Department of Computer Science
Series of Publications A
Report A-?2003?-?

# Sum-Product Algorithms for the Analysis of Genetic Risks

Mikko Koivisto

*To be presented, with the permission of the Faculty of Science of the University of Helsinki, for public criticism in Auditorium III, Porthania, on December 5th, 2001, at 12 o'clock noon.*

University of Helsinki
Finland

## Contact information

Postal address:
Department of Computer Science
P.O. Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki
Finland

Email address: postmaster@cs.Helsinki.FI

URL: http://www.cs.Helsinki.FI/

Telephone: +358 9 1911

Telefax: +358 9 191 44441


ISSN 1238-8645
ISBN 952-10-1251-? (paperback)
ISBN 952-10-1252-? (PDF)
Computing Reviews (1998) Classification: F.2.1, F.2.2, G.3, J.3
Helsinki ?2003?
Helsinki University Printing House

**Sum-Product Algorithms for the Analysis of Genetic Risks**

Mikko Koivisto

Department of Computer Science
P.O. Box 26, FIN-00014 University of Helsinki, Finland
Mikko.Koivisto@cs.helsinki.fi, http://www.cs.helsinki.fi/u/mkhkoivi/

PhD Thesis, Series of Publications A, Report A-?2003?-?
Helsinki, ?September? ?2003?, 153 pages
ISSN 1238-8645
ISBN 952-10-1251-? (paperback)
ISBN 952-10-1252-? (PDF)

**Abstract**

This work is motivated by a genetic data analysis task: recurrence risk analysis. Recurrence risks in various types of relatives, e.g. offsprings and siblings, characterize the inheritance pattern of a genetic disorder. Given observations of the population prevalence and different recurrence risks the task is to infer the number of underlying genes and the frequencies and effects of different variants of the genes. Due to a complex relationship of relatively simple data and a large number of model parameters, this problem is challenging both statistically and computationally. Straightforward application of existing techniques is not sufficient. This motivates deeper studies of a few related methodological themes that are interesting and important also in their own right.

In the first part of this thesis, we study three general methodological issues. First, we review the Bayesian paradigm for statistical inference. Special emphasis is on certain fundamental difficulties of practical Bayesian methods. Second, we study the sum-product problem, that is, marginalization of a multidimensional function that factorizes into a product of low-dimensional functions. We introduce a novel algorithm that improves the well-known variable elimination method by employing fast matrix multiplication methods. A special class of sum-product problems called transformation problems are studied. We generalize a technique known as the Yates' algorithm and show how the Möbius transformation on a subset lattice can be computed efficiently. Third, we consider the problem of integrating a multidimensional function. We describe a sophisticated method based on a tempering technique and Metropolis Coupled Markov chain Monte Carlo simulation. We argue that this method is particularly suitable to computations required in Bayesian data analysis. Connections to related methods are also explored.

The second part is devoted to the genetic application. We introduce a genetic model called epistatic Mendelian model. For computing recurrence risks under a fully specified model, we present a method that employs the Yates' algorithm. Based on a different problem representation, we also give another algorithm that uses the fast Möbius transform. To integrate over the model parameters we present a version of the Metropolis-coupled Markov chain Monte Carlo method. Finally, experiments are carried out to support two main conclusions: Bayesian analysis of recurrence risks is computationally feasible. And, recurrence risk data provides interesting information concerning competitive genetic hypotheses, but the amount of information varies depending on the data set.

**Computing Reviews (1998) Categories and Subject Descriptors:**

- F.2.1 Analysis of Algorithms and Problem Complexity: Numerical Algorithms and Problems
- F.2.2 Analysis of Algorithms and Problem Complexity: Nonnumerical Algorithms and Problems
- G.3 Probability and Statistics: Multivariate statistics, Probabilistic algorithms
- J.3 Life and Medical Sciences: Biology and genetics

**General Terms:**
Algorithms, Experimentation, Theory

**Additional Key Words and Phrases:**
Bayesianism, Genetic model, Markov chain Monte Carlo, Möbius transform, Recurrence risk, Variable elimination, Yates' algorithm

# Contents

# Chapter 3

# Sum-products and the elimination algorithm

In this chapter we study the problem of computing a marginal of a function that decomposes into a product of lower dimensional functions. We specify this sum-product problem formally in Section 3.1. In Section 3.2 we review a well-known, general algorithmic solution, the variable elimination algorithm. Using the results on fast matrix multiplication, we show in Section 3.3 that the conventional variable elimination algorithm is not optimal in general. An important special class of sum-product problems is formed by transformation problems introduced in Section 3.4. Some special transforms and related algorithms, including the Yates' algorithm, are derived. We also show how the Möbius transform on subset lattice can be mapped to a sum-product form. In Section 3.5, we emphasize the importance of finding appropriate sum-product representations and discuss the limitations of the variable elimination algorithm. Finally, Section 3.6 provides some remarks and open problems.

This chapter offers a generic, algorithmic perspective on certain issues studied elsewhere in this work. The study of computing marginals is, in general, motivated by the Bayesian paradigm reviewed in Chapter 2, and specially, by the genetic application problem described in Chapter 5. Results on Yates' algorithm and Möbius transform, given in this chapter, are applied in Chapter 6 for the computation of recurrence risks. Chapter 4 considers a related problem of marginalization over infinite domains.

## 3.1 The problem of marginalizing a product

We start by considering an example of a sum-product problem.

**Example 3.1** Consider a 4-dimensional marginal of an 8-dimensional function:

$$g(x_1, x_2, x_3, x_4) = \sum_{x_5=1}^{m} \sum_{x_6=1}^{m} \sum_{x_7=1}^{m} \sum_{x_8=1}^{m} f(x).$$

Here $x = (x_1, \ldots, x_8)$. Suppose further that $f$ decomposes into a product of four "local" functions:

$$f(x) = a(x_1, x_5, x_8)\, b(x_2, x_5, x_6)\, c(x_3, x_6, x_7)\, d(x_4, x_7, x_8).$$

How fast can the marginal function $g$ be computed given $a, b, c$, and $d$? Clearly, the straightforward method requires $\mathcal{O}(m^8)$ arithmetic operations. However, by exploiting the assumed problem structure, it is possible to derive an algorithm that uses only $\mathcal{O}(m^6)$, or even $\mathcal{O}(m^5)$, operations; such algorithms will be given in examples 3.16 and 3.23. □

In order to cover a wider class of computational problems similar to that of Example 3.1, we use a more abstract formulation. Following Aji and McEliece [AM00] we rely on certain general properties of addition and multiplication. The appropriate framework is the commutative semiring.

**Definition 3.2** A *commutative semiring* is a set $\mathcal{R}$, together with two binary operations called "$+$" and "$\cdot$", which satisfy the following axioms:

S1. The operation "$+$" is associative and commutative, and there is an additive identity element called "0" such that $r + 0 = r$ for all $r \in \mathcal{R}$.

S2. The operation "$\cdot$" is associative and commutative, and there is a multiplicative identity element called "1" such that $r \cdot 1 = r$ for all $r \in \mathcal{R}$.

S3. The distributive law holds, i.e., for all $a, b, c, \in \mathcal{R}$,

$$(a \cdot b) + (a \cdot c) = a \cdot (b + c).$$

The semiring is denoted as a triple $(\mathcal{R}, +, \cdot)$. □

Commutative semirings have many important instances. For example, the set of real or complex numbers, with ordinary addition and multiplication, forms a commutative semiring. Also, on an appropriate set, maximization or minimization operation, together with ordinary addition or multiplication, form a commutative semiring. Some most common commutative semirings are summarized in Table 3.1, adapted from [AM00]. An example

Table 3.1: Some Commutative Semirings

| $\mathcal{R}$ | "$(+,0)$" | "$(\cdot,1)$" | NOTE |
|---|---|---|---|
| $(-\infty,\infty)$ | $(+,0)$ | $(\cdot,1)$ | Also a ring |
| $(0,\infty]$ | $(\min,\infty)$ | $(\cdot,1)$ | |
| $[0,\infty)$ | $(\max,0)$ | $(\cdot,1)$ | |
| $(-\infty,\infty]$ | $(\min,\infty)$ | $(+,0)$ | |
| $[-\infty,\infty)$ | $(\max,-\infty)$ | $(+,0)$ | |
| $[-\infty,\infty)$ | $(\max,-\infty)$ | $(\min,\infty)$ | |
| $\{0,1\}$ | $(+,0)$ | $(\cdot,1)$ | Modulo 2, also a ring |
| $\{0,1\}$ | $(\mathrm{OR},0)$ | $(\mathrm{AND},1)$ | Boolean |
| $\mathcal{P}(S)$ | $(\cup,\emptyset)$ | $(\cap,S)$ | $S$ a finite set |
| $\Lambda$ | $(\vee,0)$ | $(\wedge,1)$ | $\Lambda$ a distributive lattice |
| $\Lambda$ | $(\wedge,1)$ | $(\vee,0)$ | $\Lambda$ a distributive lattice |

perhaps interesting, but not included in [AM00], is the maximin-semiring; one can easily verify the distributive law, for all $a, b, c$,

$$\max\{\min\{a,b\},\min\{a,c\}\} = \min\{a,\max\{b,c\}\}.$$

A more important fact, yet not included in Table 3.1, is that the semiring properties hold for mappings to a semiring when the addition and multiplication of two mappings is defined in a point-wise manner. For instance, the set of polynomials in one or more indeterminates over any commutative semiring forms a commutative semiring.

Sometimes stronger properties hold for the algebraic structure in question. An important restriction to the commutative semiring is the commutative ring.

**Definition 3.3** A commutative semiring $(\mathcal{R},+,\cdot)$ is a *commutative ring* if every element $a$ in $\mathcal{R}$ has an additive inverse in $\mathcal{R}$, i.e., an element $b$ such that $a+b=0$. □

An example of commutative ring is the set of integers together with ordinary addition and multiplication. If we drop the requirement of commutative multiplication, we get a *semiring* or a *ring*.[1] For example, the set of square matrices over a (semi)ring, with ordinary matrix addition and multiplication, forms a (semi)ring, but not a commutative (semi)ring. Also, recall that if for every nonzero element in a ring there is a multiplicative inverse, then the ring is called a *field*.

