap\lambda(ch(i))}) \\ &\supseteq W_{\psi_i^{(m)}}^{\omega_i\cap\lambda(ch(i))}(\mathbf{c}^{\downarrow\omega_i\cap\lambda(ch(i))}) \supset \emptyset. \end{aligned} \tag{8.12}$$

This means for the statement of Lemma 8.8 that a possible solution extension for $\phi^{\downarrow\lambda(i)}$ can always be found in the solution extension set relative to $\psi_i^{(m)}$ and since the latter can be computed from the node content at completion of collect, we do not depend on the execution of distribute anymore. However, the prize we pay for this gain of efficiency is that some solution configurations get lost. Consequently, this algorithm can only be used to find some solution configurations. It is in general impossible to identify c_ϕ completely.

Corollary 8.11. *For $i = m-1,\ldots,1$ we have*

$$c_\phi^{\downarrow\lambda(m)\cup\ldots\cup\lambda(i)} \supseteq \Big\{(\mathbf{x},\mathbf{y}) : \mathbf{x}\in c_\phi^{\downarrow\lambda(m)\cup\ldots\cup\lambda(i+1)},\ \mathbf{y}\in W_{\psi_i^{(m)}}^{\omega_i\cap\lambda(ch(i))}(\mathbf{c}^{\downarrow\omega_i\cap\lambda(ch(i))})\Big\}.$$

We round out this section by putting the different components together:

Algorithm 2:

- Execute collect on the join tree factor set $\{\psi_1,\ldots,\psi_m\}$.
- Compute $c_\phi^{\downarrow\lambda(m)}$ by Equation (8.11).
- Choose $\mathbf{c}^{\downarrow\lambda(m)}\in c_\phi^{\downarrow\lambda(m)}$ and proceed recursively for $i = m-1,\ldots,1$:
  - compute $w_i = W_{\psi_i^{(m)}}^{\omega_i\cap\lambda(ch(i))}(\mathbf{c}^{\downarrow\omega_i\cap\lambda(ch(i))})$,
  - choose $\mathbf{y}\in w_i$ and set $\mathbf{c}^{\downarrow\lambda(m)\cup\ldots\cup\lambda(i)} = (\mathbf{c}^{\downarrow\lambda(m)\cup\ldots\cup\lambda(i+1)},\mathbf{y})$.

Let us illustrate a complete run of Algorithm 2:

Example 8.2. *We consider binary variables A,B,C with frames $\Omega_A = \{a,\overline{a}\}$ to $\Omega_C = \{c,\overline{c}\}$. Let ψ_1,ψ_2 and ψ_3 be three join tree factors defined over the bottleneck semiring from Section 6.2.2 with domains $d(\psi_1) = \{A,B\}$, $d(\psi_2) = \{B,C\}$, $\psi_3 = \{B\}$ and the following values:*

$\psi_1 =$

A	B	
a	b	2
a	$\overline{b}$	4
$\overline{a}$	b	3
$\overline{a}$	$\overline{b}$	2

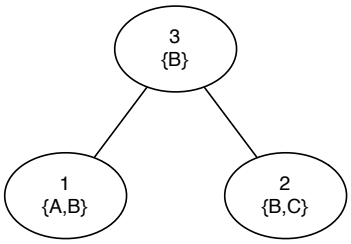
$\psi_2 =$

B	C	
b	c	5
b	$\overline{c}$	2
$\overline{b}$	c	3
$\overline{b}$	$\overline{c}$	3

$\psi_3 =$

B	
b	1
$\overline{b}$	6

The join tree below corresponds to this factorization and numbering:



Let us first compute $\phi = \psi_1 \otimes \psi_2 \otimes \psi_3$ directly to verify later results:

$\phi \;=$

A	**B**	**C**	
a	b	c	1
a	b	$\overline{c}$	1
a	$\overline{b}$	c	3
a	$\overline{b}$	$\overline{c}$	3
$\overline{a}$	b	c	1
$\overline{a}$	b	$\overline{c}$	2
$\overline{a}$	$\overline{b}$	c	2
$\overline{a}$	$\overline{b}$	$\overline{c}$	2

Observe that $c_\phi = \{(a, \overline{b}, c), (a, \overline{b}, \overline{c})\}$. We now start Algorithm 2 by executing first a complete run of collect:

$\mu_{1\to 3} =$

B	
b	3
$\overline{b}$	4

$\psi_3^{(2)} =$

B	
b	1
$\overline{b}$	4

$\mu_{2\to 3} =$

B	
b	5
$\overline{b}$	3

$\psi_3^{(3)} =$

B	
b	1
$\overline{b}$	3

Thus, the maximum value of ϕ is $\phi^{\downarrow\emptyset}(\diamond) = \psi_3^{(3)\downarrow\emptyset}(\diamond) = 3$. Next we compute

$$c_\phi^{\downarrow\{B\}} \;=\; W^{\emptyset}_{\phi^{\downarrow\{B\}}}(\diamond) \;=\; W^{\emptyset}_{\psi_3^{(3)}}(\diamond) \;=\; \{(\overline{b})\}.$$

We therefore choose $\mathbf{c}^{\downarrow\{B\}} = (\overline{b})$ and proceed for $i = 2$:

$$W^{\{B\}}_{\psi_2^{(3)}}(\overline{b}) = \{(c), (\overline{c})\},\ \textit{we choose}\ (c) \;\rightsquigarrow\; \mathbf{c}^{\downarrow\{B,C\}} = (\overline{b}, c).$$

Finally, for $i = 1$ we obtain:

$$W^{\{B\}}_{\psi_1^{(3)}}(\overline{b}) = \{(a)\} \;\rightsquigarrow\; \mathbf{c} = (a, \overline{b}, c).$$

We identified the configuration $\mathbf{c} = (a, \overline{b}, c) \in c_\phi$ with $\phi(a, \overline{b}, c) = 3$. Note that the second solution configuration of ϕ can be found by choosing the partial configuration $(\overline{c})$ in step 2. In this special case, it is possible to identify c_ϕ completely. However, as we already know, this is generally not the case. Moreover, we can not even know without computing ϕ explicitly if all solution configurations have been found or not. The user may convince himself by applying this algorithm to the factor set given in Example 8.1. $\ominus$

8.3.3 Identifying all Solution Configurations without Distribute

So far, we have seen that all solution configurations can be found if we agree on a complete run of the distribute algorithm. Without this computational effort, only some solution configurations can be identified. However, this section will show that if we impose further restrictions on the underlying semiring, all solution configurations are acquired even with the second method. For this purpose, we remark that equality can be achieved in Theorem 8.10 if the semiring is strictly monotonic.

Theorem 8.12. *Let (Φ, D) be a valuation algebra induced by a totally ordered, idempotent and strictly monotonic semiring. If $\phi \in \Phi_s$, $\psi \in \Phi_t$ and $s \subseteq u \subseteq s \cup t$, we have for all $\mathbf{x} \in \Omega_u$ with $\phi(\mathbf{x}^{\downarrow s}) \neq \mathbf{0}$*

$$W_\psi^{u\cap t}(\mathbf{x}^{\downarrow u\cap t}) \;=\; W^u_{\phi\otimes\psi}(\mathbf{x}).$$

Proof. It remains to prove that

$$W_\psi^{u\cap t}(\mathbf{x}^{\downarrow u\cap t}) \;\supseteq\; W^u_{\phi\otimes\psi}(\mathbf{x}).$$

Assume $\mathbf{x} \in \Omega_u$ with $\phi(\mathbf{x}^{\downarrow s}) \neq \mathbf{0}$ and $\mathbf{c} \in W^u_{\phi\otimes\psi}(\mathbf{x})$. By Definition 8.4,

$$(\phi \otimes \psi)^{\downarrow u}(\mathbf{x}) \;=\; (\phi \otimes \psi)(\mathbf{x}, \mathbf{c}).$$

Applying the definition of combination, we obtain

$$(\phi \otimes \psi)(\mathbf{x}, \mathbf{c}) \;=\; \phi(\mathbf{x}^{\downarrow s}) \times \psi(\mathbf{x}^{\downarrow u\cap t}, \mathbf{c}).$$

Similarly, we deduce from the combination axiom and the definition of combination

$$(\phi \otimes \psi)^{\downarrow u}(\mathbf{x}) \;=\; (\phi \otimes \psi^{\downarrow u\cap t})(\mathbf{x}) \;=\; \phi(\mathbf{x}^{\downarrow s}) \times \psi^{\downarrow u\cap t}(\mathbf{x}^{\downarrow u\cap t}).$$

Therefore,

$$\phi(\mathbf{x}^{\downarrow s}) \times \psi(\mathbf{x}^{\downarrow u\cap t}, \mathbf{c}) \;=\; \phi(\mathbf{x}^{\downarrow s}) \times \psi^{\downarrow u\cap t}(\mathbf{x}^{\downarrow u\cap t}).$$

From Lemma 6.7, we conclude that

$$\psi(\mathbf{x}^{\downarrow u\cap t}, \mathbf{c}) \;=\; \psi^{\downarrow u\cap t}(\mathbf{x}^{\downarrow u\cap t})$$

and consequently $\mathbf{c} \in W_\psi^{u\cap t}(\mathbf{x}^{\downarrow u\cap t})$. □

This result can now be used in Equation (8.12) if we assume that

$$\mu_{ch(i)\to i}^{\downarrow \omega_i\cap\lambda(ch(i))}(\mathbf{c}^{\downarrow \omega_i\cap\lambda(ch(i))}) \;\neq\; \mathbf{0}.$$

Because $\mathbf{c}$ is a solution configuration, we would have

$$\phi^{\downarrow\emptyset}(\diamond) \;=\; \phi^{\downarrow\lambda(i)}(\mathbf{c}^{\downarrow\lambda(i)}) = \mu_{ch(i)\to i}^{\downarrow \omega_i\cap\lambda(ch(i))}(\mathbf{c}^{\downarrow \omega_i\cap\lambda(ch(i))}) \times \psi_i^{(m)}(\mathbf{c}^{\downarrow\omega_i}) \;=\; \mathbf{0},$$

if this condition was not satisfied. Then, all configurations of ϕ are solution configurations due to Lemma 8.3 and the identification of the solution configuration set becomes trivial. We therefore adopt this assumption and obtain

$$W_{\phi^{\downarrow\omega_i}}^{\omega_i\cap\lambda(ch(i))}(\mathbf{c}^{\downarrow \omega_i\cap\lambda(ch(i))}) \;=\; W_{\psi_i^{(m)}}^{\omega_i\cap\lambda(ch(i))}(\mathbf{c}^{\downarrow \omega_i\cap\lambda(ch(i))}), \tag{8.13}$$

which guarantees equality in Corollary 8.11:

Corollary 8.13. *If the semiring is totally ordered, idempotent and strictly monotonic, we have for* $i = m-1, \ldots, 1$,

$$c_\phi^{\downarrow\lambda(m)\cup\ldots\cup\lambda(i)} = \left\{(\mathbf{x},\mathbf{y}) : \mathbf{x} \in c_\phi^{\downarrow\lambda(m)\cup\ldots\cup\lambda(i+1)},\ \mathbf{y} \in W_{\psi_i^{(m)}}^{\omega_i\cap\lambda(ch(i))}(\mathbf{c}^{\downarrow\omega_i\cap\lambda(ch(i))})\right\}.$$

To sum it up, the following algorithm determines c_ϕ in the case of a total order that is strictly monotonic over $\times$.

Algorithm 3:

1. Execute collect on the join tree factor set $\{\psi_1, \ldots, \psi_m\}$.
2. Identify $c_\phi^{\downarrow\lambda(m)}$ by use of Equation (8.11).
3. Build c_ϕ by repeated application of Corollary 8.13.

8.4 Compiling Solution Configurations

The last method presented in the foregoing section, that aims for the identification of all solution configurations, results in an explicit listing of all elements in c_ϕ. This is perfect for the forecited task but there are many related applications for which this enumeration is needless or even inefficient. Let us consider some concrete examples of such scenarios:

- **Solution Counting:** Determining the number of solution configurations $|c_\phi|$ is an important subtask of many applications, and it is clearly dispensable in this case to explicitly enumerate all elements in c_ϕ.
- **Countersolution Configurations:** Another example where it is unhandy to enumerate c_ϕ is the identification of configurations that are not solution configurations, i.e. the set $\overline{c}_\phi = \Omega_s - c_\phi$.
- **Validity Check:** This query concerns the question if all configurations of ϕ are solution configurations, or equivalently, if all configurations of ϕ adopt the same value. For an expected yes-no answer, the enumeration of all exponentially many solution configurations is unacceptable.

This is only a very small collection of typical applications where listing solution configurations is debilitating. Therefore, we propose for the more efficient treatment of these and related *queries* an alternative method that completely omits the explicit enumeration of configurations (Pouly *et al.*, 2007). Instead, we *compile* solution configurations into a Boolean function $f : \{0,1\}^{|s|} \to \{0,1\}$ with $s = d(\phi)$, such that they are reflected by the *models* of the latter. In order to stress this duality of solution configurations and models of Boolean functions, we restrict ourselves to propositional variables. Nevertheless, we emphasize that a generalization to arbitrary finite variables is possible (Wachter & Haenni, 2007). In order to link frame values with their corresponding variable, we denote the frame values of a variable X as $\Omega_X = \{0_X, 1_X\}$, without changing their usual interpretation as truth values.

Hence, the task of interest in this section concerns the construction of a Boolean function f over variables in s such that

$$\mathbf{Models}(f) \;=\; \{\mathbf{x} \in \Omega_s : f(\mathbf{x}) = 1\} \;=\; c_\phi. \tag{8.14}$$

Let us concretize this idea by a first example:

Example 8.3. *Consider propositional variables A, B, C and the following valuation defined over a semiring with maximization for addition:*

$\phi \;=$

A	**B**	**C**	
0_A	0_B	0_C	5
0_A	0_B	1_C	10
0_A	1_B	0_C	10
0_A	1_B	1_C	3
1_A	0_B	0_C	9
1_A	0_B	1_C	0
1_A	1_B	0_C	7
1_A	1_B	1_C	3

We have $\phi^{\downarrow\emptyset}(\diamond) = 10$ and therefore $c_\phi = \{(0_\mathrm{A}, 0_\mathrm{B}, 1_\mathrm{C}),\ (0_\mathrm{A}, 1_\mathrm{B}, 0_\mathrm{C})\}$. This solution configuration set can be described by the following propositional sentence:

$$\neg A \wedge ((\neg B \wedge C) \vee (B \wedge \neg C))\,.$$

As we can see, the semiring valuation ϕ and the given propositional sentence agree in their model sets. This illustrates well the keynote of the current section. $\ominus$

A similar strategy is embarked in (Mateescu & Dechter, 2006). There, already the knowledgebase factors (constraints) are compiled into a particular Boolean function and graphically represented as *AND/OR multi-valued decision diagrams*. The compilation process consists then in combining these graphs to the final compilation of the solution configuration set, with special focus on the retention of properties that make fast query execution possible. However, we will see that in our approach, the compilation process is independent from the representation of semiring valuations. Furthermore, the underlying join tree guarantees that all properties needed for fast query execution are naturally retained during the compilation process.

8.4.1 Memorizing Semiring Valuations

We assume a finite set of propositional variables r and write $\mathcal{P}(r)$ for its power set. Let A be a totally ordered, idempotent semiring whose order is strictly monotonic over $\times$. A *memorizing semiring valuation* ϕ with domain $s \subseteq r$ is defined to be a function that associates a two-dimensional vector with each configuration $\mathbf{x} \in \Omega_s$. The first element of this vector $\phi_A(\mathbf{x})$ corresponds again to a semiring value whereas the second element $\phi_F(\mathbf{x})$ constitutes a Boolean function defined over propositional variables in r. More formally, we have

$$\phi : \Omega_s \to A \times F_r \quad \text{and} \quad \mathbf{x} \mapsto (\phi_A(\mathbf{x}),\ \phi_F(\mathbf{x})), \tag{8.15}$$

where F_r is the set of Boolean functions over variables in r,

$$F_r \;=\; \{f : \{0,1\}^r \to \{0,1\}\}.$$

Essentially, the buildup of a memorizing semiring valuation is similar to the one of usual semiring valuations introduced in Section 6.3, with the extension that we additionally associate a Boolean function with each configuration. This new component represents the memory part and will be updated whenever a step towards the identification of solution configurations is performed. Note that we assume the Boolean functions to be defined over all variables in r. Since Boolean functions are considered in this context as a part of a configuration's value, they have no meaning with respect to the domain of a memorizing semiring valuation, which is defined as

$$d(\phi) = s \quad \text{if} \quad \phi : \Omega_s \to A \times F_r. \tag{8.16}$$

It is reasonable to assume the memory originally to be empty. Thus, memorizing semiring valuations are obtained from usual semiring valuations by initializing their memory to the *tautology function* $f_{\mathbf{1}} \in F_r$ with $f_{\mathbf{1}}(\mathbf{x}) = 1$ for all $\mathbf{x} \in \Omega_r$. In this way, we define for a semiring valuation ψ with $d(\psi) = t$,

$$\tilde{\psi} : \; \mathbf{x} \in \Omega_t \quad \mapsto \quad (\psi(\mathbf{x}),\; f_{\mathbf{1}})\,. \tag{8.17}$$

We shall now examine how memorizing semiring valuations are processed and how their memory component is updated. For this purpose, we again switch over to the notation of variable elimination instead of marginalization. Assume that we eliminate some propositional variable Y from a memorizing semiring valuation ϕ with $Y \in d(\phi) = s$. The semiring components are processed in the usual way as

$$\phi_A^{-Y}(\mathbf{x}) \;=\; \phi_A(\mathbf{x}, 0_Y) + \phi_A(\mathbf{x}, 1_Y) \;=\; \max\{\phi_A(\mathbf{x}, 0_Y),\; \phi_A(\mathbf{x}, 1_Y)\}$$

with $\mathbf{x} \in \Omega_{s-\{Y\}}$. For the memory component, we first introduce a particular Boolean function that takes center stage in the following definition. Let $X \in r$, then we define its *Boolean identity function* $f_X \in \mathcal{F}_r$ by $f_X(\mathbf{x}) = 0$ if $X = 0$ holds in the configuration $\mathbf{x} \in \Omega_r$. Otherwise, if $X = 1$ we define $f_X(\mathbf{x}) = 1$. Thus, f_X always maps on the current value of variable X. Similarly, we define the *inverse Boolean identity function* $f_{\overline{X}} \in \mathcal{F}_r$ by $f_{\overline{X}}(\mathbf{x}) = 0$ if $X = 1$ holds in the configuration $\mathbf{x} \in \Omega_r$ and $f_{\overline{X}}(\mathbf{x}) = 1$ otherwise.

For the definition of variable elimination for the memory component, we distinguish the following three cases:

1. If $\phi_A(\mathbf{x}, 0_Y) < \phi_A(\mathbf{x}, 1_Y)$, $Y = 1$ is part of a valid solution configuration of ϕ. In terms of Boolean functions, this requirement is reflected by the identity function $f_Y \in F_r$. Additionally, we can ignore the memory $\phi_F(\mathbf{x}, 0_Y)$ since it would lead to a configuration where $Y = 0$, and this cannot be a solution configuration. Hence, we connect the memory $\phi_F(\mathbf{x}, 1_Y)$ conjunctively with f_Y and obtain $\phi_F(\mathbf{x}) = \min\{f_Y, \phi_F(\mathbf{x}, 1_Y)\}$ according to Section 6.2.6. Alternatively, we will also write $f_Y \wedge \phi_F(\mathbf{x}, 1_Y)$ to accent the conjunctive combination of the two Boolean functions.

2. If on the other hand $\phi_A(\mathbf{x}, 1_\mathrm{Y}) < \phi_A(\mathbf{x}, 0_\mathrm{Y})$, $Y = 0$ is part of a valid solution configuration and this can be achieved by the inverse identity function $f_{\overline{Y}} \in F_r$. Following the same argumentation as above, we obtain $\phi_F(\mathbf{x}) = \min\{f_{\overline{Y}}, \phi_F(\mathbf{x}, 0_\mathrm{Y})\}$ or equivalently $f_{\overline{Y}} \wedge \phi_F(\mathbf{x}, 0_\mathrm{Y})$.
3. Finally, if $\phi_A(\mathbf{x}, 0_\mathrm{Y}) = \phi_A(\mathbf{x}, 1_\mathrm{Y})$, both assignments $Y = 0$ and $Y = 1$ lead to solution configurations. Therefore, the memory can either be updated as $\min\{f_Y, \phi_F(\mathbf{x}, 1_\mathrm{Y})\}$ or as $\min\{f_{\overline{Y}}, \phi_F(\mathbf{x}, 0_\mathrm{Y})\}$. Since we aim for the identification of all solution configurations, we connect the two updates disjunctively and obtain $\phi_F(\mathbf{x}) = \max\{\min\{f_Y, \phi_F(\mathbf{x}, 1_\mathrm{Y})\}, \min\{f_{\overline{Y}}, \phi_F(\mathbf{x}, 0_\mathrm{Y})\}\}$. In the same way, we also write $(f_Y \wedge \phi_F(\mathbf{x}, 1_\mathrm{Y})) \vee (f_{\overline{Y}} \wedge \phi_F(\mathbf{x}, 0_\mathrm{Y}))$.

From this train of thought, we derive the following definition of variable elimination for the set Φ of memorizing semiring valuations over propositional variables in r:

Variable Elimination: $\Phi \times r \to \Phi$: for $\phi \in \Phi$, $Y \in d(\phi)$ and $\mathbf{x} \in \Omega_{d(\phi)-\{Y\}}$

$$\phi^{-Y}(\mathbf{x}) \quad = \quad (\phi_A^{-Y}(\mathbf{x}),\ \phi_F^{-Y}(\mathbf{x})), \tag{8.18}$$

where

$$\phi_A^{-Y}(\mathbf{x}) \quad = \quad \phi_A(\mathbf{x}, 0_\mathrm{Y}) + \phi_A(\mathbf{x}, 1_\mathrm{Y})$$

and

$$\phi_F^{-Y}(\mathbf{x}) \quad = \quad \begin{cases} f_{\overline{Y}} \wedge \phi_F(\mathbf{x}, 0_\mathrm{Y}), & \text{if } \phi_A(\mathbf{x}, 0_\mathrm{Y}) > \phi_A(\mathbf{x}, 1_\mathrm{Y}) \\ f_Y \wedge \phi_F(\mathbf{x}, 1_\mathrm{Y}), & \text{if } \phi_A(\mathbf{x}, 0_\mathrm{Y}) < \phi_A(\mathbf{x}, 1_\mathrm{Y}) \\ (f_{\overline{Y}} \wedge \phi_F(\mathbf{x}, 0_\mathrm{Y})) \vee (f_Y \wedge \phi_F(\mathbf{x}, 1_\mathrm{Y})) & \text{otherwise.} \end{cases}$$

The following example illustrates the application of these rules:

Example 8.4. *Consider propositional variables A, B and the following valuation defined over a semiring with maximization for addition:*

$\phi \ =$

A	**B**		
0_A	0_B	10	$f_\mathbf{1}$
0_A	1_B	8	$f_\mathbf{1}$
1_A	0_B	3	$f_\mathbf{1}$
1_A	1_B	10	$f_\mathbf{1}$

First, we eliminate variable A and observe that $\phi_A(0_\mathrm{A}, 0_\mathrm{B}) > \phi_A(1_\mathrm{A}, 0_\mathrm{B})$. Therefore, we connect $f_{\overline{A}}$ conjunctively with the memory $\phi_F(0_\mathrm{A}, 0_\mathrm{B})$. Contrariwise, we have $\phi_A(0_\mathrm{A}, 1_\mathrm{B}) < \phi_A(1_\mathrm{A}, 1_\mathrm{B})$ which requires to connect f_A with the memory $\phi_F(1_\mathrm{A}, 1_\mathrm{B})$:

$\phi^{-A} \ =$

B		
0_B	10	$f_\mathbf{1} \wedge f_{\overline{A}} = f_{\overline{A}}$
1_B	10	$f_\mathbf{1} \wedge f_A = f_A$

Finally, variable B remains to be eliminated. Since $\phi_A^{-A}(0_\mathrm{B}) = \phi_A^{-A}(1_\mathrm{B})$, both constructions are performed and connected disjunctively. We obtain:

$$\phi^{-\{A,B\}} \;=\; \begin{array}{|c||c||l|}\hline & & \\ \hline \diamond & 10 & (f_{\overline{B}} \wedge f_{\overline{A}}) \vee (f_B \wedge f_A) \\ \hline \end{array}$$

Thus,

$$f \;=\; \left(\phi^{\downarrow\emptyset}\right)_F \;=\; (f_{\overline{B}} \wedge f_{\overline{A}}) \vee (f_B \wedge f_A)$$

and finally

$$\mathbf{Models}(f) \;=\; \{(0_{\mathrm{A}}, 0_{\mathrm{B}}), (1_A, 1_{\mathrm{B}})\} \;=\; c_\phi.$$

$\ominus$

The following lemma links the memory component of memorizing semiring valuations with the solution extension sets introduced in Section 8.2.

