Found Comput Math (2016) 16:1–31
DOI 10.1007/s10208-014-9231-y

FOUNDATIONS OF COMPUTATIONAL MATHEMATICS
The Journal of the Society for the Foundations of Computational Mathematics

Subtraction-Free Complexity, Cluster Transformations, and Spanning Trees

Sergey Fomin · Dima Grigoriev · Gleb Koshevoy

Received: 28 August 2013 / Accepted: 3 April 2014 / Published online: 6 November 2014

Abstract Subtraction-free computational complexity is the version of arithmetic circuit complexity that allows only three operations: addition, multiplication, and division. We use cluster transformations to design efficient subtraction-free algorithms for computing Schur functions and their skew, double, and supersymmetric analogues, thereby generalizing earlier results by P. Koev. We develop such algorithms for computing generating functions of spanning trees, both directed and undirected. A comparison to the lower bound due to M. Jerrum and M. Snir shows that in subtraction-free computations, "division can be exponentially powerful." Finally, we give a simple example where the gap between ordinary and subtraction-free complexity is exponential.

Communicated by Peter Bürgisser.

We thank the Max-Planck Institut für Mathematik for its hospitality during the writing of this paper. Partially supported by NSF Grant DMS-1101152 (S. F.), RFBR/CNRS Grant 10-01-9311-CNRSL-a, and MPIM (G. K.).

S. Fomin (✉)
Department of Mathematics, University of Michigan, 530 Church Street, Ann Arbor, MI 48109-1043, USA
e-mail: fomin@umich.edu
URL: http://www.math.lsa.umich.edu/∼fomin/

D. Grigoriev
CNRS, Mathématiques, Université de Lille, 59655 Villeneuve d'Ascq, France
e-mail: Dmitry.Grigoryev@math.univ-lille1.fr
URL: http://en.wikipedia.org/wiki/Dima_Grigoriev

G. Koshevoy
Central Institute of Economics and Mathematics, Nahimovskii pr. 47, 117418 Moscow, Russia
e-mail: koshevoy@cemi.rssi.ru
URL: http://mathecon.cemi.rssi.ru/en/koshevoy/

Springer

Keywords Subtraction-free · Arithmetic circuit · Schur function · Spanning tree · Cluster transformation · Star–mesh transformation

Mathematics Subject Classification Primary 68Q25 · Secondary 05E05 · 13F60

1 Introduction

This paper is motivated by the problem of dependence of algebraic complexity on the set of allowed operations. Suppose that a rational function f can in principle be computed using a restricted set of arithmetic operations $M \subset \{+, -, *, /\}$; how does the complexity of f (i.e., the minimal number of steps in such a computation) depend on the choice of M? For example, let f be a polynomial with nonnegative coefficients; then it can be computed without using subtraction (we call this a *subtraction-free* computation). Could this restriction dramatically alter the complexity of f? What if we also forbid using division?

One natural test is provided by the *Schur functions* and their various generalizations. Combinatorial descriptions of these polynomials are quite complicated, and the (nonnegative) coefficients in their monomial expansions are known to be hard to compute. On the other hand, well-known determinantal formulas for Schur functions yield fast (but not subtraction-free) algorithms for computing them.

In fact, one *can* compute a Schur function in polynomial time without using subtraction. An outline of such an algorithm was first proposed by Koev [18] in 2007. In this paper, we describe an alternative algorithm utilizing the machinery of *cluster transformations*, a family of subtraction-free rational maps that play a key role in the theory of cluster algebras [11]. We then further develop this approach to obtain subtraction-free polynomial algorithms for computing *skew*, *double*, and *supersymmetric* Schur functions.

We also look at another natural class of polynomials: the generating functions of spanning trees (either directed or undirected) in a connected (di)graph with weighted edges. We use *star–mesh transformations* to develop subtraction-free algorithms that compute these generating functions in polynomial time. In the directed case, this sharply contrasts with the exponential lower bound due to Jerrum and Snir [15] who showed that if one only allows additions and multiplications (but no subtractions or divisions), then the arithmetic circuit complexity of the generating function for directed spanning trees in an n-vertex complete digraph grows exponentially in n. We thus obtain an exponential gap between subtraction-free and semiring complexity, which can be informally expressed by saying that in the absence of subtraction; division can be "exponentially powerful" (cf. Valiant's result [35] on the power of subtraction). Recall that if subtraction is allowed, then division gates can be eliminated at polynomial cost, as shown by Strassen [32]. One could say that forbidding subtraction can dramatically increase the power of division.

Jerrum and Snir [15] have shown that their exponential lower bound also holds in the *tropical semiring* $(\mathbb{R}, +, \min)$ (see, e.g., [19, Section 8.5] and references therein). Since our algorithms extend straightforwardly into the tropical setting, we conclude that the circuit complexity of the *minimum cost arborescence* problem drops from

exponential to polynomial as one passes from the tropical semiring to the *tropical semifield* $(\mathbb{R}, +, -, \min)$.

At the end of the paper, we present a simple example of a rational function f_n whose ordinary circuit complexity is linear in n, whereas its subtraction-free complexity, while finite, grows at least exponentially in n.

The paper is organized as follows. Section 2 reviews basic prerequisites in algebraic complexity, along with some relevant historical background. In Sect. 3 we present our main results. Their proofs occupy the rest of the paper. Sections 4–6 are devoted to subtraction-free algorithms for computing Schur functions and their variations, while in Sects. 7–8, we develop such algorithms for computing generating functions for spanning trees, either ordinary or directed. In Sect. 9, we demonstrate the existence of exponential gaps between ordinary and subtraction-free complexities.

2 Computational Complexity

We start by reviewing the relevant basic notions of computational complexity, more specifically complexity of arithmetic circuits (with restrictions). See [3,13,30] for in-depth treatment and further references.

An *arithmetic circuit* is an oriented network each of whose nodes (called *gates*) perform a single arithmetic operation: addition, subtraction, multiplication, or division. The circuit inputs a collection of *variables* (or indeterminates) as well as some scalars and outputs a rational function in those variables. The *arithmetic circuit complexity* of a rational function is the smallest size of an arithmetic circuit that computes this function.

The following disclaimers further clarify the setup considered in this paper:

- we define complexity as the number of gates in a circuit rather than its depth;
- we do not concern ourselves with parallel computations;
- we allow arbitrary positive integer scalars as inputs.

Although we focus on arithmetic circuit complexity, we also provide *bit complexity* estimates for our algorithms. For the latter purpose, the input variables should be viewed as numbers rather than formal variables.