[1]In literature a ring that contains a multiplicative identity element is sometimes called *unit ring*. Here we do not make this distinction.

The extra property of commutative ring has mostly been neglected in related studies of sum-product algorithms [SI96, LJ97, BMR97, AM00, KFL01]. We show in Section 3.3 that in some cases the existence of additive inverses can be used to reduce the computational complexity. This is interesting, not only because one often operates with commutative rings, but also because in some cases a commutative semiring, that is not a commutative ring as such, can be embedded into a commutative ring with little computational overhead. Until Section 3.3 we, however, work only on the general case of commutative semirings.

We are now ready to describe the generalized sum-product problem. First, let us introduce some notation. Throughout this chapter, we let $x_1, \ldots, x_n$ be variables taking values in the finite sets $\mathcal{A}_1, \ldots, \mathcal{A}_n$. We use $N$ as a shorthand for the set $\{1, \ldots, n\}$. If $S = \{i_1, \ldots, i_r\}$ where $i_1 < \ldots < i_r$, is a subset of $N$, we denote the product $\mathcal{A}_{i_1} \times \cdots \times \mathcal{A}_{i_r}$ by $\mathcal{A}_S$, and the variable list $(x_{i_1}, \ldots, x_{i_r})$ similarly by $x_S$. We call $S$ the *scope* of $x_S$ and $\mathcal{A}_S$ the *domain* of $x_S$. We may drop the subscript when it refers to the whole universe $N$. The complement of $S$ (relative to the universe $N$) is denoted by $S^c$. We need also the following definition.

**Definition 3.4** Let $\mathcal{T}$ be a collection of subsets of a set $V$. We say that $\mathcal{T}$ is a *cover* on $V$ if the union of the elements in $\mathcal{T}$ equals $V$, i.e., $\bigcup \mathcal{T} = \bigcup_{T \in \mathcal{T}} T = V$. □

The generalized sum-product problem is now defined as follows.

**Definition 3.5** Let $\mathcal{A}_1, \ldots, \mathcal{A}_n$ be finite sets. Let $(\mathcal{R}, +, \cdot)$ be a semiring. Let $\mathcal{T}$ be a cover on $N$ and let $S$ be a subset of $N$. The tuple $(\mathcal{A}, \mathcal{R}, \mathcal{T}, S)$ defines an evaluation problem: Given functions $f_T : \mathcal{A}_T \to \mathcal{R}$ for $T \in \mathcal{T}$, evaluate for all $x_S \in \mathcal{A}_S$ the value $g_S(x_S)$ of the *marginal function* $g_S : \mathcal{A}_S \to \mathcal{R}$ defined by

$$g_S(x_S) = \sum_{x_{S^c} \in \mathcal{A}_{S^c}} \prod_{T \in \mathcal{T}} f_T(x_T).$$

The problem is called the *single marginalization of a product function* problem, and is denoted by SMPF$(\mathcal{A}, \mathcal{R}, \mathcal{T}, S)$. A problem *instance* is a set $\mathcal{F} = \{f_T : T \in \mathcal{T}\}$ of fully specified functions. □

We note that an SMPF problem essentially specifies an operator (or functional), i.e., a mapping from the function space specified by $\mathcal{A}, \mathcal{R}$, and $\mathcal{T}$ onto the function space specified by $\mathcal{A}, \mathcal{R}$, and $S$. This view suggests and supports a more compact notation for the expressions. For instance, we may write the marginal function simply as

$$g_S = \sum_{S^c} \prod_{T \in \mathcal{T}} f_T.$$

Here the addition and multiplication of functions are defined in a point-wise manner. This requires that we understand a function $f_T : \mathcal{A}_T \to \mathcal{R}$ as a mapping from $\mathcal{A}$ onto $\mathcal{R}$ for which only the arguments in $\mathcal{A}_T$ are relevant. More precisely, for any $x, x' \in \mathcal{A}$ we have $f_T(x) = f_T(x')$ whenever $x_T = x'_T$. In this case we say that $f_T$ is a function *over* $T$. This way the sum $f_S + f_T$ and the product $f_S\, f_T$ become well defined, the result being a function over $S \cup T$.

The following example is to clarify the SMPF problem representation and notation.

**Example 3.6** We continue Example 3.1. We represent the sum-product problem described in Example 3.1 in the form of Definition 3.5. First, we have $n = 8$ variables with domains $\mathcal{A}_i = \{1, \ldots, m\}$ for $i = 1, 2, 3$. Second, we operate in a commutative semiring $\mathcal{R}$ that is the set of real numbers equipped with ordinal addition and multiplication. We notice that this semiring is also a commutative ring. Third, the set of local scopes is $\mathcal{T} = \{T_1, T_2, T_3, T_4\}$, where $T_1 = \{1, 5, 8\}$, $T_2 = \{2, 5, 6\}$, $T_3 = \{3, 6, 7\}$, and $T_4 = \{4, 7, 8\}$. The correspondence is obvious: $a = f_{T_1}$, $b = f_{T_2}$, $c = f_{T_3}$, and $d = f_{T_4}$. Clearly, $\mathcal{T}$ is a cover on $N = \{1, \ldots, n\} = \{1, \ldots, 8\}$. Fourth, the marginal over $S = \{1, 2, 3, 4\}$ is to be computed, implying $S^c = \{5, 6, 7, 8\}$. With this specification the problem of Example 3.1 is an instance $\mathcal{F} = \{f_T : T \in \mathcal{T}\}$ of SMPF$(\mathcal{A}, \mathcal{R}, \mathcal{T}, S)$. A notationally convenient way to write the marginal function is

$$g_{1234} = \sum_{5678} f_{158}\, f_{256}\, f_{367}\, f_{478}.$$

Here 158, for example, is a shorthand for the set $T_1 = \{1, 5, 8\}$. □

To describe SMPF as an algorithmic problem we need to fix the input and output representation. Since a problem instantiation is speficied by a set of functions, it is natural to let the input be a list of function representations. Similarly the output is a single function representation. As the representation we assume a basic, search tree data structure. More precisely, for a scope $T = \{i_1, \ldots, i_r\}$ where $i_1 < \ldots < i_r$, we represent a function $f_T : \mathcal{A}_T \to \mathcal{R}$ by a tree of height $r$. At $(k-1)$-th level of the tree every node has $|\mathcal{A}_{i_k}|$ child nodes, one per each $x_{i_k} \in \mathcal{A}_{i_k}$. Consequently, the tree has $|\mathcal{A}_T|$ leafs, one per each $x_T \in \mathcal{A}_T$. Note that we treat the *problem structure* $(\mathcal{A}, \mathcal{R}, \mathcal{T}, S)$ as a part of the problem specification which is the same for all instances of the SMPF problem.

This brings us to the next issue. How do we measure the computational complexity of SMPF problems? For a single SMPF problem we do this mainly in terms of arithmetic complexity. By arithmetic complexity we mean here the total number of arithmetic operations performed by an algorithm. However, when the nature of the operations so requires, or when table-lookups or similar operations play an important role in the algorithm, or when space complexity is of interest, then these features are incorporated into complexity considerations. For instance, in general we cannot assume that the evaluation of a function $f_T$ at a fixed $x_T$ takes a constant time. Instead, the time is proportional to $|T|$ when using the tree representation. For an algorithm for an SMPF problem we get in this way the arithmetic or time complexity.

To consider asymptotic complexities meaningfully, we need to consider infinite *problem classes*. In general, any set of SMPF instances forms a problem class. However, we consider mainly classes that are unions of SMPF problems. For example, if letting $m = 1, 2, 3, \ldots$ for the problem described in Example 3.6, we get an infinite class of SMPF problems. Similarly we may consider problems on different structures (covers) and different number of variables.

When considering problem classes we face an extended algoritmic problem. The input is not just a set of functions on a fixed structure, but also the structure itself is taken as a part of the input. By structure we mean here the $\mathcal{A}$, $\mathcal{T}$, $S$ (using the notation of Definition 3.5), whereas the commutative semiring is assumed to be fixed. We will see in the next section that the problem is naturally solved in two phases. In the first phase the structure of a given problem is analyzed, with no reference to the actual instance of the problem (i.e., the local functions). That is, one operates with the graphical structure determined by $\mathcal{T}$ and $S$:

**Definition 3.7** The *graph structure* of a problem SMPF$(\mathcal{A}, \mathcal{R}, \mathcal{T}, S)$ is a pair $(G_\mathcal{T}, S)$, where $G_\mathcal{T} = (N, R)$ is an undirected graph with $(i,j) \in R$ if and only if $\{i,j\} \subseteq T$ for some $T \in \mathcal{T}$. □

For an illustration see Figure 3.1. In the second phase the required marginal is evaluated utilizing the results of the structure analysis made in the first phase.

Many interesting and important computational problems can be interpreted as SMPF problems (or problem classes) in a natural way. For examples in signal processing, coding theory, and belief networks, see

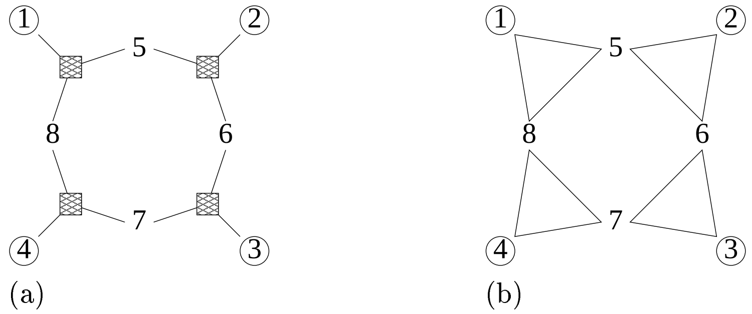


Figure 3.1: Two graphical visualizations of the structure of the problem described in Example 3.6. (a) A graph where a node corresponds to a variable or a local function (boxes, not labelled). There is an edge between a variable node and a function node if the variable is an argument of the function. (b) A graph where a node corresponds to a variable. There is an edge between two nodes if some local function has the corresponding variables as arguments. In both visualizations the marginal variables are indicated by circles.