Lemma 8.14. *For $\phi \in \Phi_s$, $t \subseteq s$ and $\mathbf{x} \in \Omega_t$ we have*

$$W_\phi^t(\mathbf{x}) \;=\; \left[\mathbf{Models}\left(\phi_F^{\downarrow t}(\mathbf{x})\right)\right]^{\downarrow d(\phi)-t}. \tag{8.19}$$

Proof. In particular, we have for $t = d(\phi)$,

$$W_\phi^t(\mathbf{x}) = \{\diamond\} = \Omega_t^{\downarrow\emptyset} \;=\; [\mathbf{Models}(f_{\mathbf{1}})]^{\downarrow\emptyset} \;=\; [\mathbf{Models}(\phi_F(\mathbf{x}))]^{\downarrow\emptyset}.$$

Thus, the property holds for this special case. Next, we assume that it also holds for some $t \subseteq d(\phi)$ and prove its validity for $t - \{X\}$ with $X \in t$. We distinguish the following cases:

1. Assume $\phi(\mathbf{x}, 0_{\mathrm{X}}) > \phi(\mathbf{x}, 1_{\mathrm{X}})$. We have

$$\begin{aligned} W_\phi^{t-\{X\}}(\mathbf{x}^{\downarrow t-\{X\}}) &= \left\{(\mathbf{c}, 0_{\mathrm{X}}) : \mathbf{c} \in W_\phi^t(\mathbf{x})\right\} \\ &= \left\{(\mathbf{c}, 0_{\mathrm{X}}) : \mathbf{c} \in \left[\mathbf{Models}(\phi_F^{\downarrow t}(\mathbf{x}))\right]^{\downarrow d(\phi)-t}\right\} \\ &= \left[\mathbf{Models}\left(f_{\overline{X}} \wedge \phi_F^{\downarrow t}(\mathbf{x})\right)\right]^{\downarrow d(\phi)-(t-\{X\})} \\ &= \left[\mathbf{Models}\left(\phi_F^{\downarrow t-\{X\}}(\mathbf{x}^{\downarrow t-\{X\}})\right)\right]^{\downarrow d(\phi)-(t-\{X\})}. \end{aligned}$$

2. Assume $\phi(\mathbf{x}, 0_{\mathrm{X}}) < \phi(\mathbf{x}, 1_{\mathrm{X}})$. We have

$$\begin{aligned} W_\phi^{t-\{X\}}(\mathbf{x}^{\downarrow t-\{X\}}) &= \left\{(\mathbf{c}, 1_{\mathrm{X}}) : \mathbf{c} \in W_\phi^t(\mathbf{x})\right\} \\ &= \left\{(\mathbf{c}, 1_{\mathrm{X}}) : \mathbf{c} \in \left[\mathbf{Models}(\phi_F^{\downarrow t}(\mathbf{x}))\right]^{\downarrow d(\phi)-t}\right\} \\ &= \left[\mathbf{Models}\left(f_X \wedge \phi_F^{\downarrow t}(\mathbf{x})\right)\right]^{\downarrow d(\phi)-(t-\{X\})} \\ &= \left[\mathbf{Models}\left(\phi_F^{\downarrow t-\{X\}}(\mathbf{x}^{\downarrow t-\{X\}})\right)\right]^{\downarrow d(\phi)-(t-\{X\})}. \end{aligned}$$

3. Assume $\phi(\mathbf{x}, 0_{\mathrm{X}}) = \phi(\mathbf{x}, 1_{\mathrm{X}})$. We have

$$\begin{aligned}
W_{\phi}^{t-\{X\}}(\mathbf{x}^{\downarrow t-\{X\}}) &= \big\{(\mathbf{c}, 0_{\mathrm{X}}) : \mathbf{c} \in W_{\phi}^{t}(\mathbf{x})\big\} \cup \big\{(\mathbf{c}, 1_{\mathrm{X}}) : \mathbf{c} \in W_{\phi}^{t}(\mathbf{x})\big\} \\
&= \left\{(\mathbf{c}, 0_{\mathrm{X}}) : \mathbf{c} \in \Big[\mathbf{Models}(\phi_F^{\downarrow t}(\mathbf{x}))\Big]^{\downarrow d(\phi)-t}\right\} \cup \\
&\quad \left\{(\mathbf{c}, 1_{\mathrm{X}}) : \mathbf{c} \in \Big[\mathbf{Models}(\phi_F^{\downarrow t}(\mathbf{x}))\Big]^{\downarrow d(\phi)-t}\right\} \\
&= \Big[\mathbf{Models}\Big(f_{\overline{X}} \wedge \phi_F^{\downarrow t}(\mathbf{x})\Big)\Big]^{\downarrow d(\phi)-(t-\{X\})} \cup \\
&\quad \Big[\mathbf{Models}\Big(f_X \wedge \phi_F^{\downarrow t}(\mathbf{x})\Big)\Big]^{\downarrow d(\phi)-(t-\{X\})} \\
&= \Big[\mathbf{Models}\Big((f_{\overline{X}} \wedge \phi_F^{\downarrow t}(\mathbf{x})) \vee (f_X \wedge \phi_F^{\downarrow t}(\mathbf{x}))\Big)\Big]^{\downarrow d(\phi)-(t-\{X\})} \\
&= \Big[\mathbf{Models}\Big(\phi_F^{\downarrow t-\{X\}}(\mathbf{x}^{\downarrow t-\{X\}})\Big)\Big]^{\downarrow d(\phi)-(t-\{X\})}.
\end{aligned}$$

□

The next theorem follows immediately from this Lemma and Equation (8.9).

Theorem 8.15. *It holds that*

$$\mathbf{Models}\Big(\phi_F^{\downarrow \emptyset}(\diamond)\Big) \;=\; W_{\phi}^{\emptyset}(\diamond) \;=\; c_{\phi}. \tag{8.20}$$

Consequently, we found an implicit representation, also called *compilation*, of the solution configuration set of ϕ.

Regarding the definition of an optimization problem, ϕ is generally given by a factorization $\phi = \psi_1 \otimes \cdots \otimes \psi_m$. In order to address such problems with memorizing semiring valuations, we need to define a combination operator for this formalism. Again, the semiring components are processed in the usual way,

$$(\phi \otimes \psi)_A(\mathbf{x}) \;=\; \phi_A(\mathbf{x}^{\downarrow d(\phi)}) \times \psi_A(\mathbf{x}^{\downarrow d(\psi)}).$$

Equally simple is the definition for the memory components. Since both memories must be taken into account, they are joined conjunctively as follows:

$$(\phi \otimes \psi)_F(\mathbf{x}) \;=\; \min\{\phi_F(\mathbf{x}^{\downarrow d(\phi)}),\; \psi_F(\mathbf{x}^{\downarrow d(\psi)})\} \;=\; \phi_F(\mathbf{x}^{\downarrow d(\phi)}) \wedge \psi_F(\mathbf{x}^{\downarrow d(\psi)}).$$

To sum it up, we adopt the following definition of combination for memorizing semiring valuations:

Combination: $\Phi \times \Phi \to \Phi$: for $\phi \in \Phi_s$, $\psi \in \Phi_t$ and $\mathbf{x} \in \Omega_{s \cup t}$

$$(\phi \otimes \psi)(\mathbf{x}) \;=\; (\phi_A(\mathbf{x}^{\downarrow s}) \times \psi_A(\mathbf{x}^{\downarrow t}),\; \phi_F(\mathbf{x}^{\downarrow s}) \wedge \psi_F(\mathbf{x}^{\downarrow t})). \tag{8.21}$$

Thus, given a factorization $\phi = \psi_1 \otimes \cdots \otimes \psi_m$ of semiring valuations for some objective function ϕ, we can build the compilation of the solution configurations of ϕ by the following procedure:

1. Construct memorizing semiring valuations $\tilde{\psi}_i$ using Equation (8.17).
2. Compute $\tilde{\phi}$ where $\tilde{\phi}_A(\mathbf{x}) = (\psi_1 \otimes \cdots \otimes \psi_m)(\mathbf{x})$ and $\tilde{\phi}_F(\mathbf{x}) = f_{\mathbf{1}}$.
3. Compute $\tilde{\phi}^{\downarrow\emptyset}$ by eliminating all variables in $d(\phi) = d(\tilde{\phi})$.
4. Extract $\tilde{\phi}_F^{\downarrow\emptyset}(\diamond)$ from $\tilde{\phi}^{\downarrow\emptyset}$.

Once more, this procedure leads indeed to the correct result, but is worthless in practice due to the computational intractability of building $\tilde{\phi}$. But with the acquired knowledge from the preceding chapters, this problem can be solved in a very smart manner by proving that memorizing semiring valuations satisfy the valuation algebra axioms from Section 2.1. Then, we compute $\tilde{\phi}^{\downarrow\emptyset}$ by application of the collect algorithm and avoid in this way the addressed complexity problems.

Theorem 8.16. *A system of memorizing semiring valuations (Φ, D) with labeling (8.16), combination (8.21) and variable elimination (8.18), satisfies the axioms of a valuation algebra.*

Proof. The Axioms (A2) and (A3) are immediate consequences of the above definitions. Furthermore, we have also seen that the commutativity and associativity of max and min (written as $\vee$ and $\wedge$) are sufficient conditions for Axiom (A1). It remains to prove that the two Axioms (A4) and (A5) are satisfied, respectively their counterpart from Lemma 2.3 using variable elimination instead of marginalization.

(A4) We will show that for $X, Y \in s = d(\phi)$ and $\mathbf{z} \in \Omega_{s-\{X,Y\}}$

$$(\phi_F^{-X})^{-Y}(\mathbf{z}) \quad = \quad (\phi_F^{-Y})^{-X}(\mathbf{z}). \tag{8.22}$$

We use the following short notation for the values that are of interest:

$$\begin{aligned} v_1 &= \phi_A(\mathbf{z}, 0_X, 0_Y), & f_1 &= \phi_F(\mathbf{z}, 0_X, 0_Y), \\ v_2 &= \phi_A(\mathbf{z}, 0_X, 1_Y), & f_2 &= \phi_F(\mathbf{z}, 0_X, 1_Y), \\ v_3 &= \phi_A(\mathbf{z}, 1_X, 0_Y), & f_3 &= \phi_F(\mathbf{z}, 1_X, 0_Y), \\ v_4 &= \phi_A(\mathbf{z}, 1_X, 1_Y), & f_4 &= \phi_F(\mathbf{z}, 1_X, 1_Y). \end{aligned}$$

The definition of variable elimination is based on the comparison of the values v_i. For this proof, we can restrict ourselves to the two cases $>_{id}$ and $=$, because all other arrangements including $<_{id}$ are symmetric. More concretely, we need to prove the following four cases:

1. $v_1 >_{id} v_2 >_{id} v_3 >_{id} v_4$ (all values are different),
2. $v_1 = v_2 >_{id} v_3 >_{id} v_4$ (two values are equal),
3. $v_1 = v_2 = v_3 >_{id} v_4$ (three values are equal),
4. $v_1 = v_2 = v_3 = v_4$ (all values are equal).

All other cases are covered by these representatives as explained subsequently. We will thus verify Equation (8.22) under the above four situations:

1. Assume that $v_1 >_{id} v_2 >_{id} v_3 >_{id} v_4$. Then we have by application of (8.18)

$$\begin{aligned}
\phi^{-X}(\mathbf{z}, 0_Y) &= \big(v_1,\ f_{\overline{X}} \wedge f_1\big),\\
\phi^{-X}(\mathbf{z}, 1_Y) &= \big(v_2,\ f_{\overline{X}} \wedge f_2\big),\\
(\phi^{-X})^{-Y}(\mathbf{z}) &= \big(v_1,\ f_{\overline{Y}} \wedge f_{\overline{X}} \wedge f_1\big).
\end{aligned}$$

In the same way, we compute

$$\begin{aligned}
\phi^{-Y}(\mathbf{z}, 0_X) &= \big(v_1,\ f_{\overline{Y}} \wedge f_1\big),\\
\phi^{-Y}(\mathbf{z}, 1_X) &= \big(v_3,\ f_{\overline{Y}} \wedge f_3\big),\\
(\phi^{-Y})^{-X}(\mathbf{z}) &= \big(v_1,\ f_{\overline{X}} \wedge f_{\overline{Y}} \wedge f_1\big).
\end{aligned}$$

Thus, Equality (8.22) holds for this first case. Note that it covers also the case where $v_1 >_{id} v_2 >_{id} v_3 = v_4$ because here only $\phi_F^{-Y}(\mathbf{z}, 0_X)$ changes, which has no influence. In the same way, the cases $v_1 >_{id} v_2 = v_3 >_{id} v_4$ and $v_1 >_{id} v_2 = v_3 = v_4$ are also handled by this argument. To sum it up, we can say that all cases are handled where one value is strictly larger than all others.

2. Assume that $v_1 = v_2 >_{id} v_3 >_{id} v_4$. Then we have again

$$\begin{aligned}
\phi^{-X}(\mathbf{z}, 0_Y) &= \big(v_1,\ f_{\overline{X}} \wedge f_1\big),\\
\phi^{-X}(\mathbf{z}, 1_Y) &= \big(v_2,\ f_{\overline{X}} \wedge f_2\big),\\
(\phi^{-X})^{-Y}(\mathbf{z}) &= \big(v_1,\ (f_{\overline{Y}} \wedge f_{\overline{X}} \wedge f_1) \vee (f_Y \wedge f_{\overline{X}} \wedge f_2)\big).
\end{aligned}$$

Since this time we have equality between v_1 and v_2, we obtain

$$\begin{aligned}
\phi^{-Y}(\mathbf{z}, 0_X) &= \big(v_1,\ (f_{\overline{Y}} \wedge f_1) \vee (f_Y \wedge f_2)\big),\\
\phi^{-Y}(\mathbf{z}, 1_X) &= \big(v_3,\ f_{\overline{Y}} \wedge f_3\big),\\
(\phi^{-Y})^{-X}(\mathbf{z}) &= \big(v_1,\ f_{\overline{X}} \wedge ((f_{\overline{Y}} \wedge f_1) \vee (f_Y \wedge f_2))\big).
\end{aligned}$$

We can see immediately that Equality (8.22) is satisfied. This covers also the case $v_1 = v_2 >_{id} v_3 = v_4$, because here only $\phi_F^{-Y}(\mathbf{z}, 0_X)$ changes which again has no influence. More generally, all cases are handled where two values are equal but strictly larger than the two others.

3. Assume that $v_1 = v_2 = v_3 >_{id} v_4$. Then we have

$$\begin{aligned}
\phi^{-X}(\mathbf{z}, 0_Y) &= \big(v_1,\ (f_{\overline{X}} \wedge f_1) \vee (f_X \wedge f_3)\big),\\
\phi^{-X}(\mathbf{z}, 1_Y) &= \big(v_2,\ f_{\overline{X}} \wedge f_2\big),\\
(\phi^{-X})^{-Y}(\mathbf{z}) &= \big(v_1,\ (f_{\overline{Y}} \wedge ((f_{\overline{X}} \wedge f_1) \vee (f_X \wedge f_3))) \vee (f_Y \wedge f_{\overline{X}} \wedge f_2)\big).
\end{aligned}$$

In the same way, we compute

$$\begin{aligned}
\phi^{-X}(\mathbf{z}, 0_X) &= \big(v_1,\ (f_{\overline{Y}} \wedge f_1) \vee (f_Y \wedge f_2)\big),\\
\phi^{-Y}(\mathbf{z}, 1_X) &= \big(v_3,\ f_{\overline{Y}} \wedge f_3\big),\\
(\phi^{-Y})^{-X}(\mathbf{z}) &= \big(v_1,\ (f_{\overline{X}} \wedge ((f_{\overline{Y}} \wedge f_1) \vee (f_Y \wedge f_2))) \vee (f_X \wedge f_{\overline{Y}} \wedge f_3)\big).
\end{aligned}$$

Because

$$\begin{aligned}(\phi^{-X})_F^{-Y}(\mathbf{z}) &= (f_{\overline{Y}} \wedge ((f_{\overline{X}} \wedge f_1) \vee (f_X \wedge f_3))) \vee (f_Y \wedge f_{\overline{X}} \wedge f_2) \\ &= (f_{\overline{Y}} \wedge f_{\overline{X}} \wedge f_1) \vee (f_{\overline{Y}} \wedge f_X \wedge f_3) \vee (f_Y \wedge f_{\overline{X}} \wedge f_2) \\ &= (f_{\overline{X}} \wedge ((f_{\overline{Y}} \wedge f_1) \vee (f_Y \wedge f_2))) \vee (f_X \wedge f_{\overline{Y}} \wedge f_3) \\ &= (\phi^{-Y})_F^{-X}(\mathbf{z}),\end{aligned}$$

Equality (8.22) is again satisfied. This covers all cases where three values are equal but strictly larger than the forth value.

4. At last, assume that $v_1 = v_2 = v_3 = v_4$. Then we have

$$\begin{aligned}\phi^{-X}(\mathbf{z}, 0_Y) &= \big(v_1,\ (f_{\overline{X}} \wedge f_1) \vee (f_X \wedge f_3)\big), \\ \phi^{-X}(\mathbf{z}, 1_Y) &= \big(v_2,\ (f_{\overline{X}} \wedge f_2) \vee (f_X \wedge f_4)\big), \\ (\phi^{-X})^{-Y}(\mathbf{z}) &= (v_1,\ (f_{\overline{Y}} \wedge ((f_{\overline{X}} \wedge f_1) \vee (f_X \wedge f_3))) \vee \\ &\qquad (f_Y \wedge ((f_{\overline{X}} \wedge f_2) \vee (f_X \wedge f_4)))).\end{aligned}$$

And in the same way

$$\begin{aligned}\phi^{-X}(\mathbf{z}, 0_X) &= (v_1,\ (f_{\overline{Y}} \wedge f_1) \vee (f_Y \wedge f_2)), \\ \phi^{-Y}(\mathbf{z}, 1_X) &= (v_1,\ (f_{\overline{Y}} \wedge f_3) \vee (f_Y \wedge f_4)), \\ (\phi^{-Y})^{-X}(\mathbf{z}) &= (v_1,\ (f_{\overline{X}} \wedge ((f_{\overline{Y}} \wedge f_1) \vee (f_Y \wedge f_2)) \vee \\ &\qquad (f_X \wedge ((f_{\overline{Y}} \wedge f_3) \vee (f_Y \wedge f_4))))).\end{aligned}$$

As in the former case, $(\phi^{-X})_F^{-Y}(\mathbf{z})$ transforms into $(\phi^{-Y})_F^{-X}(\mathbf{z})$ by successive application of the distributive law. This completes the proof of the variable elimination axiom.