As is customary in complexity theory, we consider families of computational problems indexed by a positive integer parameter n and only care about the rough asymptotics of the arithmetic complexity as a function of n. The number of variables may depend on n.

Of central importance is the dichotomy between polynomial and superpolynomial (in particular exponential) complexity classes. We use the shorthand $\mathrm{poly}(n)$ to denote the dependence of complexity on n that can be bounded from above by a polynomial in n.

Perhaps, the most important (if simple) example of a sequence of functions whose arithmetic circuit complexity is $\mathrm{poly}(n)$ is the determinant of an n by n matrix. (The entries of a matrix are treated as indeterminates.) The simplest—though not the fastest—polynomial algorithm for computing the determinant is Gaussian elimination.

Table 1 Rational functions computable with restricted set of operations

	No multiplicative operations	Multiplication only	Multiplication and division
No additive operations	Scalars	Monomials	Laurent monomials
Addition only	Nonnegative linear combinations	Nonnegative polynomials	Subtraction-free expressions
Addition and subtraction	Linear combinations	Polynomials	Rational functions

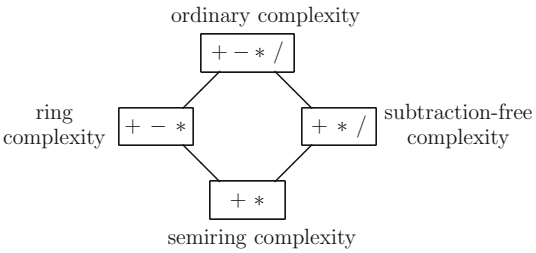


Fig. 1 Notions of M-complexity, with $M \supset \{+, *\}$

In this paper, we are motivated by the following fundamental question: How does the complexity of an algebraic expression depend on the set of operations allowed?

Let us formulate the question more precisely. Let M be a subset of the set $\{+, -, *, /\}$ of arithmetic operations. Let $Z\{M\} = Z\{M\}(x, y, \ldots)$ denote the class of rational functions in the variables $x, y, \ldots$ which can be defined using only operations in M. For example, the class $Z\{+, *, /\}$ consists of *subtraction-free expressions*, i.e., those rational functions which can be written without using subtraction (note that negative scalars are not allowed as inputs). To illustrate, $x^2 - xy + y^2 \in Z\{+, *, /\}(x, y)$ because $x^2 - xy + y^2 = (x^3 + y^3)/(x + y)$.

While the class $Z\{M\}$ can be defined for each of the $2^4 = 16$ subsets $M \subset \{+, -, *, /\}$, there are only 9 distinct classes among these 16. This is because addition can be emulated by subtraction: $x + y = x - ((x - y) - x)$. Similarly, multiplication can be emulated by division. This leaves 3 essentially distinct possibilities for the additive (resp., multiplicative) operations. The corresponding 9 computational models are shown in Table 1.

For each subset of arithmetic operations $M \subset \{+, -, *, /\}$, there is the corresponding notion of (arithmetic circuit) *M-complexity* (of an element of $Z\{M\}$). The interesting cases are those where both additive and multiplicative operations appear, see Fig. 1.

Now, how does M-complexity depend on M, when there is a choice? Here is one way to make this question precise:

Problem 2.1 Let $f_1, f_2, \ldots$ be a sequence of rational functions (depending on a potentially changing set of variables) which can be computed using the gates in $M' \subsetneq M \subset \{+, -, *, /\}$. If the M-complexity of f_n is poly(n), does it follow that its M'-complexity is also poly(n)?

The nontrivial instances of Problem 2.1, discussed in Examples 2.2–2.5 below, concern the four notions of M-complexity that involve both additive and multiplicative operations.

Example 2.2 $M = \{+, -, *, /\}$, $M' = \{+, -, *\}$. In 1973, Strassen [32] (cf. [30, Theorem 2.11]) proved that in this case, the answer to Problem 2.1 is essentially *yes*: division gates can be eliminated (at polynomial cost) provided the total degree of the polynomial f_n is $\mathrm{poly}(n)$.

As a consequence, one for example obtains a division-free polynomial algorithm for computing a determinant. More efficient algorithms of this kind can be constructed directly (ditto for the Pfaffian), see [27] and references therein.

Example 2.3 $M = \{+, -, *\}$, $M' = \{+, *\}$. (In view of Strassen's theorem, this setting is essentially equivalent to taking $M = \{+, -, *, /\}$, $M' = \{+, *\}$). In 1980, Valiant [35] has shown that in this case, the answer to Problem 2.1 is *no*: for a certain sequence of polynomials f_n with nonnegative integer coefficients, the $\{+, *\}$-complexity of f_n is exponential in n whereas their $\{+, -, *\}$-complexity (equivalently, ordinary arithmetic circuit complexity) is $\mathrm{poly}(n)$. The polynomial f_n used by Valiant is defined as a generating function for perfect matchings in a particular planar graph (a triangular grid). By a classical result of Kasteleyn [16], such generating functions can be computed as certain Pfaffians; hence, their ordinary complexity is polynomial.

It is unknown whether subtraction-free complexity of Valiant's test function f_n is $\mathrm{poly}(n)$. If the answer is *yes*, then f_n exhibits a (superpolynomial) complexity gap between subtraction-free and $\{+, *\}$-complexity. If the answer is *no*, then we get a complexity gap between ordinary and subtraction-free complexities. Thus, we have known since Valiant's work that one of these two gaps is present in his example—but we still do not know which one!

Other examples of polynomials f_n which exhibit an exponential gap between ordinary and $\{+, *\}$-complexity were given by Jerrum and Snir [15], cf. Theorem 3.7.

The notion of $\{+, *\}$-complexity of a polynomial with nonnegative coefficients was already considered in 1976 by Schnorr [28] (He used the terminology "monotone rational computations" which we shun). Schnorr gave a lower bound for $\{+, *\}$-complexity which only depends on the *support* of a polynomial, i.e., on the set of monomials that contribute with a positive coefficient. Valiant's argument uses a further refinement of Schnorr's lower bound, cf. [29].

Example 2.4 $M = \{+, -, *, /\}$, $M' = \{+, *, /\}$. In this case, Problem 2.1 asks whether any subtraction-free rational expression that can be computed by an arithmetic circuit of polynomial size can be computed by such a circuit without subtraction gates. In Sect. 9, we show the answer to this question to be negative, by constructing a sequence of polynomials f_n whose ordinary arithmetic circuit complexity is $O(n)$ while their $\{+, *, /\}$-complexity is at least exponential in n. Unfortunately, this example is somewhat artificial; it would be interesting to find an example of a natural computational problem with an exponential gap between ordinary and subtraction-free complexities.