[AM00, KFL01]. Classical "combinatorial problems", including SAT and MAXSAT, are included in [SI96]. According to Stearns and Hunt [SI96], the representations for these problems are "direct and structure-preserving".

Stearns [Ste03] suggests analyzing sum-product problems in order to establish lower bounds for the problem complexity. For the discrimination of hard and easy problems, he suggests focusing on cases where the primitive variable domains are small, say $|\mathcal{A}_i| = 2$, and where the local scopes are small, say $|T| \leq 3$ for all $T \in \mathcal{T}$. This proposal is understandable when one is interested in analyzing how hard problems may arise from a seemingly simple system, or, which constraints are needed to ensure that there exists an efficient (polynomial) algorihm for the problem. In this work we do *not* follow this idea, since we are also interested in cases where the input size may be very large, even exponential in $n$. For such problems we seek fast polynomial algorithms with respect to the input size, though often exponential in $n$.

Even more general sum-product problem formulations than that of SMPF have been presented. The problem of computing multiple marginals, called "marginalize-a-product-function" (MPF) is introduced in [AM00]. That problem has important applications and can be solved, using the so called "junction trees", more efficiently than by simply solving multiple SMPF problems. In different guises this problem and method appear also in [SI96, KFL01]. From the perspective of this chapter, however, the difference to the SMPF problem defined above is not important. Yet, we reserve the abbreviation MPF for that problem, and consequently we use the modifier "single" for our restricted problem. Still slighly more general than MPF are the formulations in [Dec96, LJ97]. A different, interesting extension is that of Stearns and Hunt [SI02] where more than two operations, e.g., ordinal "max", "+", and "$\cdot$" on $\mathbb{R}$, are considered simultaneously.

## 3.2 The variable elimination algorithm

Since important special cases of the SMPF problem have appeared for a long time in different contexts, it is not a surprise that the same solution has been discovered many times independently. Various versions of essentially the same technique have appeared under many names: Pearl's algorithm [Pea88], bucket elimination [Dec96], generalized distributive law [AM00]. The same method appears in different guises also in [SI96, LJ97, KFL01]. Here we call the method the *variable elimination algorithm*.

As hinted by its name, the method proceeds iteratively marginalizing the variables one by one. Accordingly, the method follows some ordering of the variables.

**Definition 3.8** An *elimination ordering* on a set $U$ is a linear order on $U$. □

A marginalization eliminates a variable. What is left after eliminating a variable is again an instance of an SMPF problem, but over a less number of variables.

A formal description of the variable elimination algorithm is given below (Algorithm 3.9). This description of the variable elimination algorithm borrows much from that of Dechter [Dec96]. The main difference is that here we use a rigorous set representation whereas Dechter somehat loosely mixes (ordered) lists and (unordered) sets. A consequence of this choice is that duplicates of local scopes are forbidden. That is, for each local scope $T$ we can have exactly one local function $f_T$ in the collection $\mathcal{F}$. To satisfy this, at each elimination step of the algorithm, a special routine is performed (line 4 of Algorithm 3.9).

**Algorithm 3.9** (the variable elimination algorithm)
**Input** An instance $\mathcal{F} = \{f_T : T \in \mathcal{T}\}$ of SMPF$(\mathcal{A}, \mathcal{R}, \mathcal{T}, S)$ and an elimination ordering $\prec$ on $S^c$.
**Output** The marginal function $g_S$ as described in Definition 3.5.
**Method**

**for** each $i \in S^c$ in the order $\prec$ **do**
    $\mathcal{T}' \leftarrow \{T \in \mathcal{T} : i \in T\}$ % Local scopes that cover $i$
    $S' \leftarrow \bigcup \mathcal{T}' - \{i\}$ % Covered variables not to be eliminated
    $\mathcal{T}' \leftarrow \mathcal{T}' \cup \{T \in \mathcal{T} : T \subseteq S'\}$ % To avoid duplicates
    $f_{S'} \leftarrow \sum_i \prod_{T \in \mathcal{T}'} f_T$ % Eliminate $i$th variable
    $\mathcal{T} \leftarrow \mathcal{T} \cup \{S'\} - \mathcal{T}'$ % Update local scopes
    $\mathcal{F} \leftarrow \mathcal{F} \cup \{f_{S'}\} - \{f_T : T \in \mathcal{T}'\}$ % Update functions, used implicitly
**return** $\prod_{T \in \mathcal{T}} f_T$

The next example shows a run of the variable elimination algorithm.

**Example 3.10** We continue examples 3.1 and 3.6. Recall that the problem is to compute the marginal function

$$g_{1234} = \sum_{5678} f_{158}\, f_{256}\, f_{367}\, f_{478}.$$

We list the elimination steps of the variable elimination algorithm when run on the elimination ordering $8 \prec 7 \prec 6 \prec 5$. At the first iteration we have $S' = 1457$ and the elimination step takes the form

$$f_{1457} \leftarrow \sum_{8} f_{158}\, f_{478}.$$

Note that if the problem was slightly different so that there was already a local function $f_{1457}$, then this function would be also included in the summation on the right-hand side. This is to avoid duplicated local scopes and to adhere to the set representation. The collection of local scopes (the cover) and local functions are updated to $\mathcal{T} = \{158, 478, 1457\}$ and $\mathcal{F} = \{f_{158}, f_{478}, f_{1457}\}$, respectively. The subsequent three iterations correspond to elimination steps

$$\begin{aligned} f_{13456} &\leftarrow \sum_{7} f_{367}\, f_{1457}, \\ f_{12345} &\leftarrow \sum_{6} f_{256}\, f_{13456}, \\ f_{1234} &\leftarrow \sum_{5} f_{12345}. \end{aligned}$$

This $f_{1234}$ is finally returned by the algorithm. □

**Theorem 3.11** Algorithm 3.9 is correct.

**Proof** Let $g_S$ be the marginal function to be computed. Let $S^c = \{i_1, \ldots, i_r\}$ with $i_1 \prec i_2 \prec \cdots \prec i_r$. We prove by induction on $k = 1, 2, \ldots, r$ that after line 7 in $k$th iteration we have

$$g_S = \sum_{C_k} \prod_{T \in \mathcal{T}} f_T, \tag{3.1}$$

where $C_k = S^c - \{i_1, \ldots, i_k\}$.

Consider the general case, $1 \leq k \leq r$. We suppose that in the beginning of the $k$th iteration the program variable $\mathcal{T}$ is such that the induction assumption (3.1) holds for $k - 1$. Note that for $k = 1$ this holds trivially. Consider the collection $\mathcal{T}$ when the algorithm performs lines 2 to 5 in the $k$th iteration. Divide $\mathcal{T}$ into two parts: $\mathcal{T}'$ as specified on lines 2–4 and the remaining part $\mathcal{T}'' = \mathcal{T} - \mathcal{T}'$. Now we may write

$$g_S = \sum_{C_{k-1}} \prod_{T \in \mathcal{T}} f_T = \sum_{C_k} \Big( \prod_{T \in \mathcal{T}''} f_T \sum_{i_k} \prod_{T \in \mathcal{T}'} f_T \Big),$$

by the commutativity of the addition and multiplication, and by the distributive law. We note that $\sum_{i_k} \prod_{T \in \mathcal{T}'} f_T$ is a function over $\bigcup \mathcal{T}' = S'$, where $S'$ as defined on line 3. Thus, by the substitutions on lines 5–6 we get that

$$g_S = \sum_{C_k} \prod_{T \in \mathcal{T}''} f_T \, f_{S'} = \sum_{C_k} \prod_{T \in \mathcal{T}} f_T.$$

Recall that $\mathcal{T}$ refers here to the updated set of local scopes, not to the original one. This completes the induction.

Finally, note that after the last, $r$th iteration we have $C_r = \emptyset$, and hence $g_S = \prod_{T \in \mathcal{T}} f_T$. This is precisely what the algorithm outputs on line 8, thus completing the proof. □

The complexity of the variable elimination algorithm is determined by the size of the local scopes that are generated along the way. This in turn depends on the elimination ordering that is an input of the algorithm. To give a more precise characterization we need to define some quantities induced by the variable elimination algorithm.

**Definition 3.12** Consider a run of Algorithm 3.9 on a problem $\mathrm{SMPF}(\mathcal{A}, \mathcal{R}, \mathcal{T}, S)$ and an elimination ordering $\prec$ on $S^c$. For each $i \in S^c$

denote by $S'_i$ the corresponding set $S'$ defined on line 3. The *induced maximum width*, *induced maximum size*, and *induced total size* are defined as

$$w_{\max} = \max_{i \in S^c} |S'_i \cup \{i\}|, \quad s_{\max} = \max_{i \in S^c} |\mathcal{A}_{S'_i \cup \{i\}}|, \quad s_{\text{tot}} = \sum_{i \in S^c} |\mathcal{A}_{S'_i \cup \{i\}}|,$$

in the respective order. □

These induced quantities play central role in the complexity of the variable elimination algorithm. A minor contribution to the complexity is due to the need of computing a product of the local functions. The following result shows that local functions can be multiplied, added, and marginalized with a relatively low cost, in linear time with respect to the size of the input and output. This observation does not appear in related studies (e.g., [SI96, AM00, KFL01]) where a constant access array representation of the functions is assumed either implicitly or explicitly.

**Proposition 3.13** Suppose that the arithmetic operations on a semiring $\mathcal{R}$ can be performed in a constant time. Then, given tree representations of two functions $f_S : \mathcal{A}_S \to \mathcal{R}$ and $f_T : \mathcal{A}_T \to \mathcal{R}$, the tree representations of the sum $f_S + f_T$ and the product $f_S\, f_T$ can be formed in time $\mathcal{O}(|\mathcal{A}_{S \cup T}|)$. Similarly, the tree representation of the marginal $\sum_T f_{S \cup T}$ of a function $f_{S \cup T} : \mathcal{A}_{S \cup T} \to \mathcal{R}$ can be formed in time $\mathcal{O}(|\mathcal{A}_{S \cup T}|)$.