(A5) We will show that for $X \in d(\psi) = t$, $X \notin d(\phi) = s$ and $\mathbf{z} \in \Omega_{s \cup t - \{X\}}$,

$$(\phi_F \otimes \psi_F)^{-X}(\mathbf{z}) = \left(\phi_F \otimes \psi_F^{-X}\right)(\mathbf{z}).$$

Let us distinguish two cases:

1. Assume $(\phi_A \otimes \psi_A)(\mathbf{z}, 0_X) >_{id} (\phi_A \otimes \psi_A)(\mathbf{z}, 1_X)$. We obtain:

$$\begin{aligned}(\phi_F \otimes \psi_F)^{-X}(\mathbf{z}) &= f_{\overline{X}} \wedge (\phi_F \otimes \psi_F)(\mathbf{z}, 0_X) \\ &= f_{\overline{X}} \wedge \phi_F(\mathbf{z}^{\downarrow s}) \wedge \psi_F(\mathbf{z}^{\downarrow t - \{X\}}, 0_X) \\ &= \phi_F(\mathbf{z}^{\downarrow s}) \wedge (f_{\overline{X}} \wedge \psi_F(\mathbf{z}^{\downarrow t - \{X\}}, 0_X)).\end{aligned}$$

Since $\times$ behaves strictly monotonic,

$$\begin{aligned}(\phi_A \otimes \psi_A)(\mathbf{z}, 0_X) &= \phi_F(\mathbf{z}^{\downarrow s}) \times \psi_F(\mathbf{z}^{\downarrow t - \{X\}}, 0_X) \\ &>_{id} \phi_F(\mathbf{z}^{\downarrow s}) \times \psi_F(\mathbf{z}^{\downarrow t - \{X\}}, 1_X) = (\phi_A \otimes \psi_A)(\mathbf{z}, 1_X)\end{aligned}$$

implies that

$$\psi_F(\mathbf{z}^{\downarrow t-\{X\}}, 0_X) \quad >_{id} \quad \psi_F(\mathbf{z}^{\downarrow t-\{X\}}, 1_X).$$

Therefore, we obtain

$$\phi_F(\mathbf{z}^{\downarrow s}) \wedge (X \wedge \psi_F(\mathbf{z}^{\downarrow t-\{X\}}, 0_X)) \;=\; \phi_F(\mathbf{z}^{\downarrow s}) \wedge \psi_F^{-X}(\mathbf{z}^{\downarrow t}) \;=\; (\phi_F \otimes \psi_F^{-X})(\mathbf{z}).$$

2. Assume $(\phi_A \otimes \psi_A)(\mathbf{z}, 0_X) = (\phi_A \otimes \psi_A)(\mathbf{z}, 1_X)$. We know by the combination axiom (A5) for semiring valuations, that

$$(\phi_A \otimes \psi_A)^{-X}(\mathbf{z}) \quad = \quad (\phi_A \otimes \psi_A^{-X})(\mathbf{z}).$$

The left-hand part of this equation is written as

$$(\phi_A \otimes \psi_A)^{-X}(\mathbf{z}) \quad = \quad (\phi_A \otimes \psi_A)(\mathbf{z}, 0_X) + (\phi_A \otimes \psi_A)(\mathbf{z}, 1_X),$$

and the right-hand part as

$$(\phi_A \otimes \psi_A^{-X})(\mathbf{z}) \quad = \quad \phi_A(\mathbf{z}^{\downarrow s}) \times \Big(\psi_A(\mathbf{z}^{\downarrow t-\{X\}}, 0_X) + \psi_A(\mathbf{z}^{\downarrow t-\{X\}}, 1_X)\Big).$$

Remembering that + is idempotent, we obtain the following two equations:

(a) $(\phi_A \otimes \psi_A)(\mathbf{z}, 0_X) \;=\; \phi_A(\mathbf{z}^{\downarrow s}) \times \big(\psi_A(\mathbf{z}^{\downarrow t-\{X\}}, 0_X) + \psi_A(\mathbf{z}^{\downarrow t-\{X\}}, 1_X)\big),$

(b) $(\phi_A \otimes \psi_A)(\mathbf{z}, 1_X) \;=\; \phi_A(\mathbf{z}^{\downarrow s}) \times \big(\psi_A(\mathbf{z}^{\downarrow t-\{X\}}, 0_X) + \psi_A(\mathbf{z}^{\downarrow t-\{X\}}, 1_X)\big).$

Since $(\phi_A \otimes \psi_A)(\mathbf{z}, 0_X) = \phi_A(\mathbf{z}^{\downarrow s}) \times \psi_A(\mathbf{z}^{\downarrow t-\{X\}}, 0_X)$ and + corresponds to maximization, we conclude from (a) that

$$\psi_A(\mathbf{z}^{\downarrow t-\{X\}}, 1_X) \quad \leq_{id} \quad \psi_A(\mathbf{z}^{\downarrow t-\{X\}}, 0_X)$$

must hold. In the same way, we derive from (b) that

$$\psi_A(\mathbf{z}^{\downarrow t-\{X\}}, 0_X) \quad \leq_{id} \quad \psi_A(\mathbf{z}^{\downarrow t-\{X\}}, 1_X),$$

and consequently

$$\psi_A(\mathbf{z}^{\downarrow t-\{X\}}, 0_X) \quad = \quad \psi_A(\mathbf{z}^{\downarrow t-\{X\}}, 1_X).$$

Finally, we obtain:

$$\begin{aligned}
(\phi_F \otimes \psi_F)^{-X}(\mathbf{z}) \;&=\; \big(f_{\overline{X}} \wedge (\phi_F \otimes \psi_F)(\mathbf{z}, 0_X)\big) \vee (f_X \wedge (\phi_F \otimes \psi_F)(\mathbf{z}, 1_X)) \\
&=\; (f_{\overline{X}} \wedge \Big(\phi_F(\mathbf{z}^{\downarrow s}) \wedge \psi_F(\mathbf{z}^{\downarrow t-\{X\}}, 0_X)\Big)) \vee \\
&\quad\; (f_X \wedge \Big(\phi_F(\mathbf{z}^{\downarrow s}) \wedge \psi_F(\mathbf{z}^{\downarrow t-\{X\}}, 1_X)\Big)) \\
&=\; \phi_F(\mathbf{z}^{\downarrow s}) \wedge \\
&\quad\; \Big((f_{\overline{X}} \wedge \psi_F(\mathbf{z}^{\downarrow t-\{X\}}, 0_X)) \vee (f_X \wedge \psi_F(\mathbf{z}^{\downarrow t-\{X\}}, 1_X)\Big) \\
&=\; \phi_F(\mathbf{z}^{\downarrow s}) \wedge \psi_F^{-X}(\mathbf{z}^{\downarrow t-\{X\}}) \;=\; (\phi_F \otimes \psi_F^{-X})(\mathbf{z}).
\end{aligned}$$

This proves the combination axiom.

□

With respect to the former sketch of algorithm, the insight that memorizing semiring valuations form a valuation algebra enables us to compute $\tilde{\phi}_F^{\downarrow\emptyset}(\diamond)$ by a simple run of the collect algorithm. Thus, we can say that the process of compiling solution configurations into a Boolean function is just another application of local computation. Consequently, this method inherits also the efficiency of the latter.

Algorithm 4:

- Construct memorizing semiring valuations $\tilde{\psi}_i$ using Equation (8.17).
- Execute collect with empty query on the factor set $\{\tilde{\psi}_1, \ldots, \tilde{\psi_m}\}$.
- Extract $\tilde{\phi}_F^{\downarrow\emptyset}(\diamond)$ from the result of the collect algorithm.

The correctness proof of this algorithm follows from Lemma 8.14.

8.4.2 Efficient Querying of Compiled Solution Configurations

The compilation technique for solution configurations satisfies already two important requirements: it is efficient due to its execution as a local computation algorithm, and the resulting Boolean function is a compact representation of the solution configuration set, because it does not depend on the number of elements in c_ϕ. In order to highlight the usefulness of this compilation process, we will next investigate which applications or queries can be performed efficiently of the resulting Boolean function. Thereby, we consider a query to be efficient, if it can be executed in polynomial time. For the identification of such queries, we should first survey a particular interesting way of representing Boolean functions in general, namely as *Propositional Directed Acyclic Graphs (PDAG)* (Wachter & Haenni, 2006).

Definition 8.17. *A PDAG over a set of propositional variables r is a rooted directed acyclic graph of the following form:*

1. *Leaves are represented by $\bigcirc$ and labeled with $\top$ (true), $\bot$ (false), or $X \in r$.*
2. *Non-leaf nodes are represented by $\triangle$ (logical conjunction), $\triangledown$ (logical disjunction) or $\diamond$ (logical negation).*
3. *$\triangle$- and $\triangledown$-nodes have at least one child, and $\diamond$-nodes have exactly one child.*

Leaves labeled with $\top$ ($\bot$) represent the Boolean function that constantly maps to 1 (0). Those labeled with $X \in r$ represent the Boolean identity function f_X. The Boolean function represented by a $\triangle$-node is the one that evaluates to 1 if, and only if, all its children evaluate to 1. Correspondingly, $\triangledown$-nodes evaluate to 1 if, and only if, at least one of their children evaluates to 1. Finally, a $\diamond$-node represents the complementary Boolean function of its child.

Figure 8.1 illustrates the PDAG structures that are created from the three variable elimination rules of memorizing semiring valuations. Combination on the other hand just connects two existing PDAGs by a conjunction node to a new PDAG. Thus, because all variables are eliminated, we obtain for $\tilde{\phi}_F^{\downarrow\emptyset}(\diamond)$ a single PDAG structure. Some particularities of this graph are summarized in the following lemma.

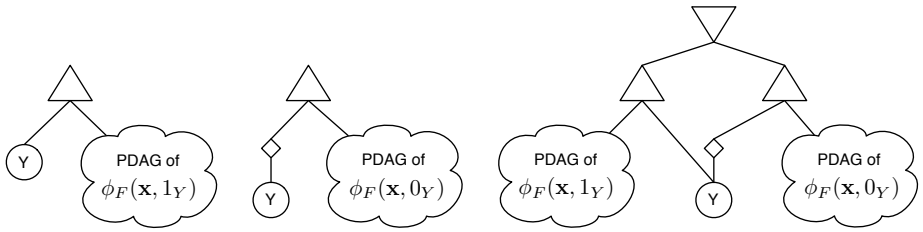


Figure 8.1: PDAG structures that are created from the variable elimination rules of memorizing semiring valuations.

Lemma 8.18. *The PDAG representation of* $\tilde{\phi}_F^{\downarrow\emptyset}(\diamond)$ *satisfies the following properties:*

1. *Simple-Negation: Each child of a* $\diamond$*-node is a leaf.*

2. *Decomposability: The sets of variables that occur in the sub-trees of every* $\triangle$*-node are disjoint.*

3. *Determinism: The children of every* $\bigtriangledown$*-node are pairwise logically contradictory., i.e. if* α_i *and* α_j *are two children of the same* $\bigtriangledown$*-node, we have* $\alpha_i \wedge \alpha_j \equiv \bot$.

Proof. The first property follows directly from PDAGs in the middle and on the right hand side of Figure 8.1 because these are the only rules that create negation nodes. A variable node is created whenever the corresponding variable is eliminated. Hence, every variable node occurs exactly once in this PDAG which proves Property 2. Finally, we can conclude from PDAG 3 in Figure 8.1 that in every model of the disjunction node's left child, $Y = 1$ must hold. Similarly, $Y = 0$ must hold in the right child and therefore, the two model sets are disjoint. This is the statement of Property 3. □

A PDAG that satisfies the three properties of Lemma 8.18 is called *cdn-PDAG* (Wachter & Haenni, 2006) or *d-DNNF* (Darwiche & Marquis, 2002). Before we shall now focus on the computational possibilities that are given by d-DNNF structures, we want to point out an important detail in the proof of Lemma 8.18. There, d-DNNFs are built from connecting existing d-DNNFs by either a conjunction or a disjunction node. However, we know from (Darwiche & Marquis, 2002) that the d-DNNF language is not closed under these operations. This means concretely that it is in general not possible to reconstruct a d-DNNF structure from the conjunction or disjunction of two d-DNNFs in polynomial time. Fortunately, this does not hold for the case at hand. Since these constructions are performed as valuation algebra operations, we directly obtain the cdn-properties whenever we join two existing d-DNNFs by the rules specified for memorizing semiring valuations.

Model Selection & Enumeration

Obviously, the first queries that have to be checked for their efficiency on d-DNNF structures are *model selection* and *model enumeration*. Remember, the models of the d-DNNF structure $\tilde{\phi}_F^{\downarrow\emptyset}(\diamond)$ are the solution configurations of c_ϕ. Therefore, if these configurations are selectable and enumerable efficiently, we can conclude that the compilation method is at least as powerful as the local computation schemes from Section 8.3. This is exactly the statement of the following theorem.

Theorem 8.19. *A d-DNNF structure allows model enumeration in polynomial time.*

(Darwiche, 2001) gives the following recursive procedure for model enumeration in general DNNFs:

Algorithm 5:

1. $\mathbf{Models}(\bigcirc) = \begin{cases} \{\{X = x\}\}, & \text{if } \bigcirc \text{ is labeled with variable } X, \\ \{\{\}\}, & \text{if } \bigcirc \text{ is labeled with } \top, \\ \{\}, & \text{if } \bigcirc \text{ is labeled with } \bot. \end{cases}$

2. $\mathbf{Models}(\diamond) = \{\{X = \overline{x}\}\}$ with X being the label of its $\bigcirc$ child node.

3. $\mathbf{Models}(\bigtriangledown) = \bigcup \mathbf{Models}(\bigtriangledown_i)$ where $\bigtriangledown_i$ are the children of node $\bigtriangledown$.

4. $\mathbf{Models}(\bigtriangleup) = \{\bigcup \mu : \mu \in \mathbf{Models}(\bigtriangleup_i)\}$ where $\bigtriangleup_i$ are the children of node $\bigtriangleup$.

(Darwiche, 2001) estimates the complexity of this procedure at $O(mn^2)$ where m is the size of the d-DNNF and n the number of its models. If we consider the number of models as a constant (which is reasonable for the task of model enumeration), the above algorithm becomes linear in the depth of the d-DNNF. If on the other hand we are interested in only one solution configuration, it is sufficient to choose (non-deterministically) a model of an arbitrary child in Rule 4 and ignore the remaining models. This procedure is called model selection.

Further Efficient Queries

To conclude this section, we want to convince the reader that the compilation approach for solution configurations is indeed a lot more powerful than the explicit methods of Section 8.3. For this purpose, we give a survey of further queries that can be performed efficiently on d-DNNFs and therefore on the result of the compilation process. Three such queries have already been mentioned in the introduction of this section. These are solution counting which determines the number of solution configurations without their explicit enumeration, countersolution configuration enumeration and selection which aim for the identification of all configurations that are not solution configurations, and validity check which determines whether all configurations of the objective function adopt the same value. (Wachter & Haenni, 2006) extend this listing by a couple of even more sophisticated applications:

- **Probabilistic Equivalence Test**: If two different factorizations over the same set of variables are given, d-DNNFs allow to test probabilistically if the two objective functions ϕ_1 and ϕ_2 adopt the same set of solution configurations, i.e. if $c_{\phi_1} = c_{\phi_2}$. This test is described in (Darwiche & Huang, 2002).
- **Probability Computation:** If we assume independent marginal probabilities $p(X)$ for all variables $X \in d(\phi)$, we can efficiently evaluate the probability of the solution configuration set c_ϕ. How exactly these probabilities are computed can be seen in (Darwiche, 2001).

8.5 Conclusion

At the very beginning of this chapter, we have seen that projection problems turn into an optimization tasks when dealing with valuations based on totally ordered, idempotent semirings. By application of the collect algorithm, one can therefore identify the optimum value of the objective function with respect to the natural semiring order. We then come across a new computational problem if we ask for a configuration that adopts the computed value. Such configurations are called solution configurations and their identification and enumeration took central stage in Section 8.3. The proposed algorithms exploit local computation techniques and are closely related to dynamic programming. In the final section, we raised further claims and asked for a compact representation of the solution configurations that allows more sophisticated manipulations such as computing the probability of some configuration or testing the equivalence of objective functions. We developed the idea of a pre-compilation step that expresses the solution configuration set by a suitable Boolean function which then allows the efficient execution of such additional queries. The compilation process itself is a local computation algorithm which ensures tractable complexity. This illuminates a new area of application for local computation techniques, and broadens in this way the horizon of classical dynamic programming.

Part IV

NENOK

9
Introduction

First and foremost, the major goal of the NENOK software is best outlined by describing it as a *generic software framework for local computation*, and it is worth spending some time on explaining what exactly is meant by this title. Evidently, a valuation algebra is a *generic* concept. It consists of three principal operations and a small set of axioms that provides sufficient structure for the application of local computation. In addition, Chapter 2 introduced many variations of this mathematical setting, which altogether span the algebraic framework of Figure 2.3. Among other things, NENOK focuses on a software representation of this framework that mirrors best possible the tight and well-studied connections between the involved algebraic components. These inheritance relationships, as well as the perception of valuations as pieces of knowledge, suggest the use of an object-oriented language for this undertaking. Based upon the algebraic software layer which also includes the semiring extension of valuation algebras, NENOK offers *generic implementations* of all local computation architectures introduced in Chapter 4 and Section 8.3. This makes NENOK an experimental platform where users can plug in their own valuation algebra implementations and access the rich library of local computation features.

The distributed nature of knowledge and information is a second aspect that informed the development process. Mathematically, a knowledgebase consists of a simple set of valuations, but in reality, the individual knowledge pieces are often distributed and accessible only through a common network. This leads to the more realistic concept of distributed knowledgebases as introduced in Chapter 5. NENOK features a complete service infrastructure that allows to locate and retrieve valuations residing on remote network hosts, but also to delegate the execution of valuation algebra operations to foreign processors. Then, all local computation architectures are implemented as intrinsically distributed algorithms which compute projection problems while striving for minimal communication effort. All this must be done in a highly transparent manner to conserve the user's perception of a simple and easy to use workbench. This principle also motivated the evolution of a graphical user interface for the monitoring of local computation.

To substantiate this first impression of the demands on the NENOK framework, we refer to the use case diagram of Figure 9.1 that summarizes the most important features. Clearly, one must not forget that all these features are generic in the sense that they refer to a user-defined valuation algebra implementation. Furthermore, all functionalities have to be realized in such a way that they do not distinguish between local and remote valuation objects.

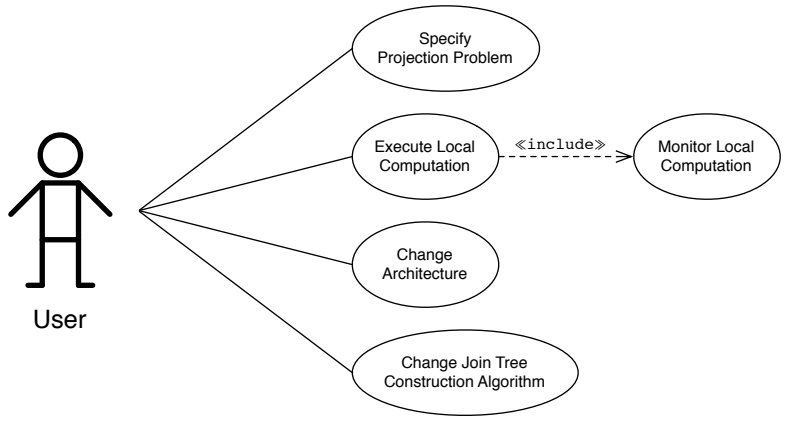


Figure 9.1: Use case diagram for the most important NENOK features.

An overview of the main idea behind NENOK should also include a question of scientific interest that is associated with the representation of the algebraic framework. More concretely, this project turns out to be a perfect case study of how far a complex mathematical framework can be reflected by an object-oriented programming language. We will have a short look at the numerous problems and pitfalls of the development process throughout the following sections.

9.1 Naming & History

In Persian mythology, *Nenok* stands for the ideal world of abstract being. According to the Iranian prophet Zarathustra, the world was first created in an only spiritual form and named *Nenok*. Three millennia later, its material form called *Geti* finally accrued. We believe that this duality of abstractness and concreteness is perfectly mirrored in the theory of valuation algebras. Therefore, the name NENOK was chosen for this software project.

The foundation of the NENOK framework was already laid in the author's master's thesis (Pouly, 2004a; Pouly, 2004b) and provided an implementation of the Shenoy-Shafer architecture based on a very restricted algebraic framework. The following release named NENOK 1.1 (Pouly, 2006) consisted of an additional communication layer on which all four local computation architectures of Chapter 4

were implemented as distributed algorithms. It was also built on a more sophisticated algebraic layer which already contained parts of the semiring framework. The steps towards NENOK 1.2 and 1.3 brought two major innovations. It became possible to run NENOK locally without setting up the server federation needed for distributed computing, and second, new possibilities for the graphical representation of join trees and valuation networks were explored. Finally, the development of the current NENOK 1.4 release was mainly focused on an improvement of performance and usability, but it also brings along an implementation of the dynamic programming algorithm of Section 8.3.2. NENOK is under GPL license and available on the project website:

- NENOK project website: `http://www.marcpouly.ch/nenok`

The worth of a software framework is best reflected by instantiations and extensions that are contributed by other authors. For the NENOK framework, there are different projects focussing on the implementation of some particular valuation algebra instances. (Langel, 2004) provides an implementation of propositional logic on model level as described in Section 3.8. (Jenal, 2006) furnishes an implementation of Gaussian potentials, specified in Section 3.7. The formalism of distance potentials from Section 3.6 is realized in (de Groote, 2006), and the relational algebra extract from Section 3.3 is developed as NENOK instance in (Spring, 2006). The last implementation has further been refined in (Schneuwly, 2007). This reference also exploits another plug-in mechanism that allows experienced users to extend NENOK with a personalized implementation of a local computation architecture. It concerns a modified collect algorithm for the efficient execution of *select queries* in relational algebras.

9.2 Technical Aspects & Notation

NENOK 1.4 is entirely written in Java 1.5 and equipped with a Jini 2.0 communication layer. Two further frameworks support the graphical facilities, namely Jung 1.7.6 (Java Universal Network/Graph Framework) and Hypergraph 0.6.3. Finally, different components of the Xerces 1.4.4 release enable efficient XML processing. Figure 9.2 illustrates this assembly of frameworks ignoring the fact that some of them are not purely written in Java. We also refer to the following project websites (last visited: 27th March 2008) for more information about these frameworks:

- Jini: `http://www.jini.org`
- Jung: `http://jung.sourceforge.net`
- Hypergraph: `http://hypergraph.sourceforge.net`
- Xerces: `http://xerces.apache.org`

The source code printed in this thesis is conform to Java 1.5, but for the sake of simplicity and clarity we will set dispensable details aside. Most of the exception

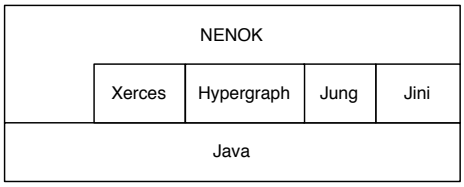


Figure 9.2: Buildup of NENOK 1.4 as framework assembly.

handling procedure is omitted, as well as package imports and main method definitions. An understanding of the internal class structure and component interaction is indispensable for users that wish to use NENOK for their own purposes. We will use UML 2.0 class and sequence diagrams for their illustration, but again, these diagrams are cut down to the important aspects.

9.3 The NENOK Architecture

Figure 9.3 outlines the NENOK architecture as a layered model. Each of its five layers benefits from the functionality of its underlying neighbor and attends only to the tasks related to itself. In the remaining chapters of this thesis, we will discuss every NENOK layer in more detail, starting with the algebraic layer. Here, we only summarize the main task of each layer:

- **Communication Layer:** The lowest layer establishes the basic communication utilities that will later be used to process knowledgebases with distributed factors. Most of this functionality communicates directly with the Jini framework.

- **Algebraic Layer:** The algebraic layer offers a generic implementation of the valuation algebra framework introduced in Chapter 2, as well as its semiring extension from Chapter 6.

- **Remote Computing Layer:** The remote computing layer provides services for storing valuation objects in globally shared memory spaces, in order to make them traceable and accessible for clients in the network. This realizes the idea of a distributed knowledgebase. Further, this layer makes the remote execution of valuation algebra operations possible.

- **Local Computation Layer:** The local computation layer consists of all requirements for applying local computation to either local or remote knowledgebase. It features implementations of all local computation architectures discussed in Chapter 4, and also includes the dynamic programming algorithm of Section 8.3.2. Due to the abstraction offered by the underlying layers, these algorithms do not distinguish between local and remote data.

- **User Interface:** The top layer offers a more intuitive access to NENOK by a set of default configurations and a graphical user interface. Additionally, an XML-based framework for generic input processing has been established on this layer.

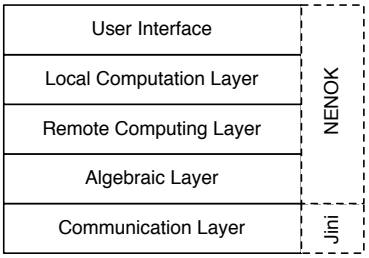


Figure 9.3: The NENOK architecture as a layered model.

The ability of NENOK to process distributed knowledgebases may become a hurdle for the unexperienced user. The reason is that a lot of configuration work is required for this setup: We need to run different Jini and NENOK services, edit configuration files and deal with security or network constraints. Although we do our best to explain the necessary preparation work, NENOK can nevertheless be used on a purely local setting. This reduces the complexity of the NENOK architecture to only three significant layers, as shown in Figure 9.4.

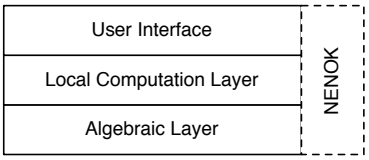


Figure 9.4: Simplified NENOK architecture for the processing of local data.

10
The Algebraic Layer

The introduction presents NENOK as a generic system into which users can plug their own valuation algebra implementations in order to experiment with local computation architectures. The algebraic layer is the central interface for this task since it defines exactly how this *plug-in mechanism* works. It is therefore indispensable for the NENOK user to become acquainted with every component of this layer, to study their interplay and to be sure of their mathematical counterparts. Valuation algebras consist of various components such as variables, domains, identity elements and valuations. They are furthermore classified into valuation algebras with division, idempotency, or semiring-based which brings up a second framework adding semiring values, properties and configurations. This gives a first impression of the complexity that comes with the construction of the algebraic layer. Subsequently, we guide the reader step by step through the realization of these mathematical structures and describe the measures that need to be taken to implement a valuation algebra instance. For better orientation, most sections will end with *developer notes* that summarize in a few words the most important points to remember for the programmer at work.