In the opposite direction, we demonstrate that for some important classes of functions, the gap between these two complexity measures is merely polynomial, in a some-

what counter-intuitive way: these functions turn out to have polynomial subtraction-free complexity even though their "naive" subtraction-free description has exponential size.

Note that subtraction is the only arithmetic operation that does not allow for an efficient control of round-up errors (for positive real inputs). Consequently, the task of eliminating subtraction gates is relevant to the design of numerical algorithms which are both efficient and precise. To rephrase, this instance of Problem 2.1 can be viewed as addressing the trade-off between speed and accuracy. See [9] for an excellent discussion of these issues.

Example 2.5 $M = \{+, *, /\}$, $M' = \{+, *\}$. This is the subtraction-free version of the problem discussed in Example 2.2. That is, can division gates be eliminated in the absence of subtraction? We will show that the answer is *no*, by demonstrating that the generating function for directed spanning trees in a complete directed graph on n vertices has poly(n) subtraction-free complexity. This contrasts with an exponential lower bound for the $\{+, *\}$-complexity of the same generating function, given by Jerrum and Snir [15].

3 Main Results

3.1 Schur Functions and Their Variations

Schur functions $s_\lambda(x_1, \ldots, x_k)$ (here, $\lambda = (\lambda_1 \geq \lambda_2 \geq \cdots \geq 0)$ is an integer partition) are remarkable symmetric polynomials that play prominent roles in representation theory, algebraic geometry, enumerative combinatorics, mathematical physics, and other mathematical disciplines; see, e.g., [21, Chapter I] [31, Chapter 7]. Among many equivalent ways to define Schur functions (also called *Schur polynomials*), let us mention two classical determinantal formulas: the bialternant formula and the Jacobi–Trudi formula. These formulas are recalled in Sects. 4 and 6, respectively.

Schur functions and their numerous variations (*skew* Schur functions, *supersymmetric* Schur functions, Q- and P-Schur functions, etc., see *loc. cit.*) provide a natural source of computational problems whose complexity might be sensitive to the set of allowable arithmetic operations. On the one hand, these polynomials can be computed efficiently in an unrestricted setting, via determinantal formulas; on the other hand, their (nonnegative) expansions, as generating functions for appropriate *tableaux*, are in general exponentially long, and coefficients of individual monomials are provably hard to compute, cf. Remark 3.3. (Admittedly, a low-complexity polynomial can have high-complexity coefficients. For example, the coefficient of $x_1 \cdots x_n$ in $\prod_i \sum_j (a_{ij} x_j)$ is the permanent of the matrix (a_{ij}).)

The interest in determining the subtraction-free complexity of Schur functions goes back at least as far as mid-1990s, when the problem attracted the attention of Demmel and the first author, cf. [8, pp. 66–67]. The following result is implicit in the work of Koev [18, Section 6]; more details can be found in [5, Section 4]).

Theorem 3.1 (P. Koev) *Subtraction-free complexity of a Schur polynomial* $s_\lambda(x_1, \ldots, x_k)$ *is at most* $O(n^3)$ *where* $n = k + \lambda_1$.

In this paper, we give an alternative Proof of Theorem 3.1 based on the technology of *cluster transformations*. The algorithm presented in Sect. 4 computes $s_\lambda(x_1, \ldots, x_k)$ via a subtraction-free arithmetic circuit of size $O(n^3)$. The bit complexity is $O(n^3 \log^2 n)$.

All known fast subtraction-free algorithms for computing Schur functions use division.

Problem 3.2 Is the $\{+, *\}$-complexity of a Schur function polynomial?

Remark 3.3 We suspect the answer to this question to be negative. In any case, Problem 3.2 is likely to be very hard. We note that Schnorr-type lower bounds are useless in the case of Schur functions. Intuitively, computing a Schur function is difficult not because of its support but because of the complexity of its coefficients (the Kostka numbers). The problem of computing an individual Kostka number is known to be **#P**-complete (Narayanan [23]), whereas the support of a Schur function is very easy to determine.

Our approach leads to the following generalizations of Theorem 3.1. See Sects. 5 and 6 for precise definitions as well as proofs.

Theorem 3.4 *A double Schur polynomial $s_\lambda(x_1, \ldots, x_k \mid y)$ can be computed by a subtraction-free arithmetic circuit of size $O(n^3)$ where $n = k + \lambda_1$. The bit complexity of the corresponding algorithm is $O(n^3 \log^2 n)$.*

Theorem 3.4 can be used to obtain an efficient subtraction-free algorithm for *supersymmetric* Schur functions, see Theorem 5.4.

Theorem 3.5 *A skew Schur polynomial $s_{\lambda/\nu}(x_1, \ldots, x_k)$ can be computed by a subtraction-free arithmetic circuit of size $O(n^5)$ where $n = k + \lambda_1$. The bit complexity of the corresponding algorithm is $O(n^5 \log^2 n)$.*

Remark 3.6 The actual subtraction-free complexity (or even the $\{+, *\}$-complexity) of a particular Schur polynomial can be significantly smaller than the upper bound of Theorem 3.1. For example, consider the bivariate Schur polynomial $s_{(\lambda_1,\lambda_2)}(x_1, x_2)$ given by

$$s_{(\lambda_1,\lambda_2)}(x_1, x_2) = (x_1 x_2)^{\lambda_2} h_{\lambda_1-\lambda_2}(x_1, x_2),$$

where $h_d(x_1, x_2) = \sum_{1 \le i \le d} x_1^i \cdot x_2^{d-i}$ (the complete homogeneous symmetric polynomial). The polynomial $s_{(\lambda_1,\lambda_2)}(x_1, x_2)$ can be computed in $O(\log(\lambda_1))$ time using addition and multiplication only, by iterating the formulas

$$h_{2d+1}(x_1, x_2) = \left(x_1^{d+1} + x_2^{d+1}\right) h_d(x_1, x_2) \tag{3.1}$$

$$h_{2d+2}(x_1, x_2) = \left(x_1^{d+2} + x_2^{d+2}\right) h_d(x_1, x_2) + x_1^{d+1} x_2^{d+1}. \tag{3.2}$$

3.2 Spanning Trees

We also develop efficient subtraction-free algorithms for another class of polynomials: the generating functions of spanning trees in weighted graphs, either ordinary (undirected) or directed. In the directed case, the edges of a tree should be directed toward the designated root vertex. The weight of a tree is defined as the product of the weights of its edges. See Sects. 7–8 for precise definitions.