**Proof** We first sketch a simple algorithm for the summation problem—the multiplication problem can be handled similarly. First form a tree for a function $f_{S \cup T}$ over $S \cup T$. Then traverse the three trees simultaneously in a straightforward depth-first manner so that for every $x_{S \cup T} \in \mathcal{A}_{S \cup T}$ the corresponding leafs of the three trees can be pointed to in a constant time. While traversing, for each $x_{S \cup T}$ put $f_{S \cup T}(x_{S \cup T}) \leftarrow f_S(x_S) + f_T(x_T)$ in a constant time.

Consider then the computation of the marginal. Let $R = S \cup T - T$ so that $R$ and $T$ are disjoint and $f_R = \sum_T f_{R \cup T}$ denotes the marginal function. We sketch an algorithm similar to that above. First form a tree for the function $f_R$ and initialize the values to zero. Then traverse the two trees of $f_R$ and $f_{R \cup T}$ simultaneously depth-first so that for every $x_{R \cup T} \in \mathcal{A}_{R \cup T}$ the corresponding leafs of the two trees can be pointed to in a constant time. While traversing, for each $x_{R \cup T}$ update $f_R(x_R) \leftarrow f_R(x_R) + f_{R \cup T}(x_{R \cup T})$ in a constant time. □

For the complexity of the variable elimination algorithm we obtain an upper bound easily. The following bound involves the number of local funcions, that is, the size of the cover.

**Theorem 3.14** The variable elimination algorithm runs in time $\mathcal{O}((n + |\mathcal{T}|)s_{\max})$, provided that the arithmetic operations can be performed in a constant time.

**Proof** Consider first the number of additions. The total number of additions performed by the algorithm is clearly $\mathcal{O}(s_{\text{tot}})$, and hence $\mathcal{O}(n\, s_{\max})$. Then consider the number of multiplications. Let the ordering be $i_1 \prec i_2 \prec \cdots \prec i_r$. Let $t_h$ denote the size of $\mathcal{T}'$ at $h$th iteration. Similarly, let $t_0$ denote the number of local scopes after the $r$ iterations. Since every elimination step combines $t_h$ functions into a single function, we have

$$t_0 + \sum_{h=1}^{r}(t_h - 1) = |\mathcal{T}|.$$

Thus the total number multiplications is bounded by $|\mathcal{T}|s_{\max}$. The total number of arithmetic operations performed by the algorithm is thus $\mathcal{O}((n+|\mathcal{T}|)s_{\max})$. By Proposition 3.13 this bound holds also for the running time, completing the proof. □

An apparent shortcoming of the above analysis is that it is conditional on the elimination ordering and a single SMPF problem. When considering a class of SMPF problems, a preliminary algorithmic phase for finding a good, if not optimal, elimination ordering for a given SMPF problem needs to be introduced. Running Algorithm 3.9 is then the second phase. This way, handling SMPF problems can be naturally divided into a preliminary phase that operates only on the problem structure, and a subsequent phase that performs artihmetic computations on a semiring.

An optimal elimination ordering can be defined with respect to any of the above defined three complexity measures. An optimal elimination ordering in turn induces the optimal value of the corresponding complexity measure.

**Definition 3.15** For an SMPF$(\mathcal{A}, \mathcal{R}, \mathcal{T}, S)$ define the *maximum width*, denoted by $w^*_{\max}$, as the minimum of the maximum widths induced by different elimination orderings on $S^c$. Similarly define *maximum size* and *total size*, denoted by $s^*_{\max}$ and $s^*_{\text{tot}}$, respectively. □

Let it be noted that the maximum width appears under many names. For example, Becker and Geiger [BG01] uses term “cliquewidth”. A more commonly used quantity is the “treewidth” which is defined as the maximum width minus 1 [SI96]. Stearns and Hunt [SI96] also use terms “weighted depth” and ‘channelwidth’ to refer similarly defined quantities.

Finding an optimal elimination ordering is hard. In the class of all SMPF problems, finding an optimal elimination ordering is NP-complete [ACP87]. Interestingly, however, for a case where the maximum width is $w^*$ there exits an $\mathcal{O}(n^{w^*+1})$ algorithm that finds an optimal ordering with respect to the maximum width [ACP87]. This bound is polynomial in $n$ if the maximum width is bounded by a constant. But this is, of course, not always the case. Consequently, several approximation algorithms have been designed to find close to optimal elimination orderings (see, e.g., [BG01], and references therein). We do not consider any approximation techniques here.

An interesting question is posed by Becker and Geiger [BG01]: does there exists an algorithm that finds the optimal elimination ordering (say, with respect to the maximum width) as fast as the variable elimination algorithm runs on the optimal elimination ordering? The requirement becomes $\mathcal{O}(n2^{w^*})$, since we may assume binary primitive domains and simple semiring operations. This question can be regarded as a highly relevant open problem. Some hope for a positive answer is provided in [Bod96] where an $\mathcal{O}(nf(w^*))$ algorithm is given, where $f$, however, is a superexponential function of $w^*$.

We conclude this section by analysing the complexity of the variable elimination algorithm for the problem considered in the earlier examples.

**Example 3.16** We continue Example 3.10. Recall that the problem is to compute the marginal function

$$g_{1234} = \sum_{5678} f_{158}\, f_{256}\, f_{367}\, f_{478}.$$

Consider the elimination algorithm on different elimination orderings. We note that there are $4! = 24$ different orderings. However, we can easily find an optimal elimination order, since there is a plenty of symmetry in the structure. By the symmetry of the variable labels, we may assume that the first element in the order is 8. Now in Algorithm 3.9 we have $S' = 1457$ and the elimination step takes the form

$$f_{1457} \leftarrow \sum_{8} f_{158}\, f_{478}.$$

We see that the computation of $f_{1457}$ requires roughly $m^5$ arithmetic operations. The remaining problem may be written as

$$g_{1234} = \sum_{567} f_{256}\, f_{367}\, f_{1457}.$$

For the second step we have three variables where to choose from. However, by the symmetry of 5 and 7 it is sufficient to consider two case: $x_7$ next or $x_6$ next. Consider first continuing by elimination of $x_7$. Now $S' = 13456$ and the elimination step results in a function $f_{13456}$. The remaining problem is written as

$$g_{1234} = \sum_{56} f_{256} f_{13456}.$$

Now 5 and 6 are symmetric in the sense that their elimination order does not affect on the required number of operations. Eliminating $x_6$ first results in a function $f_{12345}$. Finally, eliminating $x_5$ result in the function $f_{1234} = g_{1234}$. We conclude that the elimination ordering $8 \prec 7 \prec 6 \prec 5$ yields the induced total size $m^5 + m^6 + m^6 + m^5$.

On the other hand, eliminating $x_6$ as the second step results in a function $f_{2357}$ for $S' = 2357$. The remaining problem is then written as

$$g_{1234} = \sum_{57} f_{2357} f_{1457}.$$

Now 5 and 7 are symmetric. Eliminating $x_7$ first results in a function $f_{12345}$. Finally, eliminating $x_5$ result in the function $f_{1234} = g_{1234}$. We conclude that the elimination ordering $8 \prec 6 \prec 7 \prec 5$ yields the induced total size $m^5 + m^5 + m^6 + m^5$ which is slighly less than what we got for the ordering we considered first. $\square$

## 3.3 Boosting the variable elimination algorithm

Is the variable elimination algorithm optimal when it is run with an optimal elimination ordering? For commutative semirings in general this seems to be a largely open problem. Interestingly, for commutative rings the answer is negative. To see this, just notice that the ordinal multiplication of two $m \times m$ matrices is a SMPF problem (class). On one hand, it is easy to see that for the matrix multiplication problem the maximum width equals three, and hence, the total size and the arithmetic complexity of the algorithm is $\Omega(m^3)$. On the other hand, the fast matrix multiplication methods require only $\mathcal{O}(m^\omega)$ arithmetic operations, where $\omega < 3$ due to Strassen's [Str69] ingenious recursive algorithm.

This suggests that the efficiency of the variable elimination can be boosted by applying the fast matrix multiplication methods. In fact, one might exploit a relatively wide repertoire of fast matrix multiplication results. Below we recall some most recent and interesting ones.

**Theorem 3.17** ([CW90]) There is an algorithm for multiplying two $m \times m$ square matrices with complexity $\mathcal{O}(m^{2.38})$. □

**Theorem 3.18** ([Cop97]) Let $\tau < 0.294$. For arbitrary $\varepsilon > 0$ there is an algorithm for multiplying an $m \times m$ square matrix by an $m \times m^{\tau}$ rectangular matrix with complexity $\mathcal{O}(m^{2+\varepsilon})$. □

**Theorem 3.19** ([HP98]) There is an algorithm for multiplying an $m \times m$ square matrix by an $m \times m^2$ rectangular matrix with complexity $\mathcal{O}(m^{3.34})$.□

In this section we discuss how these results can be applied to boost the variable elimination algorithm in the context of commutative rings. As far as the author knows, the general connections of rectangular matrix multiplication and sum-product problems have not been studied earlier. Thus, the conceptual and technical development in the rest of this section, may present some novel, yet fairly straightforward, findings.

How much can the elimination algorithm be accelerated by the fast matrix multiplication methods? An obvious way to (asymptotically) accelerate the elimination algorithm is to replace the single-variable marginalization steps by Strassen-type fast matrix multiplication algorithms. This approach is possible, though nontrivial, even when the SMPF problem is defined on a semiring that is not itself a ring. In that case, one may be able to embed the semiring into a ring with a little computational overhead. This is the idea, for instance, in the fast Boolean matrix multiplication and its extensions recently developed for the all-pairs-shortest-paths problem [AGM97, Zwi02]. However, by this approach one may hope to get only a marginal improvement on the variable elimination algorithm when the primitive domains $\mathcal{A}_i$ are large. Especially, if the variables are binary-valued, i.e., $|\mathcal{A}_i| = 2$ for all $i$, then no acceleretation is possible.