10.1 Valuation Algebra Core Components

We start out with the description of the smallest part of the algebraic framework whose instantiation is sufficient for the application of the Shenoy-Shafer architecture. This corresponds essentially to the standard valuation algebra definition given in Section 2.1 and consists of the components and relationships shown in Figure 10.1. The gray-framed interface does not belong to NENOK but is part of the standard Java distribution. All other components are contained in the package `nenok.va`. The first point to note is that every component within this framework extract implements either directly or indirectly the `java.io.Serializable` interface. This is a *marker interface*, i.e. an empty interface that is only used for typing purposes. It marks the serializability of valuation objects and their components which will later be important to transmit valuations between NENOK clients. Thus, an important consequence is that users are only allowed to use serializable components for

their implementation. This may sound constrictive but in fact, there are only a few non-serializable classes in Java, most of them are related to networking issues.

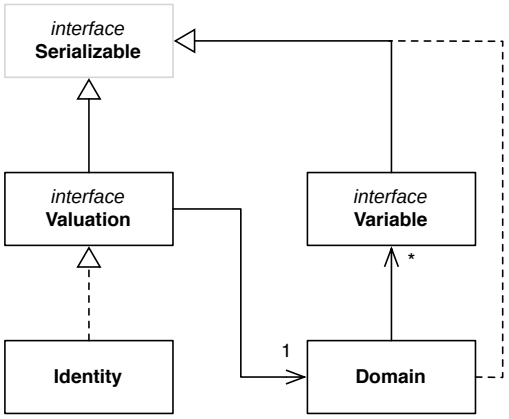


Figure 10.1: The core components of the algebraic framework.

Every user instance of the algebraic framework consists of at least two classes implementing the `Variable` and `Valuation` interface. We shall start our guided tour by discussing these two interfaces in more detail.

Developer Notes

- Use only serializable components within your implementation and remember that static fields are never serialized.

10.1.1 Variables

Although the `Variable` interface is by itself a marker interface, there are nevertheless some important remarks concerning its implementation. Every Java component is a descendant of the root component `java.lang.Object`, which provides a set of default methods that are frequently called. These methods need to be personalized:

- **public boolean** `equals(Object o)`;
  Mathematically, domains are sets of variables and therefore, variables need to be distinguishable. The default implementation of this equality check compares the memory address of the involved objects. This must be changed to an implementation based on the variable's name or identifier.

- **public int** `hashCode()`;
  We will see in a short while that NENOK represents domains as hash sets of `Variable` objects. In order to ensure their efficient and correct arrangement, the user should provide a reasonable hash code function. This can most easily be done by returning the hash code of the variable's name or identifier.

- **public** `String` `toString`();
  Domain objects are printed by successively calling this string conversion for variables. We therefore advise to the user to overwrite this method by a suitable representation of the current variable.

Developer Notes

- Make sure that your `Variable` implementation overwrites the `equals`, `hashCode` and `toString` methods inherited from `java.lang.Object`.

10.1.2 Domains

Domains are essentially sets of `Variable` objects. They are implemented as hash sets that offer constant time performance for most basic set operations such as `add`, `remove`, `contains` and `size`. As mentioned above, this requires that the `Variable` implementation is equipped with a hash function (of constant time performance) that disperses the elements properly among the buckets. Note also that `Domain` is an *immutable type* – once an object is created, it cannot be modified by any of the available methods.

10.1.3 Valuations

The `Valuation` interface is the lion's share of a valuation algebra implementation. Its source code is printed in Listing 10.1. The three methods at the head named `label`, `combine` and `marginalize` represent their mathematical namesakes defining a valuation algebra. (It is indicated that the `label` method does not directly belong to this interface but it is inherited from `nenok.adapt.Labeled`. This should not bother us for the moment.) The fourth method of the `Valuation` interface computes a valuation's weight. This will be important for the minimization of communication costs when performing distributed computations.

```
public interface Valuation extends Labeled {

  public Domain label();

  public Valuation combine(Valuation val);

  public Valuation marginalize(Domain dom) throws VAException;

  public int weight();
}
```

Listing 10.1: The `Valuation` interface.

- **public** `Domain label()`;
  The labeling method returns the domain of a valuation object.

- **public** `Valuation combine(Valuation val)`;
  It is indispensable that the combination of two valuations is implemented in such a way that its execution does not affect the two arguments. This is already a first source of danger in the design process of this algebraic framework. As mental exercise, assume the task of computing $(\phi \otimes \psi) \otimes \phi$. If by carelessness the factor ϕ is altered during the computation of $\phi \otimes \psi$, the final computation will for sure be defective. Another important point is that developers must be aware of identity elements that may be involved in the combination. We will see below, how identity elements are represented in NENOK but to give already an indication, the code of a possible implementation could start as follows:

```
public Valuation combine(Valuation val) {
  if(val instanceof Identity) {
    return this;
  }
  ⋮
}
```

- **public** `Valuation marginalize(Domain dom)` **throws** `VAException`;
  Marginalization of valuations is the third basic valuation algebra operation to implement. Again, there are some important points to remember in order to get a mathematically correct implementation. First, it must again be ensured that the computation of some marginal does not affect the original factor. Or imagine the result of $\phi \otimes \phi^{\downarrow x}$ if this advice has not been respected. Second, the method signature allows to throw a `VAException` standing for Valuation Algebra Exception. Throw such exceptions whenever marginalizations to illegal domains are tried. A possible implementation could therefore start as follows:

```
public Valuation marginalize(Domain dom) throws VAException {
  if(dom.equals(this.label())) {
    return this;
  }
  if(!dom.subSetOf(this.label())) {
    throw new VAException("Illegal Argument.");
  }
  ⋮
}
```

  Throwing a `VAException` in case of impossible marginalizations is also the suggested way to implement partial marginalization as introduced in Section 2.5.

- **public int** `weight()`;
  This method computes the weight of the current valuation according to Definition 5.2. NENOK will use this measure to estimate the communication costs

of transmitting valuations, which in turn will be important to go for efficient communication.

Beside the methods that are explicitly listed in the `Valuation` interface, we should also provide some methods to represent valuations on screen. We refer to Section 10.1.5 for a complete discussion of this topic and close by summarizing again the most important instructions related to the implementation of this interface.

Developer Notes

- Recall the special case of identity elements in the `combine` method.
- Avert marginalizations to illegal domains by throwing a `VAException`.
- Make sure that `combine` and `marginalize` do not alter their arguments.

10.1.4 Identity Elements

The `Identity` class is a default implementation of the `Valuation` interface in order to equip NENOK with identity elements. Because of its simplicity and to illustrate at least once a complete implementation, the code of this class is printed in Listing 10.2. There are several interesting points to retain from this source code: apart from the methods specified by the `Valuation` interface or inherited from `java.lang.Object`, the `Identity` class implements two additional interfaces of the algebraic framework, namely `Idempotency` which itself extends `Scalability`. They both extend the `Valuation` interface such that the latter does not need to be enumerated in the listing of implemented interfaces. The content of the two new interfaces will be topic of Section 10.2. We furthermore know that identity elements are mathematically unique which allows for an implementation based on the SINGLETON design pattern (Gamma *et al.*, 1993). The role of the `Representor` annotation will be explained in the following subsection.

10.1.5 Printing Valuation Objects

Naturally, a suitable output format to print valuation objects is a central point but the way of doing depends greatly on the implemented valuation algebra instance. Relations are traditionally represented in tabular form. The same holds for probability potentials, but a graphical diagram could be a valuable alternative for this formalism. For belief functions, there are perhaps two textual representations to be considered, one for the complete mass function and another one only for its focal sets. These varying requirements show that we can neither specify the number nor the signature of output methods in the `Valuation` interface. NENOK encounters this fact by exploiting annotations which require the abidance of the following rules:

1. Output methods have empty parameter lists.
2. Every output method is annotated with `nenok.va.Representator`.
3. Two possible return types are accepted: `java.lang.String` for pure textual output and `javax.swing.JComponent` for arbitrary graphical components.

```
public final class Identity implements Idempotency {

  public static final Identity INSTANCE = new Identity();

  private Identity() {}

  /**
   * Methods from Valuation:
   */

  public Domain label()  {
    return Domain.EMPTY;
  }

  public Valuation combine(Valuation val) {
    return val;
  }

  public Valuation marginalize(Domain dom) throws VAException {
    if(dom.size() == 0) {
      return INSTANCE;
    }
    throw new VAException("Illegal Domain.");
  }

  public int weight() {
    return 0;
  }

  /**
   * Methods from Object:
   */

  @Representor
  public String toString() {
    return "Identity Element";
  }

  /**
   * Methods from Idempotency & Scalability:
   */

  public Scalability scale() {
    return INSTANCE;
  }

  public Separativity inverse() {
    return INSTANCE;
  }
}
```

Listing 10.2: The `Identity` class.

Listing 10.2 annotates for example the `toString` method. With these arrangements, software projects can display valuation objects in a generic manner by invoking the annotated methods with the *Java Reflection* framework. The class `nenok.Utilities` offers a collection of pre-implemented methods that simplify this task considerably.

10.2 Extended Valuation Algebra Framework

The framework cutout discussed in the previous section is sufficient to implement a valuation algebra instance onto which the Shenoy-Shafer architecture can be applied. However, Chapter 2 introduced many extensions of this framework that have been exploited in Chapter 4 for the definition of more efficient local computation architectures. Another extension called *weight predictability* has been studied in Section 5.1 and turned out to be of prime importance for distributed computing. Realizing these properties upon the framework discussed so far is challenging. We must reflect the mathematical relationships between properties and allow to furnish valuation algebra instances with every reasonable combination of them. Figure 10.2 shows how the `Valuation` interface is accordingly refined.

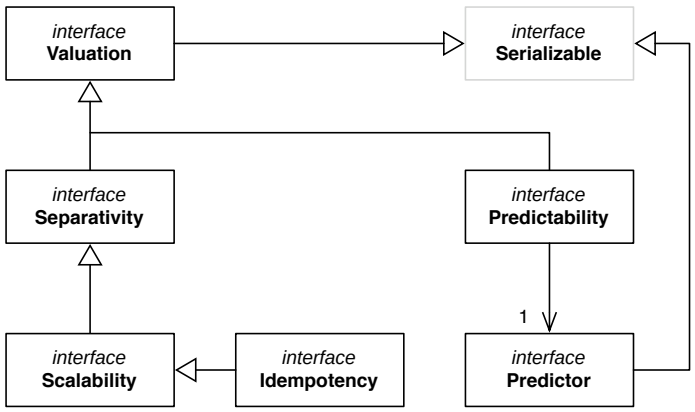


Figure 10.2: The extended valuation algebra framework.

10.2.1 Valuation Algebras with Division

The `Separativity` interface concentrates both separative and regular valuation algebras. There is in fact no mean for NENOK to mark some valuation algebras as regular, since this property has no special importance for local computation that goes beyond separativity. In short, the following method has to be implemented to equip a framework instance with a division operator:

- **public** `Separativity inverse`();
  Computes the inverse of the current valuation.

Developer Notes

- All important properties are reflected by NENOK interfaces.
- Implement `Separativity` to provide a division operator.

10.2.2 Scalable Valuation Algebras

We remarked in Section 2.7 that the existence of scaled valuations mainly depends on semantic reasons. Nevertheless, separativity is an algebraic requirement for this property. Seeing that, the `Scalability` interface must clearly extend `Separativity` as shown in Figure 10.2. Due to Equation (2.44), scaling is essentially a simple application of division and should therefore be pre-implemented, possibly by converting `Scalability` into an abstract class. However, this design proposition has severe drawbacks. It is well-known that Java does not support multiple inheritance. Thus, if `Scalability` had been implemented as an abstract class, we would implicitly hazard the consequences that a scalable instance cannot extend any other class. This is unacceptable, not only because there are other properties which allow the pre-implementation of some operators (see Section 10.2.3 for example). There exist a lot of design concepts to avoid multiple inheritance. In the case at hand, we plumped for an approach named *delegation* (Gamma *et al.*, 1993), which essentially equips every such interface with an inner class uniformly named `Implementor` that provides the pre-implementations. For better performance, these `Implementor` classes are realized as SINGLETON design patterns (Gamma *et al.*, 1993). Listing 10.3 contains the source code of the `Scalability` interface together with its `Implementor` class. The delegator method `scale` computes Equation (2.44).

Every instance that wants to use a certain method pre-implementation only calls the implementor object and uses its corresponding delegator method with the current object as argument. Multiple inheritance is circumvented since `Implementor` is not coupled with the algebraic framework. If any doubt still lurks in your mind, the following code snippet shows how to call the scaling delegator:

```
public Scalability scale() {
  return Scalability.Implementor.getInstance().scale(this);
}
```

Developer Notes

- Implement `Scalability` if the valuation algebra supports scaling.
- Delegator methods for default implementations are pooled in so-called `Implementor` classes in order to avert multiple inheritance dead ends.
- `Scalability` contains an implementor for its own `scale` method.

```
public interface Scalability extends Separativity {

  public Scalability scale();

  public class Implementor {

    private static Implementor implementor = new Implementor();

    private Implementor() {}

    public static Implementor getInstance() {
      return implementor;
    }

    public Scalability scale(Scalability scaler) {
      Scalability v = (Scalability)scaler.marginalize(Domain.EMPTY);
      v = (Scalability)v.inverse();
      return (Scalability)scaler.combine(v);
    }
  }
}
```

Listing 10.3: The `Scalability` interface.

10.2.3 Idempotent Valuation Algebras

Idempotency is a property that applies to the combination operator and that implicates trivial division and scaling. However, NENOK must be able to identify idempotent valuation algebras as a precondition for the application of the idempotent architecture of Section 4.7. The corresponding marker interface named `Idempotency` extends `Scalability` and provides an implementor class, which in turn offers the trivial pre-implementations for both methods `scale` and `inverse`. The corresponding source code is given in Listing 10.4. The attentive reader may rightly say that in this case, it is a lot more laborious to use the implementor class of `Idempotency` instead of directly programming the trivial implementations of `scale` and `inverse`. In fact, implementors should be seen as a service that simplifies the use of NENOK especially for those users that are not completely aware of the mathematical background. From this point of view, it is more important to provide a uniform access to pre-implementations than economizing some letters in the source code.

Developer Notes

- The interface `Idempotency` marks idempotent valuation algebras.
- `Idempotency` contains an implementor for the methods `scale` and `inverse`.

```
public interface Idempotency extends Scalability {

  public class Implementor {

    private static Implementor implementor = new Implementor();

    private Implementor() {}

    public static Implementor getInstance() {
      return implementor;
    }

    public Separativity inverse(Idempotency factor) {
      return factor;
    }

    public Scalability scale(Idempotency factor) {
      return factor;
    }
  }
}
```

Listing 10.4: The `Idempotency` interface.

10.2.4 Weight Predictable Valuation Algebras

Chapter 5 proves that the property of weight predictability is sufficient for a valuation algebra to minimize the communication costs caused by solving projection problems over distributed knowledgebases. Remember, weight predictability betokens the ability to compute a valuation's weight only by use of its domain. Although this concept seems rather simple, its realization turned out to be cumbersome due to some conflictive requirements. So far, all properties have been represented by appropriate Java types. To be consistent, this should also be the case for weight predictability. However, opposing is the requirement to compute a valuation's weight even before this object actually exists. In Java terms, this demands for a static implementation. But every valuation algebra possesses its own weight predictor which in turn contradicts the design based on a static method. This very short brain storming shall give an impression of this task's complexity. In order to meet the requirements for the most part, the delegator technique has again been applied. Weight predictable valuation algebras are mirrored in NENOK by the `Predictability` interface that contains the following method signature:

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

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

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

The design's only drawback is that we must dispose of at least one valuation object in order to ask for the weight predictor. But then, we can compute the weight of an arbitrary non-existing valuation using the `Predictor` object. We recommend to all trained Java programmers to implement the `Predictor` interface as a SINGLETON design pattern (Gamma *et al.*, 1993) in order to highlight that only one such object per valuation algebra needs to exist. The application of the delegator design strategy may seem complicated, but a closer inspection proves its value. First, each valuation algebra can possess its own weight predictor. Second, we can demand the predictor of an arbitrary object in order to compute the weight of any other instance of the same valuation algebra. Finally, weight predictable valuation algebras are realized by a new type which harmonizes with the implementation of other properties.

Provided that a valuation algebra is weight predictable, we can give a pre-implementation of the generic `weight` method inherited from the `Valuation` interface. It should not be surprising that `Predictability` contains an implementor class which offers this functionality via the following delegation:

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

Developer Notes

- Implement `Predictability` if the valuation algebra is weight predictable.
- The weight predictor itself is defined by the `Predictor` interface.
- `Predictability` contains an implementor for the inherited `weight` method.

10.3 Semiring Valuation Algebra Framework

Semiring valuation algebras, as introduced in Chapter 6, obey a very restrictive buildup, which enables us to implement generically most parts of the algebraic framework. This simplifies the embedding of own formalisms into the NENOK framework considerably. In fact, semiring valuation algebras only require a user implementation of semiring values and operations which generally is fewer and simpler work than the complete implementation of the `Valuation` interface. We outline in this section the components of this additional framework part, with the special focus on how they relate to the general framework part discussed in the foregoing sections. All NENOK components that will be addressed are contained in the package `nenok.va.sr`, except the class `FiniteVariable` that resides in `nenok.va`. A first picture of the semiring related components in NENOK is given in Figure 10.3.

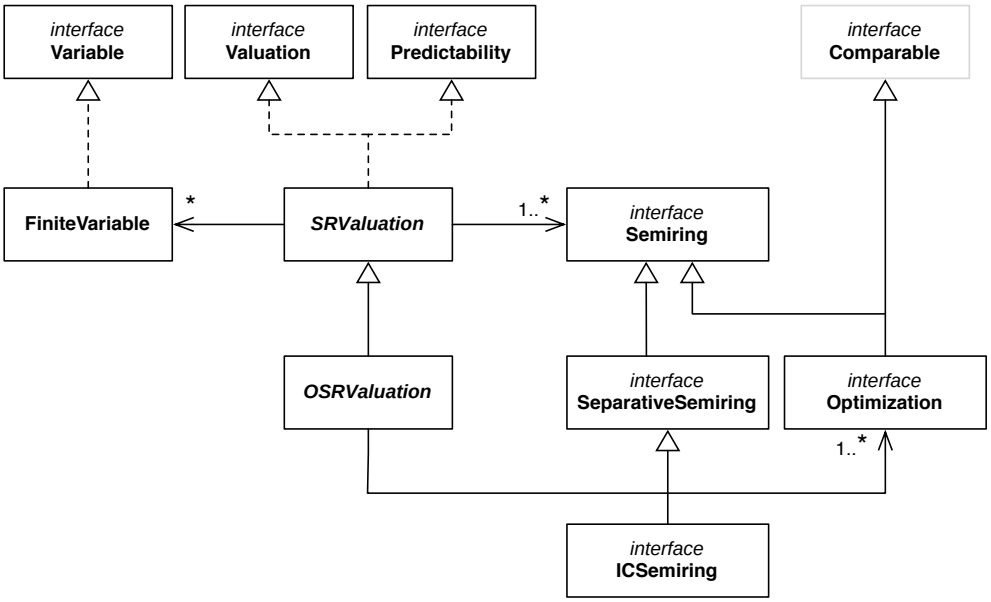


Figure 10.3: Semiring framework extension.

10.3.1 Finite Variables

In the context of semiring valuation algebras, frames of variables are always finite. NENOK offers a default implementation of such a variable type named `FiniteVariable` with the following constructor signature:

- **public** `FiniteVariable`(`String name`, `String`[] `frame`);
  Constructs a `Variable` instance from the variable's name and its frame values.

As we can see, variables are simply identified by a string argument which also affects the internal realization of the equality check. Furthermore, they take values out of the given string array. No additional work concerning the implementation of variables is required in case of semiring valuation algebras.

10.3.2 Semiring Elements

The second important component needed to derive a semiring valuation algebra is the implementation of the semiring itself. For this purpose, NENOK defines the `Semiring` interface which appoints the following two standard semiring operations:

- **public** `Semiring add`(`Semiring e`);
  Computes the addition of two semiring elements.

- **public** `Semiring multiply`(`Semiring e`);
  Computes the multiplication of two semiring elements.

Here, we meet a similar situation to Section 10.1.3, because the semiring axioms require both operations to be associative and commutative. Therefore, it must be ensure that neither of the involved factors will ever be modified during the execution of semiring operations. Besides, we need again to adapt the methods `equals` and `toString` inherited from `java.lang.Object` for an equality test and a suitable output format of semiring values.

Developer Notes

- Make sure that `add` and `multiply` do not alter their arguments.
- Use only serializable components within your implementation.
- Overwrite `equals` with an equality check for semiring elements.
- Overwrite `toString` for a suitable output format of semiring elements.

10.3.3 Semiring Valuations

It was already brought up multiple times that semiring valuation algebras qualify for a particular simple embedding into the NENOK framework. Indeed, a user implementation of the `Semiring` interface proves to be sufficient for this task. The mapping from configurations to semiring values is thereby realized within the abstract class `SRValuation`, as one might conclude from the signature of its default constructor:

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

This constructor always bears on the standard enumeration of the configuration set, which results from the ordering within the variable array. Doing so, the configuration set does not need to be produced explicitly, which provides a measurable increase of performance. The ordering of the semiring values within the second argument therefore refers to the configuration ordering implicitly defined by the variable array. This will be illustrated concretely in Section 10.4. If the number of semiring values does not match with the computed size of the configuration set, an `IllegalArgumentException` will be thrown. Further, this abstract class `SRValuation` implements all methods from the `Valuation` interface by leading the corresponding methods back to the semiring operations. Additionally, `SRValuation` implements the `Predictability` interface with a realization of the weight predictor from Example 5.1. All this is illustrated in the class diagram of Figure 10.3.

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

- **public abstract** `SRValuation create(FiniteVariable[] vars, Semiring[] values);`
  Factory method within `SRValuation`.

Its importance and functionality can best be understood by reflecting on a user implementation of a certain semiring valuation algebra. Thus, the user starts writing a class that extends `SRValuation` and that contains to the programmer's pride

a very sophisticated output method called `output`. If then a combination of two such instances is executed, the result will be of type `SRValuation` because the call of `combine` was delegated to the superclass `SRValuation`. Consequently, it is no longer possible to apply `output` on the resulting instance. This problem is addressed by the above factory method. Instead of directly calling a constructor, all methods within `SRValuation` that return new instances, build them by executing `create`. Thus, if the user delegates `create` to its own class constructor, the above problem ends in smoke.

Developer Notes

- Redirect `create` to your own class constructor.

10.3.4 Semiring Valuation Algebras with Division

The interfaces of the algebraic layer which are used to endue some implementation with mathematical properties have been discussed at the beginning of this chapter. Therefore, no further specialization of the `SRValuation` class is needed here. The user simply extends `SRValuation` and implements the corresponding interfaces with respect to the properties of its semiring valuation algebra. However, some more work is required for the implementation of division. As learned in Section 6.6, division for semiring valuations presupposes inverse semiring elements. Therefore, the interface `SeparativeSemiring` extends `Semiring` with a method to compute inverse elements:

- **public** `SeparativeSemiring inverse`();
  Returns the inverse of the current semiring element.