Determinantal formulas for these generating functions (the Matrix-Tree Theorems) go back to Kirchhoff [17] (1847, undirected case) and Tutte [33, Theorem 6.27] (1948, directed case). Consequently, their ordinary complexity is polynomial. Amazingly, the $\{+, *\}$-complexity is exponential in the directed case:

Theorem 3.7 (Jerrum and Snir [15, 4.5]) *Let φ_n denote the generating function for directed spanning trees in a complete directed graph on n vertices. Then the $\{+, *\}$-complexity of φ_n can be bounded from below by $n^{-1}(4/3)^{n-1}$.*

In Sects. 7–8, we establish the following results.

Theorem 3.8 *Let G be a weighted simple graph (respectively, simple directed graph) on n vertices. Then the generating function for spanning trees in G (respectively, directed spanning trees rooted at a given vertex) can be computed by a subtraction-free arithmetic circuit of size $O(n^3)$.*

In particular, the $\{+, *, /\}$-complexity of the polynomials φ_n from Theorem 3.7 is $O(n^3)$, in sharp contrast with the Jerrum–Snir lower bound.

4 Subtraction-Free Computation of a Schur Function

This section presents our Proof of Theorem 3.1, i.e., an efficient subtraction-free algorithm for computing a Schur function. The basic idea of our approach is rather simple, provided the reader is already familiar with the basics of cluster algebras. (Otherwise, (s)he can safely skip the next paragraph, as we shall keep our presentation self-contained.)

A Schur function can be given by a determinantal formula, as a minor of a certain matrix, and consequently can be viewed as a specialization of some cluster variable in an appropriate cluster algebra. It can, therefore, be obtained by a sequence of subtraction-free rational transformations (the "cluster transformations" corresponding to exchanges of cluster variables under cluster mutations) from a wisely chosen initial extended cluster. An upper bound on subtraction-free complexity is then obtained by combining the number of mutation steps with the complexity of computing the initial seed.

The most naive version of this approach starts with the classical Jacobi–Trudi formula (reproduced in Sect. 6) that expresses a (more generally, skew) Schur function as a minor of the Toeplitz matrix $(h_{i-j}(x_1, \ldots, x_k))$ where h_d denotes the dth complete homogeneous symmetric polynomial, i.e., the sum of all monomials of degree d. Unfortunately, this approach (or its version employing elementary symmetric polynomials) does not seem to yield a solution: Even though the number of mutation steps

can be polynomially bounded, we were unable to identify an initial cluster all of whose elements are easier to compute (by a polynomial subtraction-free algorithm) than a general Schur function.

The key idea is to employ a different cluster recurrence that iteratively computes Schur polynomials in *varying* number of arguments. This leads us to an algorithm that ultimately relies—as Koev's original approach did [18]—on another classical determinantal formula for a Schur function, which goes back to Cauchy and Jacobi. This formula expresses s_λ as a ratio of two "alternants," i.e., Vandermonde-like determinants. Let us recall this formula in the form that will be convenient for our purposes; an uninitiated reader can view it as a *definition* of a Schur function.

Let n be a positive integer. Consider the $n \times n$ "rescaled Vandermonde" matrix

$$X = (X_{ij}) = \left(\frac{x_j^{i-1}}{\prod_{a<j}(x_j - x_a)} \right)_{i,j=1}^{n} = \begin{pmatrix} 1 & \frac{1}{x_2 - x_1} & \frac{1}{(x_3 - x_1)(x_3 - x_2)} & \cdots \\ x_1 & \frac{x_2}{x_2 - x_1} & \frac{x_3}{(x_3 - x_1)(x_3 - x_2)} & \cdots \\ x_1^2 & \frac{x_2^2}{x_2 - x_1} & \frac{x_3^2}{(x_3 - x_1)(x_3 - x_2)} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}. \tag{4.1}$$

For a subset $I \subset \{1, \ldots, n\}$, say of cardinality k, let s_I denote the corresponding "flag minor" of X, i.e., the determinant of the square submatrix of X formed by the intersections of the rows in I and the first k columns:

$$s_I = s_I(x_1, \ldots, x_k) = \det(X_{ij})_{i \in I, j \le k}. \tag{4.2}$$

(For example, $s_{1,\ldots,k} = \det(X_{ij})_{i,j=1}^{k} = 1$.) It is easy to see that s_I is a symmetric polynomial in the variables $x_1, \ldots, x_k$.

Now, let $\lambda = (\lambda_1, \ldots, \lambda_k)$ be a partition with at most k parts satisfying $\lambda_1 + k \le n$. Define the k-element subset $I(\lambda) \subset \{1, \ldots, n\}$ by

$$I(\lambda) = \{\lambda_k + 1, \lambda_{k-1} + 2, \ldots, \lambda_1 + k\}. \tag{4.3}$$

The Schur function/polynomial $s_\lambda(x_1, \ldots, x_k)$ is then given by

$$s_\lambda(x_1, \ldots, x_k) = s_{I(\lambda)}(x_1, \ldots, x_k) = \det(X_{ij})_{i \in I(\lambda), j \le k}. \tag{4.4}$$

If λ has more than k parts, then $s_\lambda(x_1, \ldots, x_k) = 0$.

We note that as I ranges over all subsets of $\{1, \ldots, n\}$, the flag minors of X range over the nonzero Schur polynomials $s_\lambda(x_1, \ldots, x_k)$ with $\lambda_1 + k \le n$.

Flag minors play a key role in one of the most important examples of cluster algebras, the coordinate ring of the base affine space. Let us briefly recall (borrowing heavily from [10] and glossing over technical details, which can be found in *loc. cit.*) the basic features of the underlying combinatorial setup, which was first introduced in [2]; cf. also [7].