We next show that based on a different technique the ordinal variable elimination algorithm may sometimes be significantly accelerated. The trick we use is extreamly simple. It is based on the observation that in some cases the sequential elimination of variables requires no less artihmetic operations than if eliminating the variables as a block; see Example 3.23 below. But the opposite may hold: a block of variables can be eliminated efficiently using the fast matrix multiplication. A more precise description of the method follows.

**Definition 3.20** A *blocked elimination ordering* on a set $S$ is a pair $(\mathcal{B}, \prec)$, where $\mathcal{B}$ is a partition of $S$ and $\prec$ is a linear order on $\mathcal{B}$. □

The algorithm is a straightforward extension of the ordinal variable elimination algorihm (Algorithm 3.9).

**Algorithm 3.21** (the boosted variable elimination algorithm)
**Input** An instance $\mathcal{F} = \{f_T : T \in \mathcal{T}\}$ of SMPF$(\mathcal{A}, \mathcal{R}, \mathcal{T}, S)$ and a blocked elimination ordering $(\mathcal{B}, \prec)$ on $S^c$.
**Output** The marginal function as described in Definition 3.5.
**Method**

1 **for** each $B \in \mathcal{B}$ in the order $\prec$ **do**
2 $\quad \mathcal{T}' \leftarrow \{T \in \mathcal{T} : B \cap T \neq \emptyset\}$ % Local scopes that overlap $B$
3 $\quad S' \leftarrow \bigcup \mathcal{T}' - B$ % Covered variables not to be eliminated
4 $\quad \mathcal{T}' \leftarrow \mathcal{T}' \cup \{T \in \mathcal{T} : T \subseteq S'\}$ % To avoid duplicates
5 $\quad f_{S'} \leftarrow \sum_B \prod_{T \in \mathcal{T}'} f_T$ % Eliminate $x_B$, possibly via fast MM
6 $\quad \mathcal{T} \leftarrow \mathcal{T} \cup \{S'\} - \mathcal{T}'$ % Update local scopes
7 $\quad \mathcal{F} \leftarrow \mathcal{F} \cup \{f_{S'}\} - \{f_T : T \in \mathcal{T}'\}$ % Update functions, used implicitly
8 **return** $\prod_{T \in \mathcal{T}} f_T$

**Theorem 3.22** Algorithm 3.21 is correct.

**Proof** Similar to the proof of the correctness of the variable elimination algorithm (Theorem 3.11) and therefore omitted. □

When is the above algorithm more efficient than the ordinal variable elimination algorihm? The possible gain depends on the structure of the subproblems formed and solved at each elimination round. To analyse this, we have to consider in more detail the elimination step on line 5 of Algorithm 3.21. Before doing this, let us revisit our concrete example. In that example problem we get a considerable reduction in the arithmetic complexity.

**Example 3.23** We continue Example 3.16. Recall that the problem is to compute the marginal function

$$g_{1234} = \sum_{5678} f_{158}\, f_{256}\, f_{367}\, f_{478}.$$

Also recall that in Example 3.16 we showed that the variable elimination algorithm requires $\mathcal{O}(m^6)$ arithmetic operations.

Consider now the boosted elimination algorithm on the blocked elimination ordering $8 \prec 6 \prec 57$. Recall, again from Example 3.16, that the elimination of the variables $x_8$ and $x_6$ can be done using $\mathcal{O}(m^5)$ arithmetic operations, and that the remaining subproblem is to compute

$$g_{1234} = \sum_{57} f_{1457}\, f_{2357}.$$

Now treating the block 57 as a "super variable" is advatageous, since by the fast matrix multiplication this last elimination step requires only $\mathcal{O}((m^2)^\omega)$

arithmetic operations. Thus, by Theorem 3.17 the total arithmetic complexity of the boosted elimination algorithm is $\mathcal{O}(m^5)$, which is less than the complexity $\mathcal{O}(m^6)$ of the ordinal algorithm.

In fact, the complexity is even better. Namely, the elimination of the variable $x_8$, similarly $x_6$, can be viewed as a multiplication of a $m^2 \times m$ matrix by a $m \times m^2$ matrix. Hence, by Theorem 3.19 the first two elimination steps together take $\mathcal{O}(m^{4.34})$ operations. Thus, the last block elimination dominates yielding the total complexity of $\mathcal{O}(m^{4.76})$. □

We now consider the elimination step of the boosted variable elimination algorithm in more detail. Consider first an ideal case where maximum acceleration is possible. This is the case if $S'$ happens to have a partition $\{A, C\}$ such that the product of local functions decomposes as

$$\prod_{T \in \mathcal{T}'} f_T = \Big( \prod_{T \in \mathcal{T}' : T \cap C = \emptyset} f_T \Big) \Big( \prod_{T \in \mathcal{T}' : T \cap A = \emptyset} f_T \Big).$$

This gives a matrix multiplication representation to the elimination step, and fast square or rectangular multiplication methods can be applied. Unfortunately, in general, there need not exist such a partition, in which case the situation is more complicated. Consider, for example, an elimination step of the form

$$f_{123} \leftarrow \sum_4 f_{124}\, f_{134}\, f_{234}. \tag{3.2}$$

In this example no acceleration seems to be possible. One might first think that it is the graph structure of the problem that determines whether boosting by fast matrix multiplication is possible or not. This view is incorrect. For example, the graph structure of

$$f_{123} \leftarrow \sum_4 f_{12}\, f_{13}\, f_{23}\, f_{14}\, f_{24}\, f_{34} \tag{3.3}$$

is the same as the structure of the above problem (3.2). Yet, the elimination step can be reorganized as

$$f_{123} \leftarrow f_{12}\, f_{13}\, f_{23} \sum_4 (f_{14}\, f_{24})(f_{34}), \tag{3.4}$$

which essentially corresponds to a rectangular matrix multiplication.

We conclude this discussion of the boosted elimination step with some summarizing remarks. The pattern how the fast matrix multiplication is most efficiently applied to a particular subproblem can be complicated. It is determined by the problem structure, but not solely by the graph structure.

Accordingly, finding the way to apply the matrix multiplication, if possible, is rather a matter of the structure analysis, than an issue to be solved by the elimination algorithm for a fixed SMPF problem. Consequently, finding the most efficient matrix multiplication patterns, if any, should be incorporated into the preliminary phase, where problem structure is analyzed in order to find a good blocked elimination ordering. Also, note that the best blocked elimination ordering need not be a "segmentation" of optimal variable elimination ordering.

We next sketch a simple algorithm for finding an optimal blocked elimination ordering. Let $S \subseteq N = \{1, \dots, n\}$ be the marginal scope. For any $A \subseteq S^c$ let $g(A)$ denote the mimimum number of arithmetic operations required by the boosted elimination algorithm for eliminating the variables over $A$ when run on a blocked ordering whose first $|A|$ elements form the set $A$. Furthermore, for any $B \subseteq S^c - A$ let $f(B, A)$ denote the number of operations required at the subsequent elimination step when the variables over $B$ are eliminated as a block. We note that the number $f(B, A)$ does not depend on the specific blocked ordering chosen for the precending variables over $A$. Therefore, for all $A \subseteq S^c$ we have the recurrence

$$g(A) = \min_{B \subseteq A} \{f(B, A - B) + g(A - B)\}.$$

On the boundary, of course, $g(\emptyset) = 0$ by the definition of $g$. Assume now that the function $f$ is precomputed. Then the above recurrence for $g(S^c)$ can be solved and an optimal blocked elimination ordering can be extracted by stardard dynamic programming techniques. The running time can be shown to be $\mathcal{O}(3^{|S^c|})$ and hence $\mathcal{O}(3^n)$. However, the total complexity is dominated by the complexity of computing the function $f$, since there are about $3^{|S^c|}$ values to be computed. It seems that the values can be computed in a time that is a low-degree polynomial in the number $n$ of variables and in the number $|\mathcal{T}|$ of local scopes.

Note that the above sketch remains valid with little changes if the size of a variable block is bounded by some parameter, say $\ell$. Specially in the case $\ell = 1$, we obtain an algorithm that finds an optimal elimination ordering for the ordinal variable elimination algorithm. Assuming $|\mathcal{T}| = \mathcal{O}(n)$ a calculation suggests the total time complexity of $\mathcal{O}(n2^n)$ for this special case.

Let us finally consider how much fast matrix multiplications can boost the elimination algorithm when all variables are binary-valued. Not surprisingly, it is easy to find an infinite sequence of SMPF problems on binary variables so that the reduction in arithmetic complexity is significant. Just treat the multiplication of two $2^k \times 2^k$ matrices as a marginalization of a product of two $2k$-dimensional functions. The improvement is from $\mathcal{O}(2^{3k})$ to $\mathcal{O}(2^{\omega k})$. This construction is, of course, trivial, and just a matter of problem representation. Namely, one could, in the first place, represent the problem in terms of three variables, each with $2^k$ possible values. However, this argument is less valid if the matrix product representation, though present, is hidden due to many smaller local scopes.

## 3.4 Transformation problems

We next consider a special type of sum-product problems we call *transformation problems*. Examples of transformation problems are the discrete Fourier transform, the Yates's transform, and the Möbius transform on subset lattice. We discuss these applications after defining the general transformation problem in which a given function is to be transformed to another similar function by a linear operator.

However, we are here interested in transformations that enjoy a structure that can be exploited when computing the transform. This leads us to consider a SMPF problem whose structure is characterized by a "sliding" sequence of local scopes. A feature of this formulation is that the function to be transformed will be included in a function with a slightly larger scope; see Example 3.26 below. Many interesting well-known transformations, including those mentioned above, can be seen as refinements of a coarser, general transformation problem.

It is convinient to define the general transformation problem (class) in a canonical form, i.e., for a specific labelling of the variables ignoring the isomorphic structures.

**Definition 3.24** A problem SMPF$(\mathcal{A}, \mathcal{R}, \mathcal{T}, S)$ is said to be a *sliding transform* if the number variables is even, $n = 2k$, and $\mathcal{T} = \{\{i, i+1, \ldots, i+k\} : i = 1, \ldots, k\}$ and $S = \{1, \ldots, k\}$. □

We may call an SMPF problem a sliding transform if it matches to the above definition for some permutation of the variable indexes. See Figure 3.2 for the graph structure of a slinding transform on $n = 8$ variables.