This interface roofs all kinds of semirings that include some notion of division. In order to realize a separative semiring valuation algebra, it is then sufficient to extend `SRValuation` in the usual way and additionally, to implement the `Separativity` interface of Section 10.2.1. A pre-implementation of the `inverse` method from `Separativity` is once more offered by its `Implementor` class, accessible in the familiar way:

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

Developer Notes

- Semiring valuations with division must implement `Separativity`.
- They require values of type `SeparativeSemiring`.
- `Separativity` contains then an implementor of its own `inverse` method that can be used for semiring valuation algebras.

10.3.5 Semiring Valuation Algebras for Optimization Problems

According to Definition 8.1, totally ordered, idempotent semirings turn projection problems into optimization tasks. This assembly of semiring attributes is represented

in NENOK by the interface `Optimization` that extends `Semiring` and the Java interface `Comparable` for the total ordering. Since idempotency only refers to the behaviour of addition, the amount of work for the implementation of this interface is confined to the comparison method `compareTo` inherited from `Comparable`. However, the natural order between elements of an idempotent semiring is defined by Equation (6.2) and can be realized in a generic way, provided that the programmer followed the instructions of Section 10.3.2 and equipped its `Semiring` instantiation with an equality check. Then, the implementor of `Optimization` can be used in the usual way for the delegation to the default implementation:

```
public int compareTo(Object o) {
  Optimization arg = (Optimization)o;
  return Optimization.Implementor.getInstance().compareTo(arg, this);
}
```

For the implementation of the valuation algebra itself, we then use the abstract class `OSRValuation` extending `SRValuation`. Besides the main difference that this class accepts only semiring values of type `Optimization`, it provides a bundle of implemented methods that will later support the dynamic programming functionalities.

Developer Notes

- Objects of type `OSRValuation` are used to model optimization problems.
- They require values of type `Optimization`.
- `Optimization` contains an implementor for the method `compareTo`.

10.3.6 Idempotent Semiring Valuation Algebras

Theorem 6.16 states that a c-semiring with idempotent multiplication is sufficient to induce an idempotent valuation algebra. This restriction is reflected in NENOK by the `ICSemiring` marker interface which extends `SeparativeSemiring`. This interface provides once more an `Implementor` class with the default implementation of the inherited `inverse` method from `SeparativeSemiring`. Then, it remains to follow the procedure described in Section 10.2.3 to complete the implementation of an idempotent semiring valuation algebra.

Developer Notes

- Idempotent semiring valuations must implement `Idempotency`.
- They require values of type `CSemiring`.
- `CSemiring` contains an implementor for `SeparativeSemiring`.

This closes the discussion of the algebraic layer. We are aware of the fact that some design decisions presented here may still be very elusive. For this reason, the user shall dare to tackle its own implementation of a valuation algebra instance.

Most of the concepts will quickly seem natural. The next section is dedicated to such an implementation of the semiring valuation algebra framework. This example will attend our studies through the remaining chapters of this paper and illustrate future introduced concepts whenever possible.

10.4 Case Study: Probability Potentials

Probability potentials have been introduced as a valuation algebra instance in Section 3.4, and it was later shown in Section 6.7.1 that the same formalism can be derived from the semiring viewpoint. They are well suited to play the leading part in this NENOK tutorial, since they are easy to understand and provide sufficient interesting properties such as weight predictability, regularity and scalability. Here, we describe their realization using the semiring framework and refer to (Pouly, 2006) where an alternative, semiring-free implementation was shown.

Probability potentials are induced by the arithmetic semiring of Section 6.2.1. It is therefore sensible to choose this component as the starting point for our implementation. The arithmetic semiring is regular which compels our class `Arithmetic` to implement `SeparativeSemiring`. Below, we show a cutout of the corresponding implementation, omitting less interesting parts like the equality check or output methods.

```
public class Arithmetic implements SeparativeSemiring {

    private double value;

    public Arithmetic(double value) {
      if(value < 0)
        throw new IllegalArgumentException("Illegal Argument.");
      this.value = value;
    }

    public Semiring add(Semiring semiring) {
        if(!(semiring instanceof Arithmetic))
            throw new IllegalArgumentException("Illegal Argument.");
        return new Arithmetic(value + ((Arithmetic)semiring).value);
    }

    public Semiring multiply(Semiring semiring) {
        if(!(semiring instanceof Arithmetic))
            throw new IllegalArgumentException("Illegal Argument.");
        return new Arithmetic(value * ((Arithmetic)semiring).value);
    }

    public SeparativeSemiring inverse() {
        return (this.value == 0)? new Arithmetic(0) :
            new Arithmetic(1 / value);
    }
    ⋮
}
```

Slightly more complicated is the implementation of the actual valuation algebra class. We create a class named `Potential` that extends `SRValuation`. Since probability potentials are regular and scalable, we also implement the `Scalability` interface (this is sufficient since `Scalability` extends `Separativity`). This is again illustrated with a cutout of the corresponding implementation:

```
public class Potential extends SRValuation implements Scalability {

  private Potential(FiniteVariable[] vars, Arithmetic[] values) {
    super(vars, values);
  }

  public Potential(FiniteVariable var, double[] values) {
    super(new FiniteVariable[] {var}, convert(values));
  }

  public SRValuation create(FiniteVariable[] vars, Semiring[] values) {
    return new Potential(vars, cast(values));
  }

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

  public Scalability scale() {
    return Scalability.Implementor.getInstance().scale(this);
  }
  ⋮
}
```

The first component is a private constructor that calls the standard constructor of the superclass `SRValuation`. Then, the second constructor allows to specify probability potentials from a single variable and an array of probability values. The helper method `convert` transforms the probability values into objects of type `Arithmetic`. In a very similar way, this class offers a further constructor that enables the creation of conditional probability table. This code is omitted in the above listing to simplify matters, but its signature will be given at the very beginning of the example just below. More absorbing is the third component that implements the only abstract method of `SRValuation`. As explained in Section 10.3.3, it ensures that all returned instances of inherited methods are of type `Potential`. The helper method `cast` is used to guarantee type conformance. The two remaining components `inverse` and `scale` implement the `Scalability` interface by dint of the corresponding implementor classes. Besides some helper and output methods that are also omitted in this code snippet, this is all we have to do in order to equip NENOK with probability potentials. Eventually, Figure 10.4 summarizes how the probability potential implementation is connected to the algebraic framework. The two grey colored classes are the only components which need to be contributed by the programmer.

To close this section about a concrete semiring valuation algebra realization, we will finally model the Bayesian network example of Section 4.2.1 using our imple-

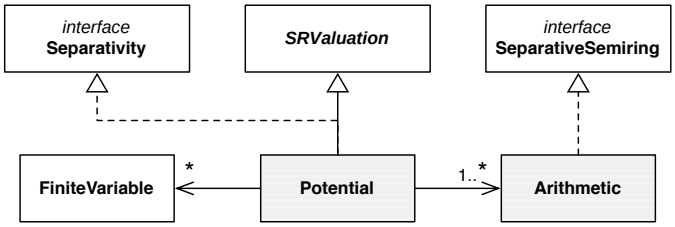


Figure 10.4: Connection of a user implementation with the algebraic layer.

mentation. Partially skipped in the above code cutout, the class `Potential` contains two public constructors that free the user from the necessity to define `Semiring` objects explicitly. Instead, we can use numerical values directly. The signatures of these constructors are:

- **public** `Potential`(`FiniteVariable var`, **double**... `values`);
  Builds a marginal probability potential for the given variable.
- **public** `Potential`(`FiniteVariable var`, `FiniteVariable`[] `cond`, **double**... `values`);
  Builds a conditional probability potential for the first variable, given an array of conditionals.

Internally, both constructors convert the probability values into `Arithmetic` objects as shown in the above listing by example of the constructor for marginal potentials.

The following code performs the computation of the ELISA test example from Section 4.2.1:

```
// Variable frames:
String[] frame1 = new String[] {"false", "true"};
String[] frame2 = new String[] {"negative", "positive"};

// Variables
FiniteVariable hiv = new FiniteVariable("HIV", frame1);
FiniteVariable test = new FiniteVariable("Test", frame2);

// Probability potentials:
Potential p1 = new Potential(hiv, 0.997, 0.003);
Potential p2 = new Potential(test,
  new FiniteVariable[] {hiv}, 0.98, 0.01, 0.02, 0.99);

// Solve first projection problem:
Valuation p = p1.combine(p2);
System.out.println(p);

// Solve second projection problem:
Domain query = new Domain(test);
Valuation result = p.marginalize(query);
System.out.println(result);
```

The code prints the two potentials needed for Equation (4.3):

HIV	Test	Value
false	negative	0.977
false	positive	0.020
true	negative	0.000
true	positive	0.003

Test	Value
negative	0.977
positive	0.023

This exercise should give a first picture of how to deal with valuation algebra operations. Significantly, all valuations used for the computation have been constructed explicitly in this example. Alternatively, we may also get these valuations from other processors which leads us to the presentation of the remote computing layer.

11
The Remote Computing Layer

It was already mentioned in the introduction of this implementation part that the ability of NENOK to process valuations on remote computers is realized with the Jini 2.0 framework. There is actually no need for the user to be familiar with every detail of the Jini framework, but nevertheless, some key concepts should be known in order to understand how NENOK works and what measures must be taken to exploit all its possibilities. We therefore start this chapter about the remote computing layer with a short and cursory introduction to the most important concepts of the Jini architecture, with a special focus on our necessities. The figures in this introduction are for the most part inspired by (Newmarch, 2006).

11.1 Jini in a Nutshell

Jini is a distributed computing environment provided by Sun Microsystems that offers network *plug and play*. In this environment, a device or a software service can connect to an existing network and announce its presence. Clients that wish to use this new service can locate it and call its functionality. New capabilities can be added to a running service without disrupting or reconfiguring it. Additionally, services can announce changes of their state to clients that currently use this service. This is, in a few words, the principal idea behind Jini.

A Jini *system* or *federation* is a collection of clients and services that communicate using the Jini protocol. Basically, there are three main players involved: services, clients and the *lookup service* that acts as a broker between the former two. This is illustrated in Figure 11.1.

11.1.1 Jini Services

Services are logical concepts commonly defined by a Java interface. A *service provider* disposes of an implementation of the appropriate interface and creates *service objects* by instantiating this implementation. Then, the service provider contacts the lookup service in order to register its service object. This is done either directly by a unicast TCP connection or by UDP multicast requests. In both

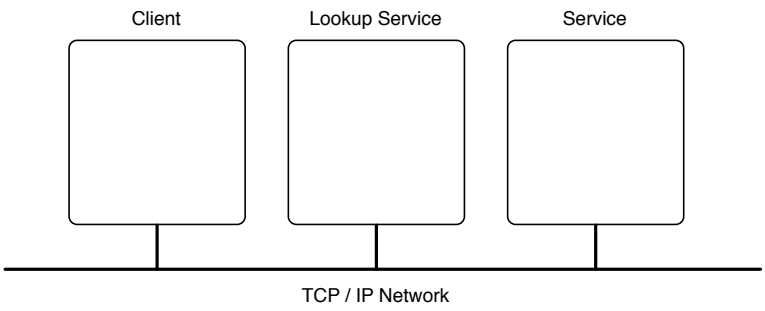


Figure 11.1: The three main players in a Jini system.

cases, the lookup service will answer by sending a *registrar object* back to the service provider. This object now acts as a proxy to the lookup service, and any requests that the service provider needs to make of the lookup service are made through this registrar. Figure 11.2 illustrates this first handshake between service provider and lookup service.

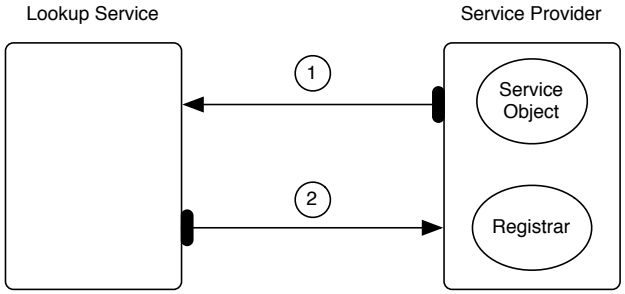


Figure 11.2: Contacting a lookup service is a two-step procedure: 1.) The lookup service is located. 2.) A registrar proxy of the lookup service is stored on the service provider. All further calls to the lookup service are made through this proxy.

The service provider registers its service using the registrar proxy. This means essentially that a copy of the service object is taken and stored on the lookup service as shown in Figure 11.3. The service is now available for the clients within the Jini federation.

11.1.2 Jini Clients

The necessary procedure for clients in order to use an exported service is almost a mirror image of Figure 11.2. They contact the lookup service and get a registrar object in response. Then, the client asks the lookup service through the registrar

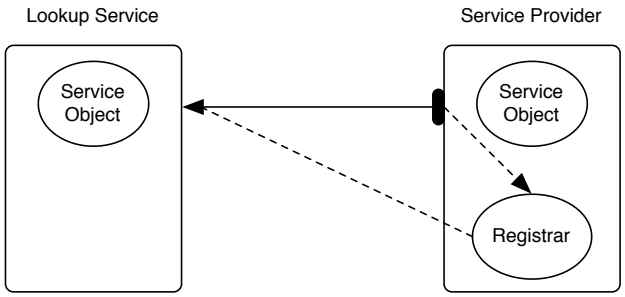


Figure 11.3: Registering a service object in a lookup service. The dashed arrows indicate that this is actually done through the registrar proxy.

proxy for an appropriate service object that implements the needed functionality. A copy of the service object is then sent to the client and the service is available within the client's virtual machine. Figures 11.4 and 11.5 illustrate how clients request services.

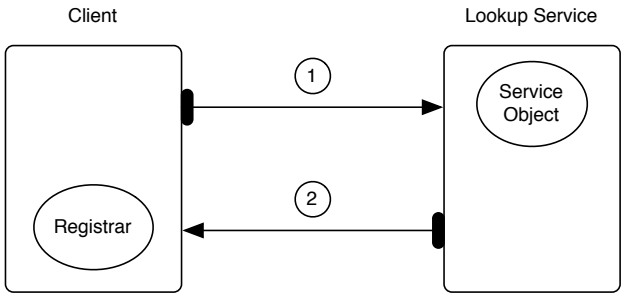


Figure 11.4: The procedure of contacting a lookup service is essentially the same for both clients and service providers.

Until now, we implicitly assumed that the service is built from a single, lightweight object. This object is exported to a lookup service and transmitted afterwards to all clients interested in its functionality. For sophisticated applications, however, this procedure is hardly suitable. Especially services designed to control hardware should be local on the corresponding machine, not to mention colossal database services being transmitted over the network. Instead of a single service object, we prefer to have at least two, one lightweight proxy object running in the client and another distinct one running in the service provider. This is sketched in Figure 11.6. The client uses the service by calling the service proxy, which in turn communicates with its implementation on the service provider. This additional communication is drawn in Figure 11.6 as a dashed double-headed arrow.

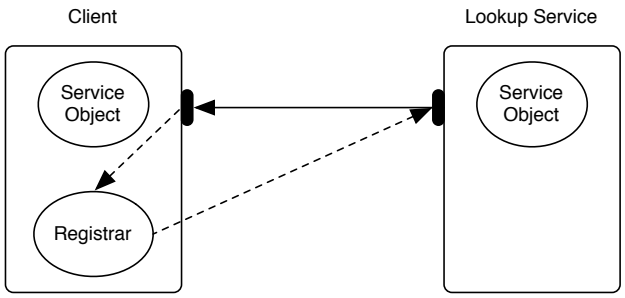


Figure 11.5: Through the proxy, the client asks the lookup service for a service object. This object is transmitted afterwards to the client's virtual machine.

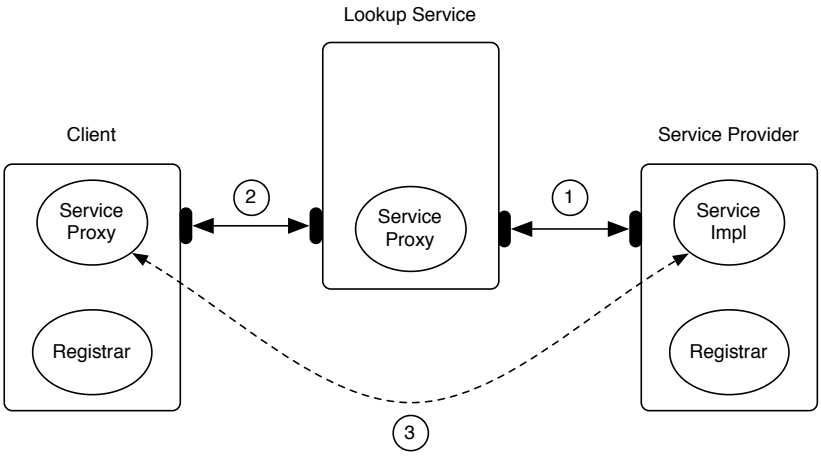


Figure 11.6: In order to prevent the transmission of huge service objects, they are divided up into a service proxy and a service implementation. The proxy is transmitted over the network and communicates with its implementation that is local on the service provider's virtual machine. 1.) A service proxy is created and uploaded to the lookup service. 2.) The service proxy is downloaded by the client. 3.) The proxy communicates with its service implementation within the virtual machine of the service provider.

11.1.3 Support Services

We have seen so far that Jini relies heavily on the ability to move objects from one virtual machine to another. This is done by using the Java *serialization* technique. Essentially, a snapshot of the object's current state is taken and serialized to a sequence of bytes. This serialized snapshot is moved around and brought back to life in the target virtual machine. Obviously, an object consists of code and data, which must both be present to reconstruct the object on a remote machine. The data part is represented by the object snapshot, but we cannot assume that every target machine has the object's class code. Every Jini federation therefore encloses a HTTP server from which the necessary code can be downloaded. This is shown in Figure 11.7, and it is, by the way, one of the main reasons for Jini's flexibility. Because the code is fetched from a remote server, one can add new services or even modify existing services at any time without relaunching the whole federation. Generally, every Jini service is divided up into two code archives (JAR files). The first contains all code that is required to start the service and is typically declared in the classpath when the service is launched. The second, by convention labeled with the postfix *dl*, contains only the source of the service that is used for reconstruction. Every Jini service has the `java.rmi.server.codebase` property set to the URL of the HTTP server that hosts its reconstruction code. This property is assigned to the serialized service whenever it needs to be transmitted over the network. On the receiving site, the reconstruction file is loaded from the appropriate HTTP server, and together with the serialized data the object can be brought to life again. This technique is generally known as *dynamic class loading*.

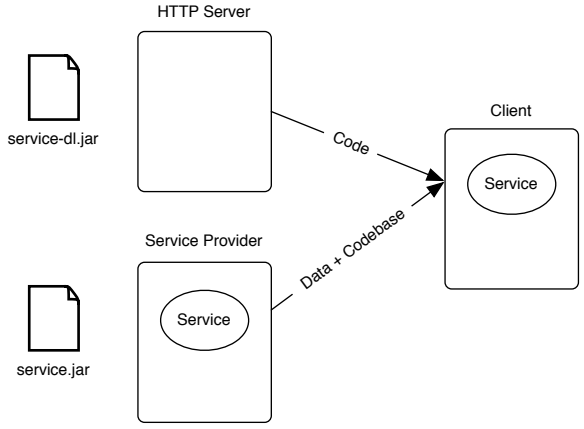


Figure 11.7: The file *service.jar* is used to start a new service instance. In order to transmit the service object to the client, its data is serialized and the URL of the HTTP server is attached. The client deserializes the service object by downloading the reconstruction code contained in *service-dl.jar* from the HTTP server, whose address is known from the codebase.

The second essential support service, the lookup service, is a real Jini service by itself. We discussed above its role as a broker between Jini services and clients. Therefore, each Jini federation must absolutely have access to a lookup service. Sun supplies a lookup service implementation called *Reggie* as part of the standard Jini distribution. How to start these services is explained in Section 11.3.

11.2 Computing with Remote Valuations

Chapter 5 examined local computation on distributed knowledgebases and identified *processor networks* as the central concept for communication. The processors themselves have been considered as independent computing units with their own memory space that can execute tasks without interactions and in parallel. NENOK adopts this fundamental idea and implements processors as ordinary Java threads based on the underlying Jini communication layer. Thus, we can build arbitrary processor networks by starting processor instances anywhere in the network, or alternatively, we can simulate such computing environments on a single machine. This section gives the needed background to set up a processor network, and we will explain how to compute with remote valuation objects. For simplicity, we assume throughout the following sections that the network we are working on allows multicast requests to contact the lookup service. If this is not the case, there are dual versions of all related methods presented here that essentially take the URL of the lookup service as second argument and perform unicast requests.

11.2.1 Processor Networks

A NENOK processor provides some sort of public memory space, comparable to the popular JavaSpace service, which is itself based on the Jini framework. However, NENOK processors are more powerful in the sense that they are real computing units. On the other hand, these processors can only store `Valuation` objects, and the current implementation does not contain any security or transaction measures. This picture of processors as public valuation storages is visualized in Figure 11.8, where three clients publish valuation objects into the memory space (cloud) of two NENOK processors.

The utility class `nenok.Services` provides a bundle of static methods that are all related to service administration. Among them, there is a method to start a new processor instance on the current machine:

```
Processor proc = Services.startProcessor();
```

There are some necessary preparation steps which must be taken to start processors. First, a HTTP server and Reggie must both run somewhere in the network. We will give a tutorial on how to set up these two services in Section 11.3. Additionally, the computer must dispose of a host name. If this is not the case, the above service method will throw an `UnknownHostException`. From a valid host name and the system time of the current computer, one can produce identifiers that are globally unique

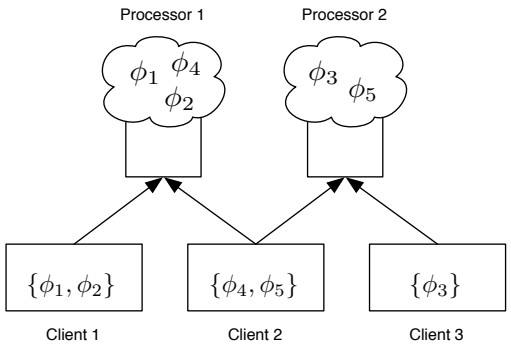


Figure 11.8: Publishing valuation objects into a processor's memory space.

within the network. Such an identifier is also assigned to every processor instance at start time in order to localize it at a later date. Because identifiers play an important role in all kinds of distributed systems, Jini provides a class named `net.jini.id.Uuid` for this task. We can request the identifier of a processor as follows:

```
Uuid pid = proc.getPID();
```

Once the identifier of a running processor is known, localizing this processor becomes a simple task by using the appropriate static method of the service class:

```
Processor proc = Services.findProcessor(pid);
```

Thus, it is possible to request the proxy of any processor that is running somewhere in the network, if we know this processor's identifier.

To bring this short description of a processor's life cycle to a close, we point out that the service class also contains methods for destroying processor instances. Calling this method will erase the affected processor's memory space and finally terminate its thread:

```
Services.destroyProcessor(pid);
```

11.2.2 Remote Valuations

The next step after starting a NENOK processor is naturally the export of valuation objects into its shared memory space. The related procedure in the `Processor` class works as follows: First, the valuation object is serialized into a byte stream and transferred to the processor's service object, which rebuilds the valuation object in its own memory space and wraps it into an `Envelope` object. These wrappers serve for addressing purposes and equip each valuation object with a unique identifier. Finally, the service object creates a so-called `Locator` object which is returned to the client. This locator contains among other things the processor and envelope

identifiers. To give a real life analogy, locators are the tickets we get from the cloakroom attendants at the concert hall which later allow us to retrieve our clothes. There are two methods in the `Processor` interface that allow to export valuations:

- **public** `Locator store(Valuation val)` **throws** `RemoteException`;
  Exports the given valuation to the processor's memory space.
- **public** `Locator[] store(Valuation[] vals)` **throws** `RemoteException`;
  Exports an array of valuations to the processor's memory space.