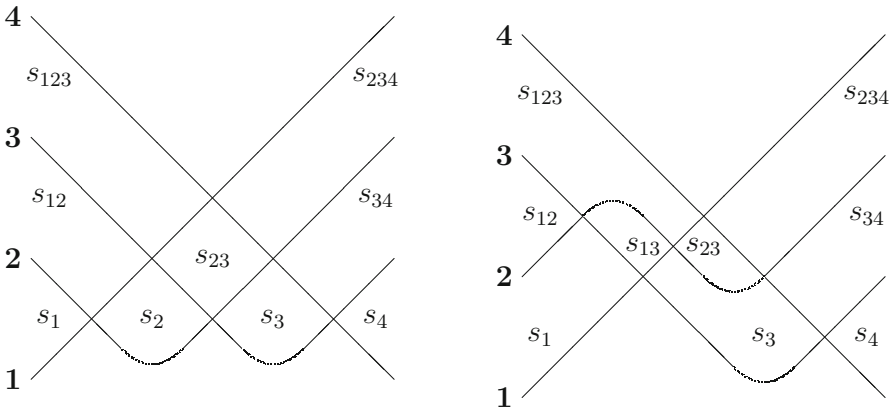
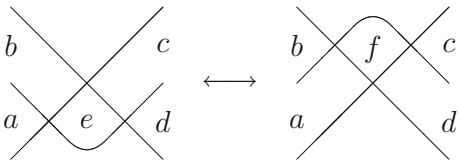


Fig. 2 Two pseudoline arrangements and associated chamber minors

Fig. 3 A local move in a pseudoline arrangement



A *pseudoline arrangement* is a collection of n curves ("pseudolines") each of which is a graph of a continuous function on $[-1, 1]$; each pair of pseudolines must have exactly one crossing point in common; no three pseudolines may intersect at a point. See Fig. 2. The pseudolines are numbered **1** through **n** from the bottom-up along the left border. The resulting pseudoline arrangement is considered up to isotopy (performed within the space of such arrangements).

To each region R of a pseudoline arrangement, except for the very top and the very bottom, we associate the flag minor $s_{I(R)}$ indexed by the set $I(R)$ of labels of the pseudolines passing below R. These are called *chamber minors*.

Pseudoline arrangements are related to each other via sequences of *local moves* of the form shown in Fig. 3. Each local move results in replacing exactly one chamber minor $s_{I(R)}$ by a new one; these two minors are denoted by e and f in Fig. 3. To illustrate, the two pseudoline arrangements in Fig. 2 are related by a local move that replaces s_2 by s_{13} (or vice versa).

The key observation made in [2] is that the chamber minors a, b, c, d, e, f associated with the regions surrounding the local move (cf. Fig. 3) satisfy the identity

$$ef = ac + bd. \tag{4.5}$$

Thus, f can be written as a subtraction-free expression in a, b, c, d, e and similarly e in terms of a, b, c, d, f. It is not hard to see that any flag minor s_I appears as a chamber minor in *some* pseudoline arrangement (we elaborate on this point later in this section). Consequently, by iterating the birational transformations associated with

local moves, one can get s_I as a subtraction-free rational expression in the chamber minors of any particular initial arrangement.

To complete the Proof of Theorem 3.1, i.e., to design a subtraction-free algorithm computing a Schur polynomial s_I in $O(n^3)$ steps, we need to identify an initial pseudoline arrangement (an "initial seed" in cluster algebras lingo) such that

the chamber minors for the initial seed can be computed by a subtraction-free arithmetic circuit of size $O\left(n^3\right)$, and (4.6)

for any subset $I \subset \{1, \ldots, n\}$, the initial pseudoline arrangement can be transformed into one containing s_I among its chamber minors by $O\left(n^3\right)$ local moves. (4.7)

Remark 4.1 At this point, some discussion of bit complexity is in order. Readers not interested in this issue may skip this Remark.

Each local move "flips" a triangle formed by some triple of pseudolines with labels $i < j < k$. (To illustrate, the arrangements in Fig. 2 are related by the local move labeled by the triple $(1, 2, 3)$). A sequence of say N local moves (cf. (4.7)) can be encoded by the corresponding sequence of triples

$$(i_1, j_1, k_1), \ldots, (i_N, j_N, k_N). \tag{4.8}$$

The bit complexity of our algorithm will be obtained by adding the following contributions:

- the bit complexity of computing the initial chamber minors;
- the bit complexity of generating the sequence of triples (4.8);
- the bit complexity of performing the corresponding local moves.

Concerning the last item, note that in order to execute each of the N local moves, we will need to determine which arithmetic operations to perform (there will be O(1) of them), and how to transform the data structure that encodes the pseudoline arrangement at hand, so as to reflect the changing combinatorics of the arrangement. The data structure that we suggest to use is a graph G on $\binom{n}{2} + 2n$ vertices which include the vertices v_{ij} representing pairwise intersections of pseudolines, together with the vertices v_i^{left} and v_i^{right} representing their left and right endpoints. At each vertex v in G, we store the following information:

- for each pseudoline passing through v, the vertex (if any) that immediately precedes v on that pseudoline, and also the vertex that immediately follows v;
- the set I labeling the chamber directly underneath v; and
- the corresponding chamber minor s_I.

With this in place, the local move labeled by a triple (i, j, k) is performed by identifying the (pairwise adjacent) vertices of G lying at the intersections of the pseudolines with labels i, j, k, changing the local combinatorics of the graph G in the vicinity of this triangle and performing the appropriate subtraction-free computation. For each of the

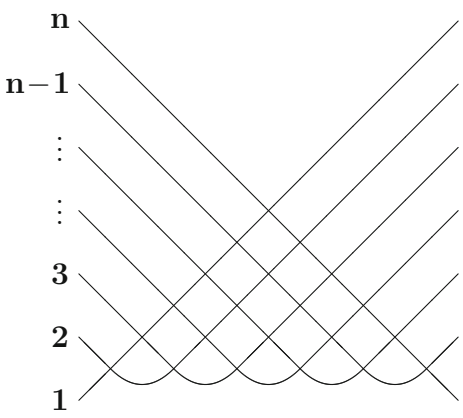


Fig. 4 The special pseudoline arrangement A°

N local moves, the number of macroscopic operations involved is $O(1)$, so the bit complexity of each move is polynomial in the size of the numbers involved (which is going to be logarithmic in n).

We proceed with the design of an efficient subtraction-free algorithm for computing a Schur polynomial s_I, following the approach outlined in (4.6)–(4.7).

Our choice of the initial arrangement is the "special" pseudoline arrangement A° shown in Fig. 4 (cf. also Fig. 2 on the left).

The special arrangement A° works well for our purposes, for the following reason. The $\frac{n(n+1)}{2} - 1$ chamber minors s_I for A° are indexed by the intervals

$$I = \{\ell, \ell+1, \ldots, \ell+k-1\} \subsetneq \{1, \ldots, n\}. \tag{4.9}$$

Moreover, such a flag minor s_I is nothing but the monomial $(x_1 \cdots x_k)^{\ell-1}$:

$$\begin{aligned} s_I &= \det\left(\frac{x_j^{i-1}}{\prod_{a<j}(x_j - x_a)}\right)_{\substack{\ell \le i \le \ell+k-1 \\ 1 \le j \le k}} \\ &= (x_1 \cdots x_k)^{\ell-1} \frac{\det\left(x_j^{i-1}\right)_{i,j=1}^{k}}{\prod_{a<j\le k}(x_j - x_a)} = (x_1 \cdots x_k)^{\ell-1}. \end{aligned} \tag{4.10}$$

(This can also be easily seen using the combinatorial definition of a Schur function in terms of Young tableaux). The collection of monomials $(x_1 \cdots x_k)^{\ell-1}$ can be computed using $O(n^2)$ multiplications, so condition (4.6) is satisfied.