How fast can one compute a sliding transform? The following result says that by taking advantage of the problem structure one obtains a linear algorithm. Note that the size of the input in a sliding transform on $2k$ variables is of size $\Theta(k\, m^{k+1})$, provided that each primitive domain has $m$ elements.

**Theorem 3.25** The arithmetic complexity of sliding transforms is $\mathcal{O}(k\, m^{k+1})$, provided $|\mathcal{A}_i| \leq m$.

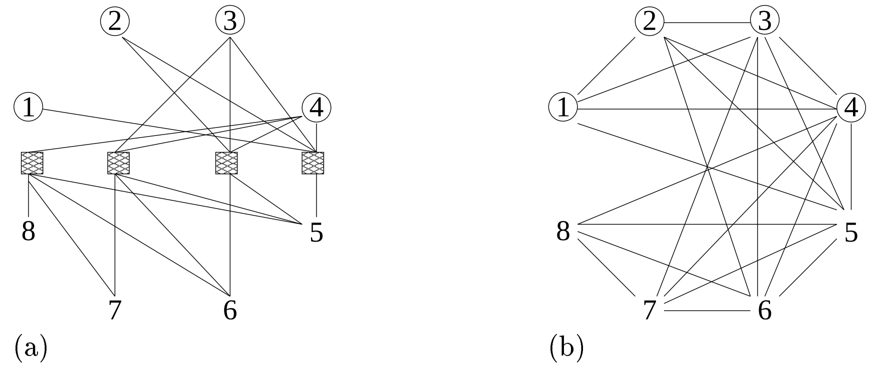


Figure 3.2: Two graphical visualizations of the structure of a sliding transform. (a) A graph where a node corresponds to a variable or a local function (boxes, not labelled). There is an edge between a variable node and a function node if the variable is an argument of the function. (b) A graph where a node corresponds to a variable. There is an edge between two nodes if some local function has the corresponding variables as arguments. In both visualizations the marginal variables are indicated by circles.

**Proof** We consider running the variable elimination on the elimination ordering $n \prec n-1 \prec \cdots \prec k+1$. Clearly, the elimination of the variable $x_n$ corresponds to the elimination step

$$f_{\{n-k,n-k+1,\ldots,n-1\}} \leftarrow \sum_{n} f_{\{n-k,n-k+1,\ldots,n-1,n\}},$$

since $n$ is included by exactly one local scope. Iteratively, for $j < n$, the elimination of $x_j$ corresponds to

$$f_{\{j-k,j-k+1,\ldots,j-1\}} \leftarrow \sum_{j} f_{\{j-k,j-k+1,\ldots,j-1,j\}}\, f_{\{j-k+1,j-k+2,\ldots,j\}}.$$

Thus, by Proposition 3.13, the artihmetic complexity is bounded by $\mathcal{O}(\sum_{j=k+1}^{n} |\mathcal{A}_{\{j-k,j-k+1,\ldots,j-1,j\}}|)$, which is $\mathcal{O}(k\, m^{k+1})$ since $|\mathcal{A}_i| \leq m$. □

Many important computational problems have the structure of a sliding transform and can be solved by the variable elimination algorithm. However, the appropriate SMPF problem representation is not always apparent. A well-known example is the discrete Fourier transform and the fast Fourier transform.

**Example 3.26** The discrete Fourier transform (DFT) of a sequence $f(0), \ldots, f(r-1)$ of complex numbers is the sequence

$$g(s) = \sum_{t=0}^{r-1} f(t)\, \alpha_r(s\, t), \qquad s = 0, 1, \ldots, r-1,$$

where $\alpha_r(q) = e^{q\, 2\pi i / r}$, i.e., the $r$th complex root of unity to the power of $q$. The direct computation takes $\mathcal{O}(r^2)$ time. However, the fast Fourier transform algorithm (FFT) solves the problem in $\mathcal{O}(r \log r)$ time.

We next show that the DFT can be viewed as a special case of a sliding transform, and respectively, the FFT can be viewed as a variable elimination algorithm; a slightly different derivation (for the case $k = 3$) is given in [KFL01]. For simplicity, we let $r = 2^k$ for some $k$. We put $n = 2k$, and introduce binary variables $x_1, \ldots, x_n$ such that $(x_k, \ldots, x_1)$ and $(x_{k+1}, \ldots, x_n)$ are the bit representations of the indices $s$ and $t$ respectively. That is, $s = \sum_{h=1}^{k} 2^{h-1} x_{k+1-h}$ and $t = \sum_{h=1}^{k} 2^{h-1} x_{k+h}$. The key fact used in the FFT is that $\alpha_r(q) = \alpha_r(q \bmod r)$. We observe that

$$\begin{aligned} s\, t \pmod r &= \Big(\sum_{h=1}^{k} 2^{h-1} x_{k+1-h}\Big)\Big(\sum_{h=1}^{k} 2^{h-1} x_{k+h}\Big) \pmod{2^k} \\ &= \sum_{h=1}^{k} 2^{h-1} x_{k+1-h} \sum_{h'=1}^{k+1-h} 2^{h'-1} x_{k+h'}. \end{aligned}$$

Hence,

$$\alpha_r(s\, t) = \prod_{h=1}^{k} \alpha_r(2^{h-1} x_{k+1-h} \sum_{h'=1}^{k+1-h} 2^{h'-1} x_{k+h'}).$$

This suggets defining $k$ local scopes and functions as follows. We let $T_j = \{j, j+1, \ldots, j+k\}$ for $j = 1, \ldots, k$. For $j = 1, \ldots, k-1$ we define

$$f_{T_j}(x_{T_j}) = \alpha_r(2^{k-j} x_j \sum_{h=1}^{j} 2^{h-1} x_{h+k}),$$

Note that only some of the variables appearing on the left hand side appear also on the right hand side. For $j = k$ we include also the function $f$ by putting

$$f_{T_k}(x_{T_k}) = f(x_{k+1}, \ldots, x_n)\, \alpha_r(x_k \sum_{h=1}^{k} 2^{h-1} x_{h+k}),$$

where $f(x_{k+1}, \ldots, x_n) = f(t)$. Note that in this case all the variables appearing on the left hand side appear also on the right hand side.

It is now easy to verify that

$$g = \sum_{\{k+1,\ldots,n\}} \prod_{j=1}^{k} f_{T_j}.$$

Furthermore, we see that the DFT of any function $f$ can be represented as an instance of a sliding transform. By Theorem 3.25 the DFT can be computed in time $\mathcal{O}(k\,2^{k+1})$ which is $\mathcal{O}(r\,\log r)$.

In the above example we saw that it is already the coarser structure of a sliding transform that yields the computational efficiency. In the case of DFT, for instance, the finer problem structure does not give further reduction in the complexity.

However, there are also important transformation problems which enjoy such a finer structure that makes it possible to reduce the computational complexity. The following definition formulates a problem structure where the linear transform operator decomposes into a product of pairwise factors. In matrix algebra such a linear mapping corresponds to a square matrix that is a tensor (or Kronecker) product of smaller matrices. We borrow the name for this type of transform from a related computational technique known as Yates' algorithm due to Yates [Yat37].

**Definition 3.27** A problem SMPF$(\mathcal{A}, \mathcal{R}, \mathcal{T}, S)$ is said to be a *Yates' transform* if the number variables is even, $n = 2k$, and if $\mathcal{T} = \{(i, i+k) : i = 1, \ldots, k\} \cup \{\{k+1, \ldots, 2k\}\}$ and $S = \{1, \ldots, k\}$. □

We may call an SMPF problem a Yates' transform if it matches to the above definition for some permutation of the variable indexes.

**Example 3.28** Let $x_1, \ldots, x_k$ and $y_1, \ldots, y_k$ be variables with the binary domain $\{0, 1\}$. The *Hadamard transform* of a function $f$ of variables $y_1, \ldots, y_k$ is the function $g$ defined by

$$g(x_1, \ldots, x_k) = \sum_{y_1,\ldots,y_k} f(y_1, \ldots, y_k)(-1)^{\sum_{i=1}^{k} x_i y_i},$$

with ordinal addition and multiplication operations. Clearly this is a special case of a Yates' transform. The Hadamard transform has applications, for instance, in digital signal and image processing (see e.g., [?]). □

It is obvious that any instance of a Yates' transform can be (efficiently) be represented as an instance of a sliding transform. Thus, the time complexity of a Yates' transform is at most that of the corresponding sliding transform. For example, computing a Hadamard transform takes $\mathcal{O}(k\,2^k)$ time. Interestingly, for large primitive domains a Yates' transform can be slightly accelerated by the fast matrix multiplication techniques. In the proof of the following result we derive the Yates's algorithm.

**Theorem 3.29** The arithmetic complexity of Yates' transforms is $\mathcal{O}(k\,m^{k+1})$, provided that the cardinality of the primitive domains is at most $m$. In a ring the complexity reduces to $\mathcal{O}(k\,m^{k+0.30})$ for $k \geq 3$ and to $\mathcal{O}(m^{2.38})$ for $k = 2$.

**Proof** The first statement is obvious, since any problem instance can be efficiently mapped to an instance of a sliding transformation problem with the same number of variables and the same primitive domains.

Suppose then that the artihmetic operations are performed on a ring. We consider running the boosted variable elimination algorithm on the (blocked) elimination ordering $n \prec n-1 \prec \cdots \prec k+1$. Clearly, the elimination of the variable $x_n$ corresponds to the elimination step

$$f_{\{n-k,\ldots,n-1\}} \leftarrow \sum_{n} f_{\{n-k,n\}}\, f_{\{n-k+1,\ldots,n\}}.$$

Iteratively, for $j < n$, the elimination of $x_j$ corresponds similarly to

$$f_{\{j-k,\ldots,j-1\}} \leftarrow \sum_{j} f_{\{j-k,j\}}\, f_{\{j-k+1,\ldots,j\}}.$$

Thus, each of the $k$ elimination steps corresponds to a rectangular matrix multiplication. The sizes of the matrices are at most $m \times m$ and $m \times m^{k-1}$. By theorems 3.17 and 3.19 the claimed complexity bound follows. This special case of the variable elimination algorithm (for a Yates' transform) is known as the Yates' algorithm [Yat37], originally designed for binary variables (i.e., $m = 2$). □

The third generic type of transform we consider in this section is the Möbius transform. We start by studying certain key features of a rather generally formulated Möbius transforms. Similarly to a sliding transform or a Yates' transform, a Möbius transform maps a function to another function by a linear operator. The difference is that in a Möbius transform the operator is defined by a graph. After briefly studying an abstract form, we move to a special case where the graph is a subset lattice (definition is given below). An intresting and somewhat nonobvious connection to the Yates' transform is found. This observation is leads to the *fast Möbius transform* (FMT) algorithm. The algorithm itself is not new. A similar, yet much heavier and somewhat nontransparent presentation of the FMT on subset lattice is given in [KS91, Ken91]. Here we give a simplified, gentle, and concise description.