Both methods throw `RemoteException` to indicate network problems.

The returned locators play a decisive role. They are essentially concatenations of the processor's ID and the envelope identifier that encloses the exported valuation within the processor's memory space. Therefore, by use of a locator object, we are able to address unambiguously the corresponding object in the processor's storage. Moreover, the necessary functionality for retrieving an exported valuation object is offered directly by the `Locator` class itself:

- **public** `Valuation retrieve()` **throws** `RemoteException`;
  Retrieves the valuation that is referenced by the current locator.

Note also that this call does not remove the valuation from the processor's memory.

Locators are very lightweight components which do not depend on the valuation's actual size. Therefore, it is much more efficient to transmit locators instead of their corresponding valuation objects. The sequence diagram of Figure 11.9 reviews this sequence of storing and retrieving valuation objects. To conclude, it remains to be said that these methods are the reason why all components of the algebraic layer need to be fully serializable.

11.2.3 Remote Computing

Being aware that we do not yet know how exactly NENOK clients exchange locators, we nevertheless assume that a certain client possesses some locators pointing to valuations somewhere in the network. These valuations are now to be used for computations. Clearly, one possibility is to retrieve all valuations and to start the computations locally. This works fine for some elementary computations, but alternatively, we can also delegate the computations to the processors that already hold the corresponding factors. The `Processor` interface contains the following three methods for this purpose:

- **public** `Locator combine(Locator loc, Locator... locs)` **throws** `RemoteException`;
  Computes the combination of all valuations referenced by the locators.
- **public** `Locator marginalize(Locator loc, Domain dom)` **throws** `RemoteException`;
  Computes the marginal of the referenced valuation to the given domain.
- **public** `Locator inverse(Locator loc)` **throws** `RemoteException`;
  Computes the inverse of the referenced valuation.

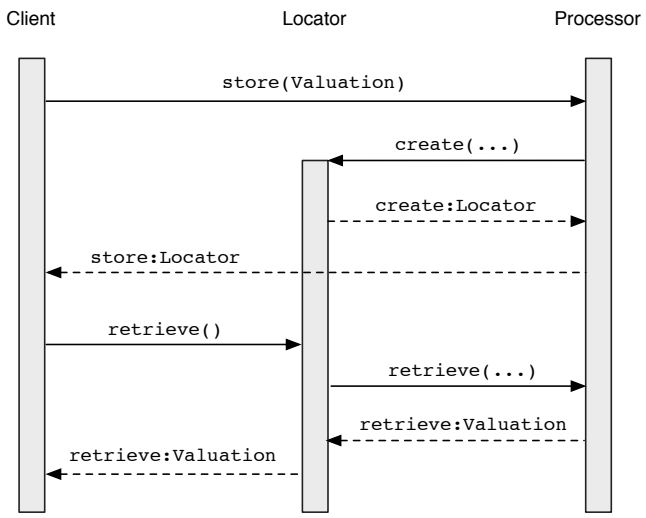


Figure 11.9: Storing and retrieving valuation objects.

All these methods return locator objects that point to the result of the corresponding computation. This facilitates the execution of sequences of valuation algebra operations considerably.

From the perspective of communication costs, we conclude for example that computing a marginal of some valuation that resides on the executing processor is cheap, since no communication costs are caused. This awareness will later be important for the minimization of communication costs during local computation.

11.2.4 Knowledgebase Registry

We have seen so far how NENOK clients can publish valuation objects into the shared memory space of a processor, how these objects can be retrieved at a later moment, and how remote valuation algebra operations are executed. However, all these activities presuppose that the initiating client has the locator objects, even if it has not stored the valuations on the processors itself. To make a long story short, we need a way to exchange locators between NENOK clients. This feature is provided by an additional Jini service called *knowledgebase registry* that basically realizes the mathematical idea of distributed knowledgebases.

Let us go through a typical scenario. Some client initially contacts the knowledgebase registry service and creates a new knowledgebase. Then, it stores the designated locators within this knowledgebase and aborts the registry service. Other clients within the same federation can also contribute to this knowledgebase. They contact the registry service by themselves, ask for the knowledgebase created by the first client and upload their own locators to this knowledgebase. Alternatively,

all clients can at any time download the knowledgebase. Doing so, they possess all locators within the knowledgebase and are now able to retrieve the corresponding valuation objects.

The following code snippets constitute an exchange of locator objects, continuing the ELISA test of Section 10.4. We first start a processor and store the two potentials. Then, the knowledgebase registry is contacted and a new knowledgebase named *Elisa* is created. Both locators are uploaded to this knowledgebase.

```
Processor proc = Services.startProcessor();
Locator loc1 = proc.store(p1);
Locator loc2 = proc.store(p2);

Registry registry = Services.findRegistry();
registry.createKnowledgebase("Elisa");

registry.add("Elisa", loc1);
registry.add("Elisa", loc2);
```

This completes the activities of the first client. Later, another client contacts the registry service and asks for the ELISA knowledgebase. The registry returns a `Knowledgebase` object that contains the locators for the two potentials originally created by the first client. Conveniently, the `Knowledgebase` interface directly offers a method to retrieve the valuations of all its locators, as shown subsequently.

```
Registry registry = Services.findRegistry();
Knowledgebase kb = registry.getKnowledgebase("Elisa");

Valuation[] vals = kb.getValuations();
```

11.2.5 Knowledgebases

In the above code extract, the registry service returned a `Knowledgebase` object to the client. Because these objects will later serve as input data for all local computation features, it is worth taking a closer look at this interface. A central promise of NENOK is that it does not distinguish between local and remote data, and the `Knowledgebase` interface is the essential component for this abstraction. The class diagram of Figure 11.10 shows that two different implementations for this interface exist. `RemoteKB` is a container for `Locator` objects and this is in fact the kind of knowledgebase that was returned in the above example. Thus, we conclude that shipping such a knowledgebase through the network is comparatively cheap because it includes only lightweight locator objects. Alternatively, `LocalKB` holds valuation objects directly, and this kind of knowledgebase is provided for a purely local use of the NENOK framework. In contrast to the first type, users can create such knowledgebases directly using a factory method contained in the `Knowledgebase` interface:

- **public static** `Knowledgebase create(Valuation[] vals, String name);`
  Creates a knowledgebase from local data.

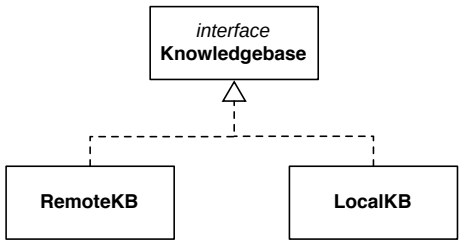


Figure 11.10: Knowledgebase implementations for local and remote factors.

A particularly important point that users should be aware of is that knowledgebases only contain valuations of the same type, or locators pointing to the same type of valuations respectively. It is therefore impossible to mix valuation objects of different algebra instances in the same knowledgebase. The only exception to this rule is the `Identity` object which can always be added. However, because locators are only pointers to remote valuations, this type checking cannot be done at compile time. So, the first element inserted determines the type that is accepted by a certain knowledgebase, and an `IllegalArgumentException` is thrown when this policy is violated at a particular time.

11.3 Setting up a NENOK Federation

The complex configuration work needed to run a Jini environment exasperates users regularly. Therefore, a major goal in the development of NENOK was to relieve the user as much as possible from this arduous task by offering preconfigured start scripts for all required Jini services. This section explains first how the two necessary support services are set up. These are the HTTP server and the lookup service called Reggie, as we know from Section 11.1. Because Reggie is by itself a Jini service that depends on a running HTTP server, the order to start these Jini components is strictly determined. As a third and last component, it will be shown how to launch the knowledgebase registry service from Section 11.2.4. NENOK provides all necessary start scripts in its *startup* folder. Nevertheless, some minor configuration work is indispensable and subsequently described. It should be clear that setting up a NENOK federation is only required if we indeed plan to perform distributed computations. Otherwise, this section can confidently be skipped.

11.3.1 HTTP Server

The standard Jini distribution provides a simple HTTP server implementation which is sufficient for our purposes. It is started by the following ANT script named *classserver.xml*:

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="classserver" default="run">
  <target name="run">
    <java jar="../jini/start.jar" fork="true">
      <arg value="start-classserver.config" />
      <sysproperty key="port" value="8081" />
      <sysproperty key="java.security.policy" value="../policy.all" />
    </java>
  </target>
</project>
```

Security is of course an important topic in every distributed system, and Jini offers generic measures to protect arbitrary services. However, remembering spirit and purpose of NENOK allows us here to ignore this aspect, and we grant all permissions to our services. More important is the specification of the NENOK port which is by default set to 8081. We recommend to not modify it unless another application on your system already uses this port.

11.3.2 Reggie

Reggie is the default implementation for a lookup service and is the second pillar of a Jini environment. Because Reggie is a real Jini service by itself, it must know the address of the HTTP server that hosts its reconstruction code. This is the `codebase` which must point to the currently running HTTP server. Modify the following ANT script named *reggie.xml* in such a way that the IP address corresponds to the machine running the HTTP server. If the port number in the HTTP server start script has been changed, the corresponding value must also be adapted.

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="reggie" default="run">
  <target name="run">
    <java jar="../jini/start.jar" fork="true">
      <arg value="start-transient-reggie.config" />
      <sysproperty key="codebase" value="http://134.21.73.91:8081/" />
      <sysproperty key="java.security.policy" value="../policy.all" />
    </java>
  </target>
</project>
```

11.3.3 Knowledgebase Registry

Once the Jini environment is running, we can finally start the knowledgebase registry using the ANT script *registry.xml*. Comparable to Reggie, this service is global to the NENOK federation and must therefore be started manually. Again, we need to ensure that the codebase refers to the running HTTP server and that the port number is specified consistently.

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="registry" default="run">
  <target name="run">
    <echo message="Nenok Registry ..." />
```

```
    <java jar="../jini/start.jar" fork="true">
      <arg value="start-transient-registry.config" />
      <sysproperty key="codebase" value="http://134.21.73.91:8081/" />
      <sysproperty key="java.security.policy" value="../policy.all" />
    </java>
  </target>
</project>
```

11.3.4 Starting a NENOK Application

A running HTTP server together with Reggie and a knowledgebase registry constitutes a complete NENOK environment for remote computing applications. We next play over the start procedure of a simple application. Imaging a compiled file *ElisaUpload.class* that creates the two probability potentials of the ELISA test example and uploads them to the knowledgebase registry service. This essentially corresponds to the first code snippet in Section 11.2.4. Here is a possible ANT script to launch such an application:

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="elisa" default="run">
  <target name="run">
    <java classname="ElisaUpload" fork="true">
      <classpath>
        <pathelement location="lib/nenok.jar"/>
        <pathelement location="ppotentials/ppotentials.jar"/>
      </classpath>
      <sysproperty key="java.security.policy" value="policy.all" />
      <sysproperty key="java.rmi.server.codebase"
       value="http://134.21.73.91:8081/ppotentials/ppotentials.jar" />
    </java>
  </target>
</project>
```

We first remark that two archives have to be added to the classpath, namely *nenok.jar* that contains the generic NENOK architecture and *ppotentials.jar* with the probability potential user implementation. Since the created objects are uploaded to the registry service, we also need to specify the codebase with the reconstruction code for client applications as described in Section 11.1.3.

Then, we assume a second application *ElisaDownload.class* that downloads the ELISA knowledgebase from the registry service according to the second code snippet at the end of Section 11.2.4. We again give an ANT script to launch this application:

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="elisa" default="run">
  <target name="run">
    <java classname="ElisaDownload" fork="true">
      <classpath>
        <pathelement location="lib/nenok.jar"/>
      </classpath>
```

```
    </java>
  </target>
</project>
```

For this client application, it is sufficient to add `nenok.jar` to the classpath. Later, if computations are performed with the probability potential objects that are referenced by the locators contained in the downloaded knowledgebase, the application fetches the reconstruction code from the attached codebase to deserialize the probability potential objects. See Section 11.1.3 for more details. This scenario illustrates once more the power and flexibility of a Jini service.

12
The Local Computation Layer

We have now arrived at the top of the layer model and describe how local computation techniques can be applied to any given valuation algebra implementation. As a whole, there are many components involved such as the various architectures of local computation or join tree construction algorithms that allow to parametrize NENOK by some means or another. But this multiplicity of possibilities naturally runs the risk of overstraining users that simply want to perform some basic computations. Therefore, the guiding idea of NENOK is to provide a fully configured system, and to allow changing these values using a simple interface in case of special requirements. This level of abstraction is realized by an important component named *local computation factory*, which marks the starting point for our expedition through the framework's local computation facilities. Every example in this section will start from a `Knowledgebase` object that has either been downloaded from a knowledgebase registry or created locally using the appropriate factory method from Section 11.2.5.

12.1 Local Computation Factory

The class `LCFactory` mirrors the concept of a local computation factory that provides all important local computation facilities. It contains an empty constructor with a default configuration for the execution of the Shenoy-Shafer architecture. Consequently, three lines of code are sufficient to perform a complete run of the Shenoy-Shafer architecture onto a given `Knowledgebase` object named `kb`:

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

Here, the local computation factory builds a join tree that covers all factors within the given knowledgebase. For more practical applications, however, we also need to ensure that additional queries, which do not necessarily correspond to some factor domain, are covered by the join tree. This requirement is reflected within the signature of the applied creator method:

- **public** `JoinTree create(Knowledgebase kb, Domain... queries)` **throws** `ConstrException;`
  Creates a join tree from a given knowledgebase that covers the specified queries.

NENOK communicates problems that occur during the join tree construction process by throwing a `ConstrException`. Further creators, that allow for example to construct join trees from file input, will be discussed in Chapter 13.

12.1.1 Changing the Local Computation Architecture

It was mentioned above that the `propagate` call executes a complete run of the Shenoy-Shafer architecture. This is because the default parametrization of the local computation factory creates `JoinTree` objects whose propagation method corresponds to the Shenoy-Shafer algorithm. In fact, it is a general rule in NENOK that every architecture is realized by its proper specialization of the `JoinTree` class. Tricking the `LCFactory` into constructing join trees of another architecture is considerably easy, as the following method signature foreshadows:

- **public void** `setArchitecture`(`Architecture arch`);
  Changes the architecture of join trees that will be constructed.

`Architecture` is a *Java enum type* that provides a value for most local computation architectures considered in this thesis. Namely, these are:

- `Shenoy_Shafer`
  Standard Shenoy-Shafer architecture specified in Section 4.4.
- `Binary_Shenoy_Shafer`
  Improved Shenoy-Shafer architecture on binary join trees from Section 4.9.3.
- `Lauritzen-Spiegelhalter`
  Lauritzen-Spiegelhalter architecture specified in Section 4.5.
- `Hugin`
  Hugin architecture specified in Section 4.6.
- `Idempotent`
  Idempotent architecture specified in Section 4.7.
- `Dynamic_Programming`
  Dynamic Programming architecture specified in Section 8.3.2.

Thus, the following code modifies the example from the previous section and executes the Hugin architecture in place of the Shenoy-Shafer architecture:

```
LCFactory factory = new LCFactory();
factory.setArchitecture(Architecture.Hugin);
JoinTree jt = factory.create(kb);
jt.propagate();
```

Note also that a `ConstrException` is thrown, if the knowledgebase does not satisfy the mathematical properties of the chosen architecture. Finally, NENOK is also supplied with an additional interface that allows experienced users to define and run their own local computation architectures. We refer to (Pouly, 2006) for the

specification of this interface and to (Schneuwly, 2007) for a possible instantiation.

The architecture for dynamic programming requires a knowledgebase that consists of `OSRValuation` objects (or locators that point to such objects). In addition to the usual functionality of the other five default architectures, it allows to find single solution configurations in the following way:

```
LCFactory factory = new LCFactory();
factory.setArchitecture(Architecture.Dynamic_Programming);
DPJoinTree jt = (DPJoinTree)factory.create(kb);
jt.propagate();

FiniteVariable[] vars = kb.getDomain().toArray(FiniteVariable.class);
String[] sol = jt.getSolutionConfiguration(vars);
```

The last code line asks the architecture for the computed solution configuration. As explained in Section 10.3.1, `FiniteVariable` accepts only string values. Therefore, configurations of such variables are naturally represented by string arrays. Of particular importance is the variable array that is created by the next to last instruction, since it defines the ordering of the values within the configuration. Thus, `sol[i]` refers to the value of variable `vars[i]`.

12.1.2 Changing the Join Tree Construction Algorithm

As a second important component, the local computation factory administrates join tree construction algorithms. Remember, we have seen in Section 4.9.2 that constructing optimum join trees is known to be NP-complete. This suggests the use of heuristics for this complex task, which in turn implies that construction algorithms should be interchangeable, since users might be interested in their comparison. Comparably simple to the architecture type, the construction algorithm used by the local computation factory can be changed by the following method:

- **public void** `setConstructionAlgorithm(Algorithm algo)`;
  Changes the construction algorithm used to build join trees.

As a basic principle, every user can equip NENOK with its own construction algorithm. For this, it is sufficient to extend the abstract class `nenok.constr.Algorithm`, and to pass a corresponding object to the above method. This approach follows the COMMAND design pattern (Gamma *et al.*, 1993). A small collection of preimplemented algorithms is naturally brought along by NENOK directly. Two of them can be employed with the following constructors:

- **public** `OSLA_SC()`
  Builds a join tree by application of the *One-Step-Look-Ahead Smallest-Clique* algorithm from (Lehmann, 2001).

- **public** `Sequence(Variable[] elim)`;
  Builds a join tree from a specified variable elimination sequence.

The default parametrization of `LCFactory` uses the `OSLA_SC` algorithm. In the code snippet below, it is shown how the `Sequence` algorithm can be used instead. Clearly, it must be ensured that the proposed variable elimination sequence comprises all variables within the knowledgebase. Since we did not specify the content of the knowledgebase in this example, the variable elimination sequence is only schematically pictured. If the variable elimination sequence does not match with the current knowledgebase, a `ConstrException` will be thrown.

```
LCFactory factory = new LCFactory();

Sequence seq = new Sequence(new Variable[] {v1, ..., vn});
factory.setConstructionAlgorithm(seq);

JoinTree jt = factory.create(kb);
jt.propagate();
```

12.2 Executing Local Computation

The code extract of Section 12.1 shows that a single method call suffices to perform local computation on a join tree that was built by the local computation factory. This call of `propagate` executes an inward followed by an outward propagation of the current local computation architecture. However, the user may perhaps be interested in executing the two runs separately or, if only one query has to be answered, we can even omit the execution of the outward propagation. The following signatures summarize all possibilities to perform local computation on a `JoinTree` object:

- **public void** `collect`() **throws** `LCException`;
  Executes the collect algorithm of the current architecture.
- **public void** `distribute`() **throws** `LCException`;
  Executes the distribute algorithm of the current architecture.
- **public void** `propagate`() **throws** `LCException`;
  Executes `collect` and `distribute` consecutively.

We furthermore know from Section 4.8 that scaling versions of most architectures exist. By default, scaling is done in the root node at the beginning of every distribute phase. However, the user can explicitly renounce scaling by passing the argument **false** to either `distribute` or `propagate`.

We also want to allude to the highly transparent way of how local computation is executed. To initiate the join tree construction process, the local computation factory asks for a `Knowledgebase` object. It is known from Section 11.2.5 that two instantiations of this interface exist: one containing local valuations, and a second with locators that point to remote valuations. For knowledgebases with local data, the above methods work as purely local algorithms, On the other hand, if we pass a distributed knowledgebase to the join tree construction process, all local computation methods are executed as distributed algorithms. The measures taken

for efficient communication depend in this case on the current valuation algebra. If weight predictability is fulfilled, all architectures minimize communication costs according to the algorithm of Section 5.2.2, otherwise, a simple greedy algorithm is applied.

12.2.1 Query Answering

The creator method to build join trees ensures that all queries given as argument are covered by the constructed tree. Regardless, the class `JoinTree` directly offers a method to answer arbitrary queries. This procedure searches for a covering join tree node and returns its content marginalized to the given query. If no such node exists, a `LCException` will be thrown. Clearly, this cannot happen for those queries that have been passed to the previously mentioned creator method. Furthermore, only queries that are covered by the root node can be answered without executing distribute. The signature of the query answering method is:

- **public** `Valuation answer(Domain query)` **throws** `LCException`;
  Answers the given query by marginalizing the content of a covering node.

The following example creates a join tree from a given knowledgebase `kb` and query array `queries`, executes a complete Shenoy-Shafer run and displays the output of all specified queries on the screen.

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

for(Domain query : queries) {
  System.out.println(jt.answer(query));
}
```

12.3 Local Computation on closer Inspection

The introduction motivated NENOK as an experimental platform for local computation. For this purpose, the framework brings along a large number of features to inspect join trees and runs of local computation. We get a better picture by listing some of the informative methods, starting with those in the `JoinTree` class:

- **public double** `getCollectTime`();
  Returns the time in milliseconds spent for the execution of collect.

- **public double** `getDistributeTime`();
  Returns the time in milliseconds spent for the execution of distribute.

- **public** `Domain getLargestDomain`();
  Returns the label of the largest join tree node as decisive factor for complexity.

- **public int** `countNodes`();
  Returns the total number of join tree nodes.

Many further informative methods are available via a second component that can be asked from the `JoinTree` object. The `Adapter` class is such an example:

- **public** `Adapter` `getAdapter`();
  Returns the adapter of the current join tree.

This class is part of an ADAPTER design pattern (Gamma *et al.*, 1993) that abstracts the execution of remote and local valuation algebra operations, dependently on the current type of knowledgebase. Besides their core functionality, adapters count the number of executed operations:

- **public int** `getCombinations`();
  Returns the number of executed combinations.

- **public int** `getMarginalizations`();
  Returns the number of executed marginalizations.

- **public int** `getDivisions`();
  Returns the number of executed divisions.

If the current join tree was built from a remote knowledgebase, its adapter will subclass `RemoteAdapter`. In this case, we can even ask for the total communication costs that have been caused so far:

- **public double** `getCommunicationCosts`();
  Returns the total communication costs of computations in this join tree.

We learned in Section 12.1.2 that `Algorithm` objects encapsulate join tree construction algorithms. Moreover, every `JoinTree` remembers the algorithm that was used for its construction. This object can be obtained through the following method:

- **public** `Algorithm` `getConstructionAlgorithm`();
  Returns the construction algorithm that was used to build this join tree.

The `Algorithm` class contains some informative methods that refer to the join tree construction process:

- **public** `String` `getName`();
  Returns the name of the construction algorithm.

- **public double** `getConstructionTime`();
  Returns the time in milliseconds spent for the join tree construction.

Most construction algorithms are somehow based on variable elimination. In this case, the algorithms are headed by the abstract class `Elimination` which naturally extends `Algorithm`. This class offers the additional possibility to ask for the variable elimination sequence of the construction process:

- **public** `Variable`[] `getEliminationSequence`();
  Returns the elimination sequence of the join tree construction process.

Although this listing is far from being complete, it nevertheless gives an impression on the inspection possibilities offered by the NENOK framework.