To satisfy condition (4.7), at least two alternative strategies can be used, described below under the headings Plan A and Plan B.

Plan A: Combinatorial deformation The pseudocode given below in (4.11) produces a sequence of $O(n^3)$ local moves transforming the special arrangement A° into a

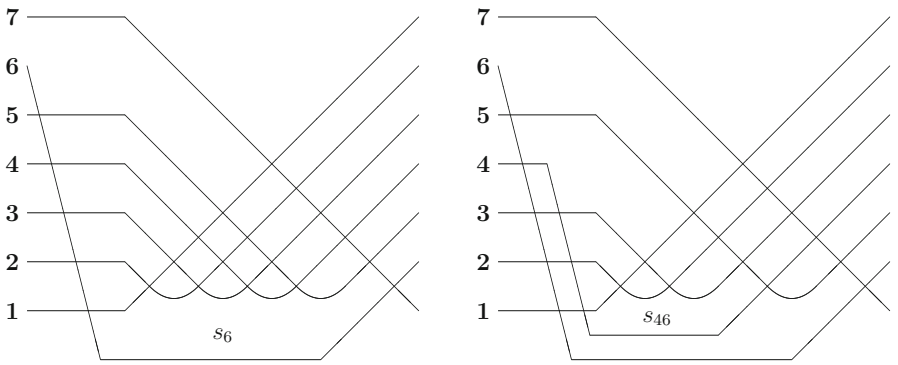


Fig. 5 Executing Algorithm (4.11) for $n = 7$. Shown are the pseudoline arrangements A_I for $I = \{6\}$ (on the *left*) and $I = \{4, 6\}$ (on the *right*). Algorithm (4.11) deforms A° (cf. Fig. 4) into $A_{\{4\}}$ and then into $A_{\{4,6\}}$

particular pseudoline arrangement A_I containing s_I as a chamber minor:

$$\begin{aligned} &\textbf{for } k := n \textbf{ downto } 3 \textbf{ do} \\ &\quad \textbf{if } k \in I \textbf{ then for } j := k-1 \textbf{ downto } 2 \textbf{ do} \\ &\qquad\qquad \textbf{for } i := j-1 \textbf{ downto } 1 \textbf{ do } \mathrm{flip}(i, j, k) \end{aligned} \tag{4.11}$$

Figure 5 illustrates the above algorithm. Its rather straightforward justification is omitted.

Plan B: Geometric deformation Here, we present an alternative solution of a more geometric flavor. The basic idea is rather simple. Fix a nonempty subset $I \subsetneq \{1, \ldots, n\}$. Suppose that we are able to build an arrangement A_I such that

- A_I consists of straight line segments L_i;
- one of the chamber minors of A_I is s_I;
- A_I is a "sufficiently generic" arrangement with these properties.

The special arrangement A° can be easily realized using straight segments. We then continuously deform A° into A_I in the following way. As the time parameter t changes from 0 to 1, each line segment $L_i(t)$ is going to change from $L_i(0) = L_i^\circ$ to $L_i(1) = L_i$ so that each endpoint of $L_i(t)$ moves at constant speed. It is possible to show that in the process of such deformation, the triangle formed by each triple of lines gets "flipped" at most once. We thus obtain a sequence of at most $\binom{n}{3}$ local moves transforming A° into A_I, as desired.

The rest of this section is devoted to filling in the gaps left over in the above outline. This can be done in many different ways; the specific implementation presented below was chosen for purely technical reasons. We assume throughout that $n \geq 3$.

First, we realize A° by the collection of straight line segments $L_1^\circ, \ldots, L_n^\circ$ where L_i° connects the points $(-1, i^2)$ and $(1, -i)$. Calculations show that the segments L_i° and L_j° intersect at a point $(u_{ij}^\circ, v_{ij}^\circ)$ with $u_{ij}^\circ = 1 - \frac{2}{i+j+1}$. Consequently, for any $i < j < k$, we have $u_{ij}^\circ < u_{ik}^\circ < u_{jk}^\circ$, implying that the arrangement's topology is as shown in Fig. 4.

We next construct the arrangement A_I. It consists of the line segments $L_1, \ldots, L_n$ such that L_i has the endpoints $(1, -i)$ and $(-1, i - 2\sigma_i\varepsilon - 2i^3\varepsilon^2)$ where

$$\varepsilon = n^{-6},$$
$$\sigma_i = \begin{cases} 0 & \text{if } i \in I; \\ -1 & \text{if } i \notin I. \end{cases}$$

Thus, L_i is a segment of the straight line given by the equation

$$y = -ix + \left(\sigma_i\varepsilon + i^3\varepsilon^2\right)(x - 1).$$

It is easy to see that the left (respectively right) endpoints of $L_1, \ldots, L_n$ are ordered bottom-up (respectively, top-down). Consequently, each pair (L_i, L_j) intersects at a point (x, y) with $-1 \le x \le 1$. Moreover, one can check that all these crossing points are distinct. Most importantly, L_i contains the point $(0, -\sigma_i\varepsilon - i^3\varepsilon^2)$, so the origin $(0, 0)$ lies above L_i if and only if $i \in I$; thus, the corresponding chamber minor is s_I.

Let us now examine the deformation of A° into A_I that we described above. As t varies from 0 to 1, the right endpoint of the ith line segment $L_i(t)$ remains fixed at $(1, -i)$, while the left endpoint moves at constant speed from its initial location at $(-1, i^2)$ to the corresponding location for A_I. Specifically, the left endpoint of $L_i(t)$ is $(-1, b_i(t))$ where

$$b_i(t) = i^2 - t\left(2\sigma_i\varepsilon + 2i^3\varepsilon^2 - i + i^2\right). \tag{4.12}$$

The ordering of the endpoints remains intact: $b_1(t) < \cdots < b_n(t)$ for $0 \le t \le 1$. Thus, the intervals $L_i(t)$ form a (pseudo)line arrangement unless some three of them are concurrent.