An abstract formulation of the Möbius transform is as follows. Let $V$ be a set and $R \subseteq V \times V$ a relation on $V$. Let $W$ be an arbitrary linear space over $\mathbb{R}$ (the field of real numbers).

**Definition 3.30** The *Möbius transform* of a function $f : V \to W$ on a graph $G = (V, R)$ is the function $f^G : V \to W$ defined by

$$f^G(s) = \sum_{t \in R^{-1}(s)} f(t) \qquad \text{for all } s \in V. \tag{3.5}$$

□

The above definition is sometimes given only for graphs that are partially ordered sets (poset), possibly because then we have the following important result known as the Möbius inversion formula; we omit the proof (see [Aig79, p. 141]).

**Theorem 3.31** For any poset $G = (V, R)$ there exists a unique function $\mu_G : V \times V \to \mathbb{R}$ such that for all $f$,

$$f(s) = \sum_{t \in R^{-1}(s)} \mu_G(t, s) f^G(t) \qquad \text{for all } s \in V. \tag{3.6}$$

□

The function $\mu_G$ above is called the *Möbius function* of $G$. Note that by Möbius transform and Möbius inversion we refer to the two functionals defined respectively in (3.5) and (3.6). Yet, in order to keep the notation relatively simple, we avoid introducing any explicit notation for these two mappings.

The above defined abstract Möbius transform has important special cases. Perhaps the best-known instance is the Möbius transform in number theory. In this case, the set $V$ is the set of integers, the relation $R$ corresponds to "divisibility", and the linear space $W$ is simply the set of complex numbers. Another well-known instance is the Möbius transform on subset lattice, mainly used in the context of belief functions (see e.g. [KS91]. We now focus on the Möbius transforms on subset lattice.

**Definition 3.32** Let $N$ be a finite set. The *subset lattice on* $N$ is the graph $(V, R)$, where $V = \mathcal{P}(N)$ is the collection of subsets of $N$ and $R = \{(T, S) : T \subseteq S \subseteq N\}$ is the inclusion relation. □

Obviously a subset lattice is a poset. The following well-known result gives the associated Möbius function (see e.g., [?, p. 330].

**Proposition 3.33** If $G$ is a subset lattice on $N$, then the Möbius function of $G$ is given by $\mu_G(T, S) = (-1)^{|S-T|}$ for all $T \subseteq S \subseteq N$. □

There is an efficient way to compute a Möbius transform of a given function, known as the *fast Möbius transform.* The key idea is to represent the subsets of $N$ as binary vectors. That is, a subset $S \subseteq N$ is understood as a binary vector $(S_1, \ldots, S_k) \in \{0, 1\}^k$, where $S_i = 1$ iff $i \in S$.

**Algorithm 3.34** (the fast Möbius transform)
**Input** A function $f : \mathcal{P}(N) \to W$, where $N = \{1, \ldots, k\}$ and $W$ is a linear space.
**Output** The transformed function $f^G$.
**Method**

```
f_k ← f
for j = k, k − 1, ..., 1 do
    for each S ⊆ N do
        f_{j−1}(S_1, ..., S_k) ← Σ_{T_j ≤ S_j} f_j(S_1, ..., S_{j−1}, T_j, S_{j+1}, ..., S_k)
return f_0
```

**Theorem 3.35** Algorithm 3.34 works correctly.

**Proof** Straightforward induction gives that the functions $f_j$ computed by the algorithm satisfy

$$f_{j-1}(S_1, \ldots, S_k) = \sum_{T_j \le S_j} \cdots \sum_{T_k \le S_k} f(S_1, \ldots, S_{j-1}, T_j, \ldots, T_k)$$

for all binary vectors $(S_1, \ldots, S_k)$. Hence, in particular,

$$f_0(S) = f_0(S_1, \ldots, S_k) = \sum_{T_1 \le S_1} \cdots \sum_{T_k \le S_k} f(T_1, \ldots, T_k) = \sum_{T \subseteq S} f(T),$$

completing the proof. □

We consider the complexity of Möbius transforms and Möbius inversions with respect to two measures. One is the number of arithmetic operations on the linear space $W$. The second is the total cost of the arithmetic operations when allowing the cost of an addition of $f(S)$ and $f(T)$ depend exponentially on the cardinality of the sets $S$ and $T$.

**Theorem 3.36** Let $N = \{1, \ldots, k\}$. Möbius transforms and inversions on a subset lattice $(\mathcal{P}(N), \subseteq)$ can be computed using $\mathcal{O}(k\,2^k)$ arithmetic operations on the space $W$.

If the time complexity of an addition $f(S) + f(T)$ is bounded by $m^{|S \cup T|}$ for some $m$ for all $S, T \subseteq X$, then Möbius transformation and inversion can be computed in time $\mathcal{O}(k(m+1)^k)$.

**Proof** We note that the arithmetic complexity follows from the time complexity in the special case of $m = 1$. Therefore, it is sufficient to consider the case where the cost of an addition $f(S) + f(T)$ is bounded by $m^{|S \cup T|}$.

We first consider the complexity of a Möbius transform. The analysis of Algorithm 3.34 is straightforward. At each of the $k$ iterations the time requirement is bounded by

$$\sum_{S \subseteq N} m^{|S|} = \sum_{h=0}^{k} \binom{k}{h} m^h = (m+1)^k,$$

up to a constant factor. Namely, the set represented by the binary vector $(S_1, \ldots, S_{j-1}, T_j, S_{j+1}, \ldots, S_k)$ must be a subset of $S$ (not necessarily a proper subset). Thus, we get the claimed bound $\mathcal{O}(k\,(m+1)^k)$.

Consider then the complexity of a Möbius inversion. It is easy to see that Algorithm 3.34 can be easily modified so that it computes the inversion instead of the transform. Namely, replace line 4 by

$$f_{j-1}(S_1, \ldots, S_k) \leftarrow \sum_{T_j \leq S_j} (-1)^{S_j - T_j} f_j(S_1, \ldots, S_{j-1}, T_j, S_{j+1}, \ldots, S_k).$$

Then, by a simple induction,

$$f_0(S) = \sum_{T \subseteq S} (-1)^{\sum_{i=1}^{k} S_i - T_i} f^G(T) = \sum_{T \subseteq S} (-1)^{|S-T|} f^G(T) = f(S),$$

assuming that $f^G$ (not $f$) is given as an input for the algorithm. The complexity is obtained similarly as for the Möbius transform. □

The binary vector representation of the subsets reveals that any Möbius transform and inversion essentially have the structure of a Yates' transform. Why did not we derive the fast Möbius transform algorithm and the complexity results based on Yates' transform? The reason lies in the fact that we define Yates' transforms for semirings. In contrast, Möbius transforms are defined for a linear space over a field of scalars. Extra work would be needed to formally match these two algebraic structures.

## 3.5 On the limitations of the variable elimination algorithm

It is important to note that the SMPF formalism can express only certain type of structural features of computational problems. In applications there is often more structure fixed. Below we discuss some most important structural features that cannot be naturally handled solely by the SMPF formalism and the variable elimination algorithm.

(a) In SMPF problems the marginalization is supposed to be taken over a domain that is a Cartesian product of primitive domains, $\mathcal{A}_{S^c} = \mathcal{A}_{i_1} \times \cdots \times \mathcal{A}_{i_r}$ for $S^c = \{i_1, \ldots, i_r\}$. A natural generalization asks the marginalization over a subset of the Cartesian product, $\mathcal{B}_{S^c} \subseteq \mathcal{A}_{S^c}$. We note, however, that any instance of such a problem can be represented as an instance of a SMPF problem by introducing an extra local function, the indicator function $f_{S^c}(x_{S^c}) = 1_{\{x_{S^c} \in \mathcal{B}_{S^c}\}}$. Such an indicator function may or may not decompose into a product of similar, local functions. In order to represent a problem in a SMPF problem this type of analysis should be done beforehand.

(b) In SMPF problems the local functions are considered indeterminates, that is, arbitrary functions specified by an instance of the problem. However, in many cases some of the local functions are fixed and only some are considered indeterminates. This is the case, for example, with the discrete Fourier transform and the Hadamard transform. Note that item (a) above can be viewed as a special case of this type of additional structure due to "fixed local functions".

(c) It is not possible to express intrinsic structure of local functions in the SMPF formalism. Here we mean functions that are not necessarily totally fixed but that hold some structure, e.g., exchangeability of the arguments. Note that item (b) above can be viewed as a special case of this type of additional structure. This representational lack may be crucial. Namely, employing the variable elimination algorithm ignoring the intrinsic structure may be significantly suboptimal. Some examples are given below.

(d) In SMPF problems the local functions are considered independently. However, in some applications, one function may be associated with many local scopes. For example, in Hadamard function the functions $\alpha(x_i, y_i) = (-1)^{x_i y_i}$ are the same for all pairs $(x_i, y_i)$ of variables. Employing the variable elimination algorithm but ignoring this type of structure may be suboptimal. As an example consider the computation of the $k$th power of an $m \times m$ matrix (over a semiring). The obvious algorithm takes $\mathcal{O}(m^3\, k)$ time, but we know that a doubling technique reduces this to $\mathcal{O}(m^3 \log k)$. See also Example 3.39 below.