13
User Interface

This final chapter of the NENOK tutorial focusses on the functionalities offered by the top layer, where two essential tasks are addressed: On the one hand, we have *generic input processing* which proposes a framework supplement to read knowledgebases from external files. The fact that NENOK works with a generic representation of valuation algebras complicates this task considerably. In simple terms, a system is required that parses files with partially unknown input, since we are generally not acquainted with the user instantiation of the algebraic layer. Thus, NENOK must be able to parse valuation objects even though no further information about the underlying formalism is available. On the other hand, the user interface layer meets the challenge of preparing NENOK for its use as experimental workbench, as promised repeatedly in this thesis.

13.1 Generic Input Processing

In Section 10.4, we modeled the ELISA test example by explicitly creating variable and valuation objects according to our valuation algebra implementation. One can easily imagine that for broader models with perhaps a few thousands of valuations, this becomes a dreadful job. Additionally, such models often come from other programs which focus essentially on the ability to create models via a graphical user interface. Typically, a very large number of such programs for a graphical specification of Bayesian networks or relational databases exist. These programs are not necessarily written in Java which complicates a direct connection to NENOK enormously. Finally, the framework should encourage the user to experiment with local computation, but each slight modification of the input data results in the need for recompilation. All these scenarios suggest that some possibility to import knowledgebases from external files is urgently needed. Beyond this feature, NENOK offers to read in a second type of input files that contain the structure of a covering join tree directly. This is of great value if we, for example, want to ensure that a multitude of local computation experiments run on the same join tree. Clearly, this cannot be guaranteed by reapplying the construction algorithms since they are normally based on heuristics and exploit randomness.

13.1.1 Knowledgebase Input Files

The first scenario to be considered is that of generic input files containing knowledgebases. Two components are especially important for this import functionality. On one hand, NENOK provides a generic XML structure that defines the skeleton of a knowledgebase input file and that can be refined to contain input data for the user's own valuation algebra implementation. On the other hand, there is an XML parser that is based on the TEMPLATE METHOD design pattern (Gamma *et al.*, 1993). It parses the predefined skeleton of the input file and calls generic methods for the user specified file content.

Let us have a closer look at this approach by first specifying the XML skeleton. In order to perform local computation with the imported data, NENOK must be fed the variables and valuations of the knowledgebase as well as the queries to be answered. These elements are incorporated into the following structure:

```
<knowledgebase>
    <variables>
        <variable varname="v1">...</variable>
            ⋮
    </variables>
    <valuations>
        <valuation>...</valuation>
            ⋮
    </valuations>
    <queries>
        <query>...</query>
            ⋮
    </queries>
</knowledgebase>
```

The <knowledgebase> tag lists the elements <variables>, <valuations> and, optionally, <queries>. Inside, all three elements are allowed to contain an arbitrary number of their corresponding sub-tags. The <variable> elements possess an attribute that denotes the name of the variable represented. This name must be unique and will be referenced by the content of the <query> tags. The remaining content together with the content of the <valuation> tags constitutes the generic part of this structure that will be addressed below. The inside of the <query> element, which has to be a list of white space-separated variable names, is more finely structured. As one can see, this file skeleton is a generic structure due to the unspecified content of the <variable> and <valuation> tags. We refer to Listing 13.1 that exemplifies the input data of the ELISA test example using this skeleton.

The requirement that XML input files should be validated before being passed to the parser is challenging. For this purpose, NENOK provides an XML Schema file named *va.xsd* that defines the buildup of the skeleton. The generic parts, namely the content of <variable> and <valuation>, are specified as follows within this schema:

```
<xs:element name="variable" type="varcontent" />
<xs:element name="valuation" type="valcontent" />

<xs:complexType name="varcontent">
    <xs:attribute name="varname" type="xs:ID" use="required"/>
</xs:complexType>

<xs:complexType name="valcontent"/>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<knowledgebase xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../../ppotential.xsd">
    <variables>
        <variable varname="HIV">
            <frame>false true</frame>
        </variable>
        <variable varname="Test">
            <frame>neg pos</frame>
        </variable>
    </variables>
    <valuations>
        <valuation>
            <label>HIV</label>
            <probabilities>0.997 0.003</probabilities>
        </valuation>
        <valuation>
            <label>Test HIV</label>
            <probabilities>0.98 0.01 0.02 0.99</probabilities>
        </valuation>
    </valuations>
    <queries>
        <query>Test</query>
    </queries>
</knowledgebase>
```

Listing 13.1: Input data of the ELISA test example of Section 10.4.

Interestingly, the schema prescribes an empty content for both generic tags. Thus, it would rightly not validate Listing 13.1. However, XML Schema provides a feature that allows to include and redefine elements, and this technique will be used to refine the global skeleton schema to a specific user instance. More precisely, the user creates its own XML Schema file that redefines the two elements <varcontent> and <valcontent> from *va.xsd*. All other elements of the file skeleton are fully specified and do not need to be redefined. This relationship between XML Schemata is illustrates in Figure 13.1, and Listing 13.2 adapts *va.xsd* to probability potentials such that our input file from Listing 13.1 validates correctly.

As we now dispose of a knowledgebase input file that validates correctly with respect to a user redefinition of the skeleton schema, we may next focus on how such

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
 elementFormDefault="qualified" attributeFormDefault="unqualified">
    <xs:redefine schemaLocation="va.xsd">
        <xs:complexType name="varcontent">
            <xs:complexContent>
                <xs:extension base="varcontent">
                    <xs:all>
                        <xs:element name="frame">
                            <xs:simpleType>
                                <xs:restriction base="xs:NMTOKENS">
                                    <xs:minLength value="1"/>
                                </xs:restriction>
                            </xs:simpleType>
                        </xs:element>
                    </xs:all>
                </xs:extension>
            </xs:complexContent>
        </xs:complexType>
        <xs:complexType name="valcontent">
            <xs:complexContent>
                <xs:extension base="valcontent">
                    <xs:sequence>
                        <xs:element name="label" type="dom_type"/>
                        <xs:element name="probabilities">
                            <xs:simpleType>
                                <xs:restriction base="plist">
                                    <xs:minLength value="1"/>
                                </xs:restriction>
                            </xs:simpleType>
                        </xs:element>
                    </xs:sequence>
                </xs:extension>
            </xs:complexContent>
        </xs:complexType>
    </xs:redefine>
    <xs:simpleType name="plist">
        <xs:list>
            <xs:simpleType>
                <xs:restriction base="xs:double">
                    <xs:minInclusive value="0.0"/>
                    <xs:maxInclusive value="1.0"/>
                </xs:restriction>
            </xs:simpleType>
        </xs:list>
    </xs:simpleType>
</xs:schema>
```

Listing 13.2: XML Schema for probability potentials.

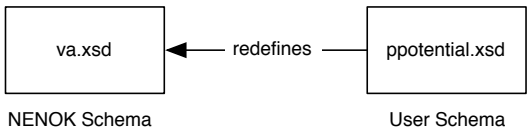


Figure 13.1: XML Schema relationships: NENOK schema refined by user schema

files are parsed. From a bird's eye view, this parsing process is shown in Figure 13.2. Essentially, the parser component transforms the input data from the given XML file into an array of `Valuation` and an array of `Domain` objects. The two components are wrapped into a class named `ResultSet` such that they can both be returned by a single call of a parsing method.

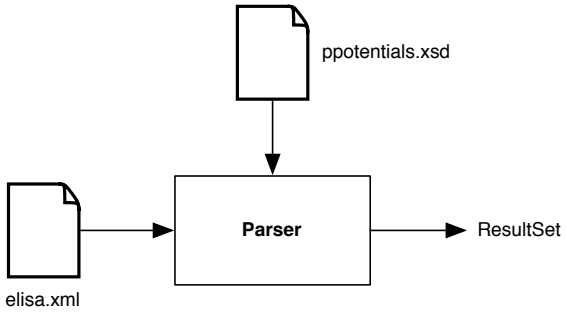


Figure 13.2: Parsing process of knowledgebase input files.

The parser component is specified within the abstract class `XmlParser`. Therein, a parsing method exists which accepts an XML file as only argument and transforms its content to a `ResultSet`:

- **public** `ResultSet parse(File file)` **throws** `ParserException`;

It is absolutely indispensable that the given file references the user refined XML Schema in its head tag. The parsing method will first check the validity of the file against the referenced schema and throw a `ParserException` in case of invalid content. Afterwards, this method parses the file skeleton that is independent from the valuation algebra instance. Whenever the parsing process meets a tag of type <variable> or <valuation>, its corresponding template method is called:

- **public abstract** `Variable parseVariable(String name, List<Element> content)`;
  Template method to parse the content of a <variable> tag. The first argument corresponds to the variable's name that has been extracted from the `varname` attribute. The second attribute contains a list of all elements inside the <variable>

tag, i.e. the generic content. The corresponding data type `org.jdom.Element` specified by the Xerces framework is a representation of XML tags whose textual content can be asked using `element.getText()`. From these informations, the user must create and return a new `Variable` instance. It is important to know that all <variable> tags are parsed strictly before the first <valuation> tag.

- **public abstract** `Valuation parseValuation(List<Element> content, Hashtable<String, Variable> content);`
  Template method to parse the content of a <valuation> tag. The first argument corresponds again to the list of elements inside the <valuation> tag. Because valuations refer to variables, the second argument delivers a hash table that maps the variable identifier from the XML file to the real `Variable` objects that have been created beforehand. Again, the user needs to create and return a new `Valuation` instance from these information.

The NENOK parsing framework is therefore completely implemented by a user defined class extending `XmlParser` that implements the two template methods in the above listing. It should not be surprising that the local computation factory offers an appropriate method to build join trees based on an input file and a parser implementation:

```
public JoinTree create(File file, KB_Parser parser);
```

Here, `KB_Parser` stands for the head interface that abstracts all parsers for knowledgebase files. This interface is implemented by `XmlParser` for the XML based knowledgebase files presented in this section.

13.1.2 Join Tree Input Files

We accented in the introduction of this chapter that good reasons exist for the possibility to read in serialized join trees. On one hand, this allows to produce identical join trees. On the other hand, it is often the case that we like to create a join tree with some special properties. Join tree construction by variable elimination is a very poor instrument for this intention, because it is in general very difficult to predict a join tree's shape based on its variable elimination sequence. Therefore, the input processing framework has been extended to accept serialized join trees besides the already discussed knowledgebases. In this style, we define the following skeleton for serialized join trees:

```
<jointree>
    <variables>
        <variable varname="v1">...</variable>
            ⋮
    </variables>
    <nodes>
        <node name="n1" child="ni">
            <domain>...</domain>
            <valuation>...</valuation>
        </node>
```

```
        ⋮
    </nodes>
</jointree>
```

The content of the <jointree> head element starts identically to knowledgebase files with a listing of all occurring variables. This is followed by the actual description of the join tree, packaged in the <nodes> element. A join tree is specified by a sequence of <node> tags of arbitrary length. Each <node> tag must contain a unique identifier as first attribute. The second attribute allows to link nodes together by referencing the identifier of their child node. This second attribute is not mandatory because the root node is childless. The inside of the <node> tags is given by two elements <domain> and <valuation>. The former expresses the node label, and its content is again a white space-separated list of variable names, identical to <query> tags in knowledgebase files. The <valuation> tag defines the node content. Again, we have already met this element in the description of the knowledgebase skeleton. Listing 13.3 shows a serialized join tree for the ELISA test example.

It is very important to observe the common parts of the two skeletons for knowledgebase and join tree files. The elements <variables>, <variable> and <valuations> have already been defined in *va.xsd*. The same holds for the data type of the <domain> tag, which is equal to <query> in the knowledgebase skeleton. It is therefore sensible to derive the XML Schema for join trees from the knowledgebase schema of the previous section. Indeed, *jointree.xsd* extends *va.xsd* by adding the definitions for the elements <jointree>, <nodes> and <node>, which are all non-generic. It is therefore sufficient to change the entry for the referenced schema in *ppotentials.xsd* to *jointree.xsd* in order to obtain an XML Schema that validates join tree files. These relationships between NENOK and user schemata are illustrated in Figure 13.3. Thus, the XML Schema *ppotentials.xsd* would validate the instance shown in Listing 13.3 as well as the knowledgebase instance of Listing 13.1.

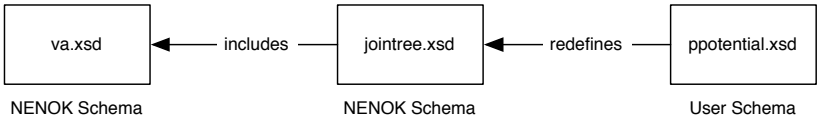


Figure 13.3: XML Schema relationships: NENOK schema refined by user schema

Corresponding to the parsing method for knowledgebase files, the local computation factory `LCFactory` provides a method for rebuilding a serialized join tree.

- **public** `JoinTree` `rebuild`(`File` `file`, `JT_Parser` `parser`) **throws** `ConstrException`;

Alternatively, this deserialization process can also be regarded as a join tree construction process itself, which is also reflected by its implementation. Indeed, the component `Deserializer` which reconstructs serialized join trees extends the `Algorithm`

```
<?xml version="1.0" encoding="UTF-8"?>
<jointree xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../../ppotential.xsd">
    <variables>
        <variable varname="HIV">
            <frame>false true</frame>
        </variable>
        <variable varname="Test">
            <frame>neg pos</frame>
        </variable>
    </variables>
    <nodes>
        <node name="1">
            <domain>HIV Test</domain>
            <valuation>
                <label>Test HIV</label>
                <probabilities>0.98 0.01 0.02 0.99</probabilities>
            </valuation>
        </node>
        <node name="2" child="1">
            <domain>HIV</domain>
            <valuation>
                <label>HIV</label>
                <probabilities>0.997 0.003</probabilities>
            </valuation>
        </node>
        <node name="3" child="1">
            <domain>Test</domain>
        </node>
    </nodes>
</jointree>
```

Listing 13.3: Serialized join tree for the ELISA test example from Section 10.4.

class and therefore functions as construction algorithm. Remembering Section 12.3, this is an example of a construction algorithm that does not use variable elimination.

13.2 Graphical User Interface

The description of the framework functionality of NENOK was so far focused on the code level. Thus, we addressed ourselves to the developers who want to embed NENOK as local computation library in their own applications. However, we also aimed for an alternative access via a graphical user interface that makes NENOK an experimental workbench for educational and analytical purposes. We start our guided tour by first showing how the graphical user interface is launched. The class `gui.Viewer` acts as entry point and offers the following methods to open the main window:

- **public static void** `display`(`Knowledgebase`... `kbs`);
  Opens the graphical interface that lists the given knowledgebases.
- **public static void** `display`(`Knowledgebase`[] `kbs`, `JoinTree`[] `jts`);
  Opens the graphical interface that lists the knowledgebases and join trees.

Figure 13.5 shows the main window of the graphical interface started with two knowledgebases called *Studfarm* and *Dog* which both model Bayesian network examples taken from (Jensen, 2001) and (Charniak, 1991). Further, we have also passed a join tree that has been constructed from a knowledgebase modeling the *Earthquake* example from (Pearl, 1988). As we can see on the control panel, the marked knowledgebase *Studfarm* contains 13 valuations over 12 variables. The knowledgebase factors are of type `Potential` which refers to our implementation of probability potentials. Further, the opened dialog displays the valuation network of this knowledgebase produced by the *Network* button.

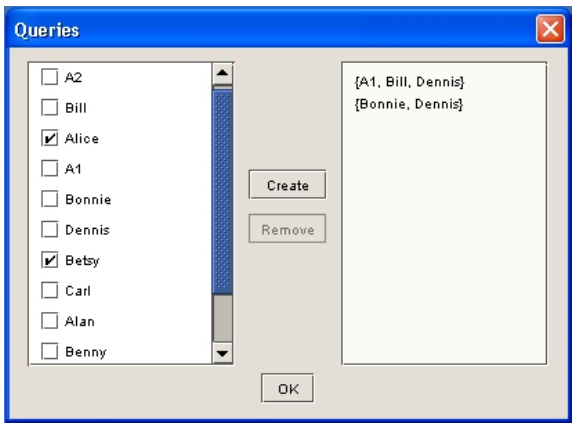


Figure 13.4: NENOK dialog to specify queries.

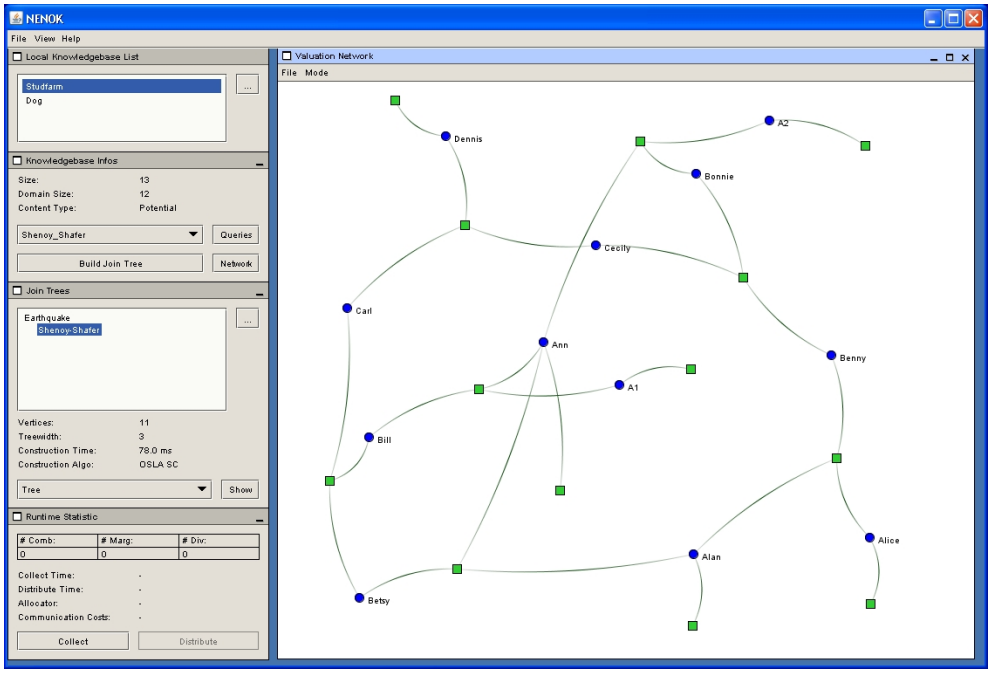


Figure 13.5: Main window of the graphical user interface.

The dialog from Figure 13.4 appears when the *Queries* button is hit and allows to specify user queries for local computation. The drop-down list finally defines the architecture of the join tree built with the *Create Join Tree* button. Figure 13.6 updates the scene with a new Hugin join tree constructed from the *Studfarm* knowledgebase. Next to the join tree list, we find the information that the newly created join tree has 24 vertices and that its largest node label contains 5 variables. Additionally, the construction time and the name of the construction algorithm are also indicated. NENOK offers different possibilities for displaying join trees on screen, available via the drop-down list. The perhaps most natural way to visualize trees is shown in the opened dialog of Figure 13.6. This visualization mode also allows to highlight the induced factor or processor distribution. Alternatively, Figure 13.7 contains the same join tree projected onto the surface of a sphere.

Local computation can be applied to the currently marked join tree using the two buttons named *Collect* and *Distribute*. Figure 13.8 shows the main window after the execution of collect and distribute on the *Studfarm* join tree. Statistics about runtime or the number of operations executed can also be retained.

NENOK naturally offers many more features and we want to close this tutorial by enumerating some of them. Via the menu *View* in the main window, we can for example scan the network for a running knowledgebase registry service. Then, all knowledgebases within this registry are listed as it was the case for the local

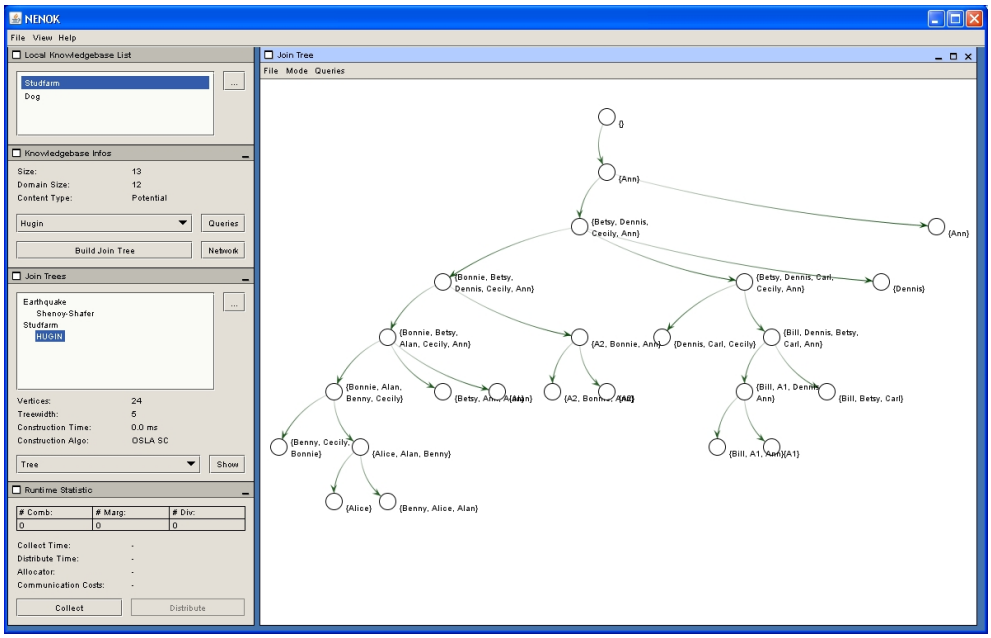


Figure 13.6: Main window updated with a new join tree.

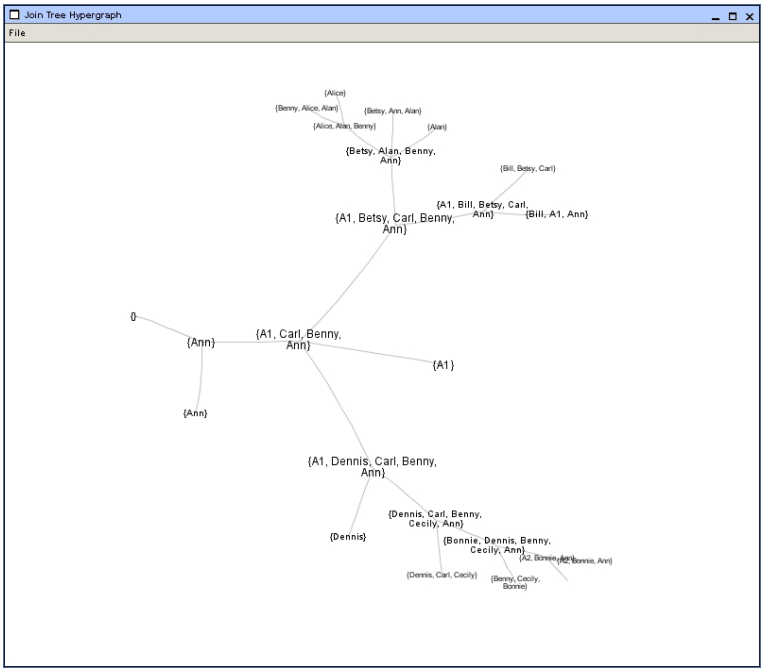


Figure 13.7: The *Studfarm* join tree projected to the surface of a sphere.

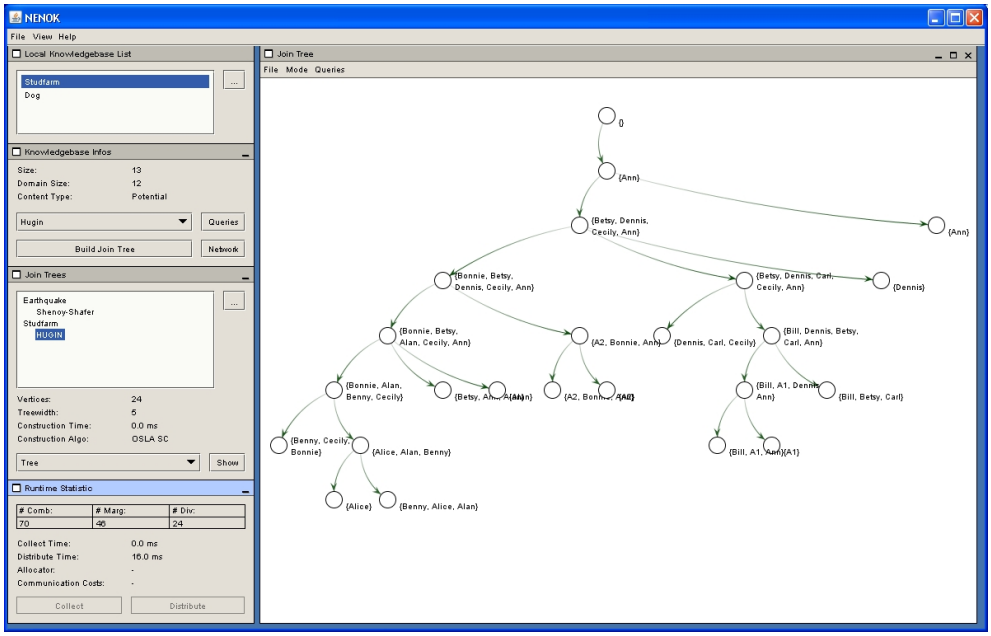


Figure 13.8: Main window after the execution of local computation.

knowledgebases in Figure 13.5. Once more, NENOK does not distinguish between local and remote data. Finally, there are also dialogs for importing knowledgebases and join trees from external files.

Part V

Conclusion

14
Summary

In this thesis, we brought out the importance of valuation algebras on two different levels. On the one hand, this framework pools all mathematical requirements for the application of a very efficient inference mechanism that manifests itself in the shape of four different architectures. On the other hand, we discovered a mathematical system that mirrors many natural properties we generally associate with the rather imprecise notions of knowledge and information, so that it can be regarded as an algebraic approach towards their definition. In contrast to the existing literature, we focused on a more general valuation algebra definition that does without the required existence of neutral elements. From the algebraic view, further important formalisms such as density functions are now covered without the need to straighten out the theory in an artificial manner. Furthermore, programmers get rid of the omnipresent problem that in some formalisms, neutral elements do not have finite representations. For algorithmic purposes, however, we nevertheless need some element to mark the absence of knowledge. This role is played by the identity element we added artificially to all valuation algebras that do not possess neutral elements, and it turned out that all valuation algebra properties are conserved under this construction. Additionally, we proved that all local computation architectures can be placed on this more general foundation, with the additional advantage that the restrictive precondition of a join tree factorization is weakened to the concept of a covering join tree that even permits more efficient local computation. The formal introduction of valuation algebras was accompanied by a large catalogue of instances that also includes distance functions which have so far never been considered in this context.

A large valuation algebra subclass is built from formalisms that map configurations to semiring values. We studied in detail how valuation algebra properties arise from the attributes of the underlying semiring. Further, we identified many other valuation algebra instances that fit into this particular framework and also explained how famous applications and algorithms from signal processing and decoding theory reduce to local computation with semiring valuations. Another subclass that we studied for the very first time covers formalisms that map configuration sets to semiring values.

Idempotent semirings are of particular interest in this thesis because they give rise to valuation algebras that model optimization problems. Here, we do not only settle for the computation of the optimum value, which can be done using the established local computation architectures, but we also focus on the identification of configurations that adopt this value. Such configurations are called solution configurations and their identification corresponds to the well-established task of dynamic programming. At first, we have presented an algorithm for this problem that exploits local computation to enumerate single or all solution configurations. Then, an alternative and more ambitious approach for the same task is developed that uses local computation techniques to compile solution configurations into an efficient representation of a Boolean function. After this compilation step, we have a very compact representation of the solution configuration set which can then be evaluated efficiently for a large number of relevant queries. This new approach is far more powerful than classical dynamic programming which contents itself with the pure enumeration of solution configurations.

The final part of this thesis was dedicated to the NENOK software framework that offers generic implementations of the algorithms presented. It is based on a realization of the algebraic framework and allows to plug in user defined valuation algebra instances in order to process them with the available methods. This outlines the use of NENOK as a software library for all kinds of local computation related applications. In addition, NENOK has a large number of elaborated tools for inspecting the local computation process and the related graphical structures. To facilitate access to these possibilities, a sophisticated user interface has been developed that suggests the use of NENOK as a workbench for educational or analytical purposes. The development process of NENOK also highlighted the often neglected requirement for an inference mechanism to be realized as distributed algorithms, since knowledge and information are naturally distributed resources. This motivated the theoretical studies on efficient communication which lead to the awareness that the join tree structure, which enables efficient computation, may under certain algebraic conditions also serve as network overlay to minimize communication during the inference process with low polynomial effort.

15
Future Work

Our last words are dedicated to enumerate some work in progress and ideas for future research activities. Though, we confine ourselves to outline only a few proposition for every hot topic of this thesis:

- **Detecting hidden projection problems:** We pointed out in this thesis that many important applications in computer science can be reworded in terms of the projection problem and therefore qualify for applying local computation techniques. This has for example be shown for the discrete Fourier and Hadamard transforms, Bayesian networks, database querying or linear and maximum likelihood decoding. It suggests itself that further applications can be tracked in a similar manner as for example the *discrete cosine transform* known from the JPEG algorithm or *typical set decoding on Gaussian channels* (Aji, 1999). Moreover, there are many fields which have not yet been considered under this perspective – *numerical interpolation* is just one example.

- **Generic valuation algebra construction:** It was shown in Section 6.3 and 7.1 that valuation algebras can be constructed in a generic manner by a simple mapping from (sets of) configurations to semiring values. Thus, similar structures are supposed behind other formalisms as for example the distance potentials of Section 3.6. Moreover, remembering that the valuations are fixpoints in this formalism, we could even guess the structure of a valuation algebra behind general fixpoint formalisms.

- **Distributed and parallel local computation:** The investigations of Chapter 5 only focussed on efficient communication when local computation is performed in a distributed system. Alternatively, we could also consider the perhaps conflictive goal of *workload balancing*, or the increase of parallelism during computations. For the latter case, there are essentially two approaches (Kozlov & Singh, 1994): *Topological parallelism* tries to eliminate node sequences in the join tree to make it *bushier*. This implies that more sub-trees can perform their computations in parallel. However, since nodes with larger width dominate the computation, there is only little effect of these measures (D'Ambrosio *et al.*, 1992). On the other hand, *inclique parallelism* aims at the parallel execution of computations within the join tree nodes.

- **NENOK development:** There are in fact many ideas to equip NENOK with additional features. On the algorithmic level, an implementation of the procedure to compile solution configurations from Section 8.4 is still missing. This however requires first the presence of a framework to represent PDAG structures. Further, the current version of NENOK allows to perform local computation in a distributed manner thanks to the underlying Jini architecture. This is based on our profound studies on how to ensure efficient communication in Chapter 5. Contrariwise, no measures are yet taken to parallelize local computation. Probably less challenging to fulfil is the need for a simpler user interface, since NENOK still requires a lot of configuration work that may discourage users. This can perhaps be avoided by the implementation of a web interface or by the integration of NENOK into a *rich client platform*. Finally, it would be interesting to make a performance analysis of NENOK's local computation facilities. On the one hand, we should compare the local and remote execution of the different local computation architectures to give evidence about the costs of communication and service administration and on the other hand, one might also be interested in comparing NENOK with existing software projects for a particular valuation algebra. However, it is clear that NENOK will be at a disadvantage due to the more limited possibilities to optimize generic algorithms.

References

Aji, S.M. 1999. *Graphical Models and Iterative Decoding.* Ph.D. thesis, California Institute of Technology.

Aji, Srinivas M., & McEliece, Robert J. 2000. The Generalized Distributive Law. *IEEE Transactions on Information Theory*, **46**(2), 325–343.

Almond, R., & Kong, A. 1991. Optimality Issues in Constructing a Markov Tree from Graphical Models. *Research Report A-3. Department of Statistics, Harvard University*, November.

Arnborg, S., Corneil, D., & Proskurowski, A. 1987. Complexity of Finding Embeddings in a k-Tree. *SIAM J. of Algebraic and Discrete Methods*, **38**, 277–284.

Bertele, Umberto, & Brioschi, Francesco. 1972. *Nonserial Dynamic Programming.* Orlando, FL, USA: Academic Press, Inc.

Bistarelli, S., Fruhwirth, T., Marte, M., & Rossi, F. 2002. *Soft Constraint Propagation and Solving in Constraint Handling Rules.*

Bistarelli, Stefano, Montanari, Ugo, & Rossi, Francesca. 1997. Semiring-Based Constraint Satisfaction and Optimization. *Journal of the ACM*, **44**(2), 201–236.

Bovet, Daniel Pierre, & Crescenzi, Pierluigi. 1994. *Introduction to the Theory of Complexity.* Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd.

Cano, A., & Moral, S. 1995. Heuristic Algorithms for the Triangulation of Graphs. *Pages 166–171 of:* Bouchon-Meunier, R.R. Yager, & Zadeh, L.A. (eds), *Proceedings of the Fifth IPMU Conference.* Springer.

Chang, C.L., & Lee, R.C.T. 1973. *Symbolic Logic and Mechanical Theorem Proving.* Boston: Academic Press.

Charniak, E. 1991. Bayesian Networks without Tears. *The American Association for Artificial Intelligence.*

Clifford, A. H., & Preston, G. B. 1967. *Algebraic Theory of Semigroups.* Providence, Rhode Island: American Mathematical Society.

Croisot, R. 1953. Demi-Groupes Inversifs et Demi-Groupes Réunions de Demi-Groupes Simples. *Ann. Sci. Ecole norm. Sup.*, **79**(3), 361–379.

Dahlhaus, E., Johnson, D. S., Papadimitriou, C. H., Seymour, P. D., & Yannakakis, M. 1994. The Complexity of Multiterminal Cuts. *SIAM J. Comput.*, **23**(4), 864–894.

D'Ambrosio, B., Fountain, T., & Li, Z. 1992. Parallelizing Probabilistic Inference: Some early Explorations. *Pages 59–66 of: Proceedings of the eighth conference on Uncertainty in Artificial Intelligence*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Darwiche, A., & Huang, J. 2002. *Testing Equivalence Probabilistically*. Tech. rept. Computer Science Department, UCLA, Los Angeles, Ca 90095.

Darwiche, Adnan. 2001. Decomposable Negation Normal Form. *J. ACM*, **48**(4), 608–647.

Darwiche, Adnan, & Marquis, Pierre. 2002. A Knowledge Compilation Map. *J. Artif. Intell. Res. (JAIR)*, **17**, 229–264.

Davey, B.A., & Priestley, H.A. 1990. *Introduction to Lattices and Order*. Cambridge University Press.

de Groote, A. 2006. *Distance Functions and Local Computation*. M.Phil. thesis, Department of Informatics, University of Fribourg.

Dechter, R. 1999. Bucket Elimination: A Unifying Framework for Reasoning. *Artificial Intelligence*, **113**, 41–85.

Dijkstra, Edsger W. 1959. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, **1**, 269–271.

Eichenberger, Ch. 2007. *Conditional Gaussian Potentials*. Tech. rept. Department of Informatics, University of Fribourg.

Erdös, Peter L., & Szekely, Laszlo A. 1994. On Weighted Multiway Cuts in Trees. *Math. Program.*, **65**(1), 93–105.

Gallager, R.C. 1963. *Low Density Parity Check Codes*.

Gamma, Erich, Helm, Richard, Johnson, Ralph, & Vlissides, John. 1993. Design Patterns: Abstraction and Reuse of Object-Oriented Design. *Lecture Notes in Computer Science*, **707**.

Garey, Michael R., & Johnson, David S. 1990. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co.

Gottlob, G., Leone, N., & Scarcello, F. 1999. A Comparison of Structural CSP Decomposition Methods. *Pages 394–399 of: Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI)*. Morgan Kaufmann.

Green, J.A. 1951. On the structure of semigroups. *Annals of Math.*, **54**, 163–172.

Haenni, R. 2004. Ordered Valuation Algebras: A Generic Framework for Approximating Inference. *International Journal of Approximate Reasoning*, **37**(1), 1–41.

Haenni, R., & Lehmann, N. 1999. *Efficient Hypertree Construction.* Tech. rept. 99-3. Institute of Informatics, University of Fribourg.

Haenni, R., Kohlas, J., & Lehmann, N. 2000. Probabilistic Argumentation Systems. *Pages 221–287 of:* Kohlas, J., & Moral, S. (eds), *Handbook of Defeasible Reasoning and Uncertainty Management Systems, Volume 5: Algorithms for Uncertainty and Defeasible Reasoning.* Kluwer, Dordrecht.

Hewitt, E., & Zuckerman, H.S. 1956. The l_1 Algebra of a Commutative Semigroup. *Amer. Math. Soc.*, **83**, 70–97.

Jenal, J. 2006. *Gaussian Potentials.* M.Phil. thesis, Department of Informatics, University of Fribourg.

Jensen, F. V. 2001. *Bayesian Networks and Decision Graphs.* Springer.

Jensen, F.V., Lauritzen, S.L., & Olesen, K.G. 1990. Bayesian Updating in Causal Probabilistic Networks by Local Computation. *Computational Statistics Quarterly*, **4**, 269–282.

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

Kohlas, J. 2004. *Valuation Algebras Induced by Semirings.* Tech. rept. 04-03. Department of Informatics, University of Fribourg.

Kohlas, J., & Wilson, N. 2006. *Exact and Approximate Local Computation in Semiring Induced Valuation Algebras.* Tech. rept. 06-06. Department of Informatics, University of Fribourg.

Kohlas, J., Haenni, R., & Moral, S. 1999. Propositional Information Systems. *Journal of Logic and Computation*, **9**(5), 651–681.

Kong, A. 1986. *Multivariate Belief Functions and Graphical Models.* Ph.D. thesis, Department of Statistics, Harvard University.

Kozlov, A. V., & Singh, J. P. 1994. A parallel Lauritzen-Spiegelhalter Algorithm for Probabilistic Inference. *Pages 320–329 of: Supercomputing '94: Proceedings of the 1994 ACM/IEEE conference on Supercomputing.* New York, NY, USA: ACM.

Langel, J. 2004. *Local Computation with Models in Propositional Logic.* M.Phil. thesis, Department of Informatics, University of Fribourg.

Langel, J., & Kohlas, J. 2007. *Algebraic Structure of Semantic Information and Questions. Predicate Logic: An Information Algebra.* Tech. rept. Department of Informatics, University of Fribourg.

Lauritzen, S. L., & Spiegelhalter, D. J. 1988. Local Computations with Probabilities on Graphical Structures and their Application to Expert Systems. *J. Royal Statis. Soc. B*, **50**, 157–224.

Lauritzen, Steffen L., & Jensen, Finn Verner. 1997. Local Computation with Valuations from a Commutative Semigroup. *Ann. Math. Artif. Intell*, **21**(1), 51–69.

Lehmann, N. 2001. *Argumentation System and Belief Functions.* Ph.D. thesis, Department of Informatics, University of Fribourg.

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

Maier, D. 1983. *The Theory of Relational Databases.* London: Pitman.

Mateescu, Robert, & Dechter, Rina. 2006. Compiling Constraint Networks into AND/OR Multi-Valued Decision Diagrams (AOMDDs). *Pages 329–343 of: CP.*

Menger, Karl. 1942. Statistical Metrics. *Proceedings of the National Academy of Sciences*, **28**, 535–537.

Mitten, L.G. 1964. Composition Principles for the Synthesis of Optimal Multi-Stage Processes. *Operations Research*, **12**.

Newmarch, Jan. 2006. *Foundations of Jini 2 Programming.* Berkely, CA, USA: Apress.

Pearl, J. 1988. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann Publishers Inc.

Pouly, M. 2004a. *A Generic Architecture for Local Computation.* M.Phil. thesis, Department of Informatics, University of Fribourg.

Pouly, M. 2004b. Implementation of a Generic Architecture for Local Computation. *Pages 31–37 of:* Kohlas, J., Mengin, J., & Wilson, N. (eds), *ECAI'04, 16th European Conference on Artificial Intelligence, Workshop 22 on "Local Computation for Logics and Uncertainty".*

Pouly, M. 2006. *NENOK 1.1 User Guide.* Tech. rept. 06-02. Department of Informatics, University of Fribourg.

Pouly, M., Haenni, R., & Wachter, M. 2007. Compiling Solution Configurations in Semiring Valuation Systems. *Pages 248–259 of:* Gelbukh, A., & Kuri Morales, A. F. (eds), *MICAI'07: 6th Mexican International Conference on Artificial Intelligence.* LNAI 4827.

Robertson, Neil, & Seymour, Paul D. 1986. Graph Minors. II. Algorithmic Aspects of Tree-Width. *J. Algorithms*, **7**(3), 309–322.

Rose, D.J. 1970. Triangulated Graphs and the Elimination Process. *J. of Math. Analysis and Applications*, **32**, 597–609.

Schiex, Thomas. 1992. Possibilistic Constraint Satisfaction Problems or "How to handle soft constraints?". *Pages 268–275 of: Proceedings of the Eight International Conference on Uncertainty in Artificial Intelligence.*

Schmidt, T., & Shenoy, P. 1998. *Some Improvements to the Shenoy-Shafer and Hugin Architectures for Computing Marginals.*

Schneuwly, C. 2007. *Computing in Valuation Algebras.* Ph.D. thesis, Department of Informatics, University of Fribourg.

Schneuwly, C., Pouly, M., & Kohlas, J. 2004. *Local Computation in Covering Join Trees.* Tech. rept. 04-16. Department of Informatics, University of Fribourg.

Schweizer, B., & Sklar, A. 1960. Statistical Metric Spaces. *Pacific J. Math.*, **10**, 313–334.

Shafer, G. 1976. *A Mathematical Theory of Evidence.* Princeton University Press.

Shafer, G. 1991. *An Axiomatic Study of Computation in Hypertrees.* Working Paper 232. School of Business, University of Kansas.

Shenoy, P. P. 1992a. Using Possibility Theory in Expert Systems. *Fuzzy Sets and Systems*, **51**(2), 129–142.

Shenoy, P. P. 1992b. Valuation-Based Systems: A Framework for Managing Uncertainty in Expert Systems. *Pages 83–104 of:* Zadeh, L.A., & Kacprzyk, J. (eds), *Fuzzy Logic for the Management of Uncertainty.* John Wiley & Sons.

Shenoy, P. P. 1997. Binary Join Trees for Computing Marginals in the Shenoy-Shafer Architecture. *International Journal of Approximate Reasoning*, **17**, 239–263.

Shenoy, P. P., & Shafer, G. 1988. Axioms for Probability and Belief-Function Propagation. *Pages 169–198 of: Proceedings of the 4th Annual Conference on Uncertainty in Artificial Intelligence.* New York: Elsevier Science.

Shenoy, P. P., & Shafer, G. 1990. Axioms for Probability and Belief-Function Propagation. *Pages 169–198 of:* Shachter, R. D., Levitt, T. S., Kanal, L. N., & Lemmer, J. F. (eds), *Uncertainty in Artificial Intelligence 4.* Amsterdam: North-Holland.

Shenoy, P.P. 1996. Axioms for Dynamic Programming. *Pages 259–275 of:* Gammerman, A. (ed), *Computational Learning and Probabilistic Reasoning.* Wiley, Chichester, UK.

Smets, Ph. 2000. *Belief Functions and the Transferable Belief Model.*

Spring, P. 2006. *Reduced Relational Algebra: An Implementation in NENOK.* Tech. rept. Department of Informatics, University of Fribourg.

Tamer Özsu, M., & Valduriez, P. 1999. *Principles of Distributed Database Systems.* Vol. Second Edition. Prentice Hall.

Tamura, T., & Kimura, N. 1954. On Decompositions of a Commutative Semigroup. *Kodai Math. Sem. Rep.*, 109–112.

Ullman, J. D. 1988. *Principles of Database and Knowledge-Base Systems Volume I.* Maryland: Computer Science Press.

Wachter, M., & Haenni, R. 2006. Propositional DAGs: A New Graph-Based Language for Representing Boolean Functions. *Pages 277–285 of:* Doherty, P., Mylopoulos, J., & Welty, C. (eds), *KR'06, 10th International Conference on Principles of Knowledge Representation and Reasoning.* Lake District, U.K.: AAAI Press.

Wachter, M., & Haenni, R. 2007. Multi-State Directed Acyclic Graphs. *Pages 464–475 of:* Kobti, Z., & Wu, D. (eds), *CanAI'07, 20th Canadian Conference on Artificial Intelligence.* LNAI 4509.

Wachter, M., Haenni, R., & Pouly, M. 2007. Optimizing Inference in Bayesian Networks and Semiring Valuation Algebras. *Pages 236–247 of:* Gelbukh, A., & Kuri Morales, A. F. (eds), *MICAI'07: 6th Mexican International Conference on Artificial Intelligence.* LNAI 4827.

Wiberg, N. 1996. *Codes and Decoding on General Graphs.*

Wilson, N. 2004. Decision Diagrams for the Computation of Semiring Valuations. *Proc. 6th International Workshop on Preferences and Soft Constraints.*

Wilson, Nic, & Mengin, Jérôme. 1999. Logical Deduction using the Local Computation Framework. *Lecture Notes in Computer Science*, **1638**.

Yannakakis, M. 1981. Computing the Minimum Fill-in is NP-Complete. *SIAM Journal of Algebraic and Discrete Methods*, **2**(1), 77–79.

Zabiyaka, Yuliya, & Darwiche, Adnan. 2007. *On Tractability and Hypertree Width.* Tech. rept. D–150. Automated Reasoning Group, Computer Science Department, UCLA.

Zadeh, L.A. 1978. Fuzzy Sets as a Basis for a Theory of Possibility. *Fuzzy Sets and Systems*, **1**, 3–28.

Zadeh, L.A. 1979. A Theory of Approximate Reasoning. *Pages 149–194 of:* Ayes, J.E., Michie, D., & Mikulich, L.I. (eds), *Machine Intelligence*, vol. 9. Ellis Horwood, Chichester, UK.

Index

Curriculum Vitae

Personalien

Vorname	:	Marc
Name	:	Pouly
Adresse	:	Engelhardstrasse 45
Wohnort	:	3280 Murten (Schweiz)
Geburtsdatum	:	29. Januar 1980
Geburtsort	:	Meyriez
Telefon	:	026 670 50 25
E-Mail	:	marc.pouly@unifr.ch
Heimatort	:	Les Cullayes VD
Zivilstand	:	ledig

Schulbildung

1986 - 1991	:	Primarschule, Murten
1991 - 1995	:	Sekundarschule, Murten
1995 - 1999	:	Kollegium Heilig Kreuz, Freiburg (Schweiz) Abschluss: *Matura Typus C*
1999 - 2004	:	Studium der Informatik mit Nebenfach Mathematik, Universität Freiburg (Schweiz) Abschluss: *Master of Science* in Informatik

Sprachen

- Muttersprache Deutsch
- Gute Kenntnisse in Französisch
- Gute Kenntnisse in Englisch