Lemma 4.2 *At any time instant $0 \le t \le 1$, no four intervals $L_i(t)$ have a common point.*

Proof Let t be such that distinct segments $L_i(t)$, $L_j(t)$, and $L_k(t)$ have a common point. Then, we have the identity

$$(b_i(t) - b_j(t))(i - j)^{-1} - (b_i(t) - b_k(t))(i - k)^{-1} = 0. \tag{4.13}$$

Substituting (4.12) into (4.13) and dividing by $j - k$, we obtain

$$1 - t + 2\varepsilon t\frac{\sigma_i(k - j) + \sigma_j(i - k) + \sigma_k(j - i)}{(i - j)(i - k)(j - k)} - 2\varepsilon^2 t(i + j + k) = 0. \tag{4.14}$$

The (unique) time instant $t = t_{ijk}$ at which $L_i(t)$, $L_j(t)$, and $L_k(t)$ are concurrent can be found from the linear equation (4.14). (If the solution does not satisfy $t_{ijk} \in [0, 1]$, then such a time instant does not exist.)

Now, suppose that $j' \notin \{i, j, k\}$ is such that $L_i(t)$, $L_{j'}(t)$, and $L_k(t)$ are concurrent at the same moment $t = t_{ijk}$. Then, (4.14) holds with j replaced by j'. Subtracting one equation from the other and dividing by $2\varepsilon t$, we obtain:

$$\frac{\sigma_i(k-j)+\sigma_j(i-k)+\sigma_k(j-i)}{(i-j)(i-k)(j-k)} - \frac{\sigma_i(k-j')+\sigma_{j'}(i-k)+\sigma_k(j'-i)}{(i-j')(i-k)(j'-k)} = \varepsilon(j-j'). \tag{4.15}$$

This yields the desired contradiction. Indeed, the right-hand side of (4.15) is nonzero, and less than n^{-5} in absolute value, whereas the left-hand side, if nonzero, is a rational number with denominator at most n^5. □

In view of Lemma 4.2, at each time instant $t = t_{ijk} \in [0, 1]$ satisfying equation (4.14) for some triple of distinct indices $i, j, k \in \{1, \ldots, n\}$, our pseudoline arrangement undergoes (potentially several, commuting with each other) local moves associated with the corresponding triple intersections of line segments $L_i(t), L_j(t), L_k(t)$.

Our algorithm computes the numbers t_{ijk} via (4.14), selects those satisfying $0 \le t_{ijk} \le 1$, and orders them in a non-decreasing order. This yields a sequence of $O(n^3)$ local moves transforming A° into A_I. To estimate the bit complexity, we refer to Remark 4.1 and note that the bit size of t_{ijk} is bounded by $O(\log n)$. The algorithm invokes a sorting algorithm [1] to order $O(n^3)$ numbers t_{ijk}, so its bit complexity is bounded by $O(n^3 \cdot \log^2 n)$.

Remark 4.3 Our algorithm demonstrates that the positivity of the coefficients of a Schur polynomial (as defined by the "bialternant formula" (4.4)) can be viewed as an instance of positivity of Laurent expansions of cluster variables, a general property that conjecturally holds in any cluster algebra, see [11, p. 499].

5 Double and Supersymmetric Schur Functions

In this section, we present efficient subtraction-free algorithms for computing double and supersymmetric Schur polynomials. These polynomials play important role in representation theory and other areas of mathematics, see, e.g., [12,20,22] and references therein. Our notational conventions are close to those in [20, 6th Variation]; the latter conventions differ from some other literature including [22].

5.1 Double Schur Functions

Let $y_1, y_2, \ldots$ be a sequence of formal variables. Double Schur functions $s_\lambda(x_1, \ldots, x_k|y)$ are generalizations of ordinary Schur functions $s_\lambda(x_1, \ldots, x_k)$ which depend on additional parameters y_i. The definition given below is a direct generalization of the definition of $s_\lambda(x_1, \ldots, x_k)$ given in Sect. 4.

Let $Z = (Z_{ij})_{i,j=1}^{n}$ be the $n \times n$ matrix defined by

$$Z_{ij} = \frac{\prod_{1 \le b < i}(x_j + y_b)}{\prod_{1 \le a < j}(x_j - x_a)}, \tag{5.1}$$

cf. (4.1). Thus

$$Z = (Z_{ij}) = \begin{pmatrix} 1 & \frac{1}{x_2 - x_1} & \frac{1}{(x_3 - x_1)(x_3 - x_2)} & \cdots \\ x_1 + y_1 & \frac{x_2 + y_1}{x_2 - x_1} & \frac{x_3 + y_1}{(x_3 - x_1)(x_3 - x_2)} & \cdots \\ (x_1 + y_1)(x_1 + y_2) & \frac{(x_2 + y_1)(x_2 + y_2)}{x_2 - x_1} & \frac{(x_3 + y_1)(x_3 + y_2)}{(x_3 - x_1)(x_3 - x_2)} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}.$$

For $I \subset \{1, \ldots, n\}$ of cardinality k, we set (cf. (4.2))

$$s_I(x_1, \ldots, x_k | y) = \det(Z_{ij})_{i \in I, j \le k}. \tag{5.2}$$

As before, $s_I(x_1, \ldots, x_k | y)$ is a symmetric polynomial in $x_1, \ldots, x_k$.

Now, let $\lambda = (\lambda_1, \ldots, \lambda_k)$ be a partition with at most k parts satisfying $\lambda_1 + k \le n$. The *double Schur polynomial* $s_\lambda(x_1, \ldots, x_k | y)$ is the polynomial in the variables $x_1, \ldots, x_k$ and $y_1, \ldots, y_{k+\lambda_1-1}$ defined by

$$s_\lambda(x_1, \ldots, x_k | y) = s_{I(\lambda)}(x_1, \ldots, x_k | y) = \det(Z_{ij})_{i \in I(\lambda), j \le k}, \tag{5.3}$$

where $I(\lambda)$ is given by (4.3); cf. (4.4). To recover the ordinary Schur function, one needs to specialize the y variables to 0.

Example 5.1 Consider $\lambda = (2, 1)$ with $k = 2$. Then $I(\lambda) = \{2, 4\}$, and (5.3) becomes

$$\begin{aligned} s_{(2,1)}(x_1, x_2 | y) &= \frac{(x_1 + y_1)(x_2 + y_1)}{x_2 - x_1} \det \begin{pmatrix} 1 & 1 \\ (x_1 + y_2)(x_1 + y_3) & (x_2 + y_2)(x_2 + y_3) \end{pmatrix} \\ &= (x_1 + y_1)(x_2 + y_1)(x_1 + x_2 + y_2 + y_3). \end{aligned} \tag{5.4}$$

In the special case, when $I = \{\ell, \ell + 1, \ldots, \ell + k - 1\}$ is an interval (cf. (4.9)), it is straightforward to verify that

$$s_I(x_1, \ldots, x_k | y) = \det\big(Z_{ij}\big)_{\substack{\ell \le i \le \ell + k - 1 \\ 1 \le j \le k}} = \prod_{1 \le j \le k} \prod_{1 \le b < \ell} (x_j + y_b), \tag{5.5}$$

generalizing (4.10).

The algorithm(s) presented in Sect. 4 can now be adapted almost verbatim to the case of double Schur functions. Indeed, the latter are nothing but the flag minors of the matrix Z; as such, they can be computed in an efficient and subtraction-free way,

using the same cluster transformations as before, from the chamber minors associated with the special pseudoline arrangement A°. The only difference is in the formulas for those special minors: Here, we use (5.5) instead of (4.10).

5.2 Supersymmetric Schur Functions

Among many equivalent definitions of supersymmetric Schur functions (or super-Schur functions for short), we choose the one most convenient for our purposes, due to Goulden–Greene [12] and Macdonald [20]. We assume the reader's familiarity with the concepts of a *Young diagram* and a *semistandard Young tableau* (of some *shape* λ); see, e.g., [21,31] for precise definitions.

We start with a version with an infinite number of variables. Let $x_1, x_2, \ldots$ and $y_1, y_2, \ldots$ be two sequences of indeterminates. The *super-Schur function* $s_\lambda(x_1, x_2, \ldots; y_1, y_2, \ldots)$ is a formal power series defined by

$$s_\lambda(x_1, x_2, \ldots; y_1, y_2, \ldots) = \sum_{|T|=\lambda} \prod_{s\in\lambda} (x_{T(s)} + y_{T(s)+C(s)}) \tag{5.6}$$

where

- the sum is over all semistandard tableaux T of shape λ with positive integer entries,
- the product is over all boxes s in the Young diagram of λ,
- $T(s)$ denotes the entry of T appearing in the box s, and
- $C(s) = j - i$ where i and j are the row and column that s is in, respectively.

We note that $T(s)+C(s)$ is always a positive integer, so the notation $y_{T(s)+C(s)}$ makes sense.

While this is not at all obvious from the above definition, $s_\lambda(x_1, x_2, \ldots; y_1, y_2, \ldots)$ is symmetric in the variables $x_1, x_2, \ldots$; it is also symmetric in $y_1, y_2, \ldots$; and is furthermore *supersymmetric* as it satisfies the cancellation rule

$$s_\lambda(x_1, x_2, \ldots; -x_1, y_2, y_3, \ldots) = s_\lambda(x_2, x_3, \ldots; y_2, y_3, \ldots).$$

We will not rely on any of these facts. We refer interested readers to aforementioned sources for proofs and further details.

In order to define the super-Schur function in *finitely* many variables, one simply specializes the unneeded variables to 0. That is, one sets

$$s_\lambda(x_1, \ldots, x_k; y_1, \ldots, y_m) = s_\lambda(x_1, x_2, \ldots; y_1, y_2, \ldots)\big|_{x_{k+1}=x_{k+2}=\cdots=y_{m+1}=y_{m+2}=\cdots=0}. \tag{5.7}$$

Note that the restriction of the set of x variables to $x_1, \ldots, x_k$ cannot be achieved simply by requiring the tableaux T in (5.6) to have entries in $\{1, \ldots, k\}$. A tableau with an entry $T(s) > k$ may in fact contribute to the (specialized) super-Schur polynomial: Even though $x_{T(s)}$ vanishes under the specialization, $y_{T(s)+C(s)}$ does not have to. See Example 5.2 below.

Example 5.2 (cf. Example 5.1) Let $\lambda = (2, 1)$, $k = m = 2$. The relevant tableaux T (i.e., the ones contributing to the specialization $x_3 = x_4 = \cdots = y_3 = y_4 = \cdots = 0$) are as follows:

$$\begin{array}{llllll} 1\,1 & 1\,2 & 1\,1 & 1\,2 & 2\,2 \\ 2 & 2 & 3 & 3 & 3 \end{array}$$

Then formulas (5.6) and (5.7) give

$$\begin{aligned} s_{(2,1)}(x_1, x_2; y_1, y_2) &= (x_1 + y_1)(x_2 + y_1)(x_1 + y_2) + (x_1 + y_1)(x_2 + y_1)x_2 \\ &\quad + (x_1 + y_1)y_2(x_1 + y_2) + (x_1 + y_1)y_2x_2 + (x_2 + y_2)y_2x_2 \\ &= x_1x_2(x_1 + x_2) + (x_1 + x_2)^2(y_1 + y_2) \\ &\quad + (x_1 + x_2)(y_1 + y_2)^2 + y_1y_2(y_1 + y_2). \end{aligned}$$

Specializing further at $y_2 = 0$, we obtain

$$s_{(2,1)}(x_1, x_2; y_1) = (x_1 + x_2)(x_1 + y_1)(x_2 + y_1). \tag{5.8}$$

The close relationship between super-Schur functions and double Schur functions was already exhibited in [12,20]. For our purposes, we will need the following version of those classical results.

We denote by $\ell(\lambda)$ the *length* of a partition λ, i.e., the number of its nonzero parts λ_i.

Proposition 5.3 *Assume that $m + \ell(\lambda) \le k + 1$. Then,*

$$s_\lambda(x_1, \ldots, x_k; y_1, \ldots, y_m) = s_\lambda(x_1, \ldots, x_k|y)\big|_{y_{m+1}=y_{m+2}=\cdots=0}. \tag{5.9}$$

To illustrate, let $\lambda = (2, 1)$, $k = 2, m = 1$. Then, the left-hand side of (5.9) is given by (5.8), which matches (5.4) specialized at $y_2 = y_3 = 0$.

The condition $m+\ell(\lambda) \le k+1$ in Proposition 5.3 cannot be dropped: For example, (5.9) is false for $\lambda = (2, 1)$ and $k = m = 2$ (the right-hand side is not even symmetric in y_1 and y_2).

Proof of Proposition 5.3 First, it has been established in [20, (6.16)] that

$$s_\lambda(x_1, \ldots, x_k|y) = \sum_{|T|=\lambda} \prod_{s\in\lambda} (x_{T(s)} + y_{T(s)+C(s)}),$$

the sum over all semistandard tableaux T with entries in $\{1, \ldots, k\}$. Second, in the formula (5.6), a tableau T with an entry $T(s) > k$ does not contribute to the specialization (5.7) since $