The above remarks (a - d) recognize that a sum-product type problem, arising in some application, should be analyzed carefully to find an appropriate SMPF formulation. Such an analysis may lead to introduction of new variables, as we did to get the fast Fourier transform and the fast Möbius transform. It may also be that the whole problem is better solved in some "dual space". A good example of this is the fast convolution algorithm for the discrete convolution problem.

**Example 3.37** The discrete convolution of two sequences $a(0), \ldots, a(r-1)$ and $b(0), \ldots, b(r-1)$ of complex numbers is the sequence

$$c(s) = \sum_{t=0}^{s} a(t)\, b(s-t), \qquad s = 0, 1, \ldots, r-1.$$

This can be trivially mapped to a SMPF problem, but the variable elimination algorithm would take $\mathcal{O}(r^2)$ time. The variable elimination algorithm is not able to exploit the intrinsic structure of the local function $b$. However, there exists a well-known more efficient algorithm that uses the fast Fourier transform. Namely, the convolution theorem (see, e.g., [CLR96, p. 790]) states that

$$c = \mathrm{DFT}^{-1}(\mathrm{DFT}(a) \cdot \mathrm{DFT}(b)),$$

where $a, b$, and $c$ are padded with 0's to length $2r$. Here DFT denotes the discrete Fourier transform mapping and "$\cdot$" denotes the inner product. Using the fast Fourier transform (and inversion) the convolution can be computed in time $\mathcal{O}(r\, \log r)$. □

Another example of the limitations of the SMPF formalism and the variable elimination algorithm is provided by the matrix determinant and permanent.

**Example 3.38** Let $A = (a_{ij})$ be an $n \times n$ matrix. The *determinant* of $A$ is defined as

$$\det(A) = \sum_{\sigma \in S_n} \mathrm{sgn}(\sigma) a_{1\sigma(1)} a_{2\sigma(2)} \cdots a_{n\sigma(n)},$$

where $S_n$ is the set of permutations of the numbers $\{1,\ldots,n\}$ and $\mathrm{sgn}(\sigma)$ is the sign of the permutation $\sigma$. The *permanent* of $A$ is defined as

$$\mathrm{per}(A) = \sum_{\sigma \in S_n} a_{1\sigma(1)} a_{2\sigma(2)} \cdots a_{n\sigma(n)}.$$

The computation of the determinant, e.g., via the Gaussian elimination, takes a polynomial time. In contrast, the fastest algorithm developed for the computation of the permanent takes $\mathcal{O}(n\,2^n)$ time [Knu98, p. 499]. At first glance this difference may be surprising, since the definition of the determinant is seemingly more complex than that of the permanent.

An algorithm that computes the permanent in time $\mathcal{O}(n\,2^n)$ is sketched as follows. For nonempty subsets $S \subseteq N = \{1,\ldots,n\}$ we define recursively

$$g(S) = \sum_{j \in S} a_{|S|j}\, g(S - \{j\}),$$

where on the boundary we define $g(\emptyset) = 1$. Note that here $j$ corresponds to $\sigma(|S|)$. It is not difficult to verify that $g(N) = \mathrm{per}(A)$. A simple calculation yields the claimed time complexity for a straightforward dynamic programming algorithm.

We note that this algorithm performs variable elimination, first eliminating $\sigma(1)$, then $\sigma(2)$, and so forth. Yet, there is no immediate way to define local functions to get an SMPF instance. Even if this was possible, the variable elimination algorithm would not be able to exploit the intrinsic structure of the local functions. The fact that the similarly defined determinant can be computed very efficiently shows that the intrinsic structure may sometimes play surprisingly important role. □

Our last example in this section considers a sum-product type computational problem where various techniques are applied to derive an efficient algorithm. Here we describe and improve an algorithm presented recently in (Kontkanen et al., 2003).

**Example 3.39** Kontkanen et al. [KBM+03] consider the computational complexity of evaluating

$$R_{K,N} = \sum_{x_1 + \cdots + x_K = N} \frac{N!}{x_1! \cdots x_K!} \prod_{i=1}^{K} \left(\frac{x_i}{N}\right)^{x_i},$$

where $K$ and $N$ are positive integers. In a special context of prediction under the principle of worst case relative utility (see Section 2.1), this expression has an interpretation as the normalizing constant of the maximum likelihood distribution for the multinomial distribution, called the *regret* in [KBM+03]. They give an algorithm that achives the time complexity of $\mathcal{O}(N^2 \log K)$ assuming that certain operations with rational numbers can be performed in a constant (amortized) time.

Here we derive Kontkanen's et al. [KBM+03] algorithm from a slightly more generic perspective. We also improve the algorithm significantly using the fast convolution algorithm. Define first

$$f(y) = \frac{(N!)^{1/K}}{y!}\left(\frac{y}{N}\right)^y \qquad \text{for all } y = 0, 1, \ldots, N.$$

Also, for all $j = 1, 2, \ldots, K$ define the function $g_j$ by

$$g_j(m) = \sum_{x_1+\cdots+x_j=m} \prod_{i=1}^{j} f(x_i) \qquad \text{for all } m = 0, 1, \ldots, N.$$

It is now easy to verify that for all integers $K_1, K_2 \geq 1$ such that $K_1 + K_2 = K$, we have recursively,

$$R_{K,N} = g_K(N) = \sum_{N_1+N_2=N} g_{K_1}(N_1) g_{K_2}(N_2),$$

with the boundary $g_1(m) = f(m)$.

How fast can we compute $R_{K,N}$? Suppose first that $K$ is a power of two, say $2^r$. Then it is clearly sufficient to compute the values $g_{2^s}(m)$ for all $m = 0, 1, \ldots, N$ and $s = 0, \ldots, r$. By the above recursion equation, this can be done in $\mathcal{O}(N^2 \log K)$ steps, since the computation of a function $g_{2^s}$ given $g_{2^{s-1}}$ can obviously be done in $\mathcal{O}(N^2)$ steps. If $K$ is not a power of two, it is easy to show that less than $\log K$ additional functions $g_j$ need to be computed, thus yielding the same asymptotic complexity $\mathcal{O}(N^2 \log K)$.

But a significant improvement is possible. We recognize that for any integers $j, j_1, j_2 \geq 1$ such that $j_1 + j_2 = j$ the function $g_j$ is the convolution of the functions $g_{j_1}$ and $g_{j_2}$. Therefore, given $g_{j_1}$ and $g_{j_2}$ the function $g_j$ can be computed via the fast convolution algorithm (see Example 3.37 above) in $\mathcal{O}(N \log N)$ steps. Thus, the complexity of computing $R_{K,N}$ is $\mathcal{O}(N \log N \log K)$. □

We may conlclude this section by noting that the hard part in solving sum-product type problems is finding an appropriate problem representation. Once a good presentation has been found, running the variable elimination algorithm, or its variant, is relatively easy.

## 3.6 Concluding remarks

The general sum-product problem formulation, SMPF, has many appealing features. Various "standard" problems can be represented as sum-product problems in a "direct and structure-preserving" way (Stearns and Hunt, 1996). The generalized variable elimination algorithm provides a unifying approach to tackle these problems, often leading to efficient methods.

But there are also shortcomings. As we discussed in Section 3.5, it is not always apparent, how a particular computational task is best represented as an SMPF problem. To get maximal gain of the sum-product framework in a fixed problem, one may have to alternate between different representations in a dynamic fashion. Though SMPF representations together with the variable elimination algorithm are effective building blocks, it seems that a plenty of creativity may be needed when designing algorithms for a particular application. There are some potentially fruitful extensions of the sum-product framework (e.g., [SI02]) that we did not include in this review.

In this chapter we only studied exact methods. Since the intermediate functions computed by the elimination algorithm may involve many variables, exact computations are no always feasible. Approximate methods can offer a practical solution. However, it seems that a unifying approach for various approximative mehods on different algebraic structures is difficult to find. For example, a complicating issue is how one should quantify approximation errors on different instances of the abstract semiring. In Chapter 4, a special family of Monte Carlo methods on the ordinal real system is studied.

We observed that fast matrix multiplication methods can be used to reduce the complexity of the variable elimination algorithm. Since the asymptotic analysis hides large constant factors, the fast methods do not have practical value until computers can handle very large matrices, say with 10000 rows and columns. More importantly, the simple observation presented here can, at best, only lead to a polynomial complexity reduction, from $\mathcal{O}(\gamma(n,m))$ to $\mathcal{O}(\gamma(n,m)^{\omega/3})$, where $\omega$ is the matrix multiplication exponent ($\omega < 2.38$), $\gamma(n,m)$ is the complexity of the ordinal variable elimination algorithm on a problem parametrized by the number of variables $n$ and the size of the primitive domains $m$. An open question is can other algebraic techniques lead to more dramatic reductions. We formulate a related open problem below. In this chapter we only presented the simple observation; more general and detailed analysis is a topic of future research.

We finally formulate two concrete open problems. First consider a problem related to application of fast matrix multiplication methods. Let $\{R, S, T\}$ be a partition of $\{1, \ldots, n\}$. Recall that the fast matrix multiplication methods apply ideally, when the task is to compute the marginal

$$g_{R \cup T} = \sum_{S} f_{R \cup S} \, f_{S \cup T},$$

for given functions $f_{R \cup S}$ and $f_{S \cup T}$. An open problem is to find an asymptotically efficient way to carry out the marginalization, given that the two function factorize as

$$f_{R \cup S} = \prod_{r \in R} \prod_{s \in S} f_{\{r,s\}} \qquad \text{and} \qquad f_{S \cup T} = \prod_{s \in S} \prod_{t \in T} f_{\{s,t\}}.$$

We see that the fast (rectangular) matrix multiplication is more efficient than the naive variable elimination. Do more efficient (algebraic) methods exist?

The second problem is as follows: For all subsets $S$ of $\{1, \ldots, n\}$ compute

$$c(S) = \sum_{T \subseteq S} a(T) \, b(S - T),$$

where $a$ and $b$ are two given functions. We note that the Möbius transform and inversion are of this form for a special function $b$. Also, we note a seeming similarity to the discrete convolution of two vectors. Do there exist algorithms faster than the straightforward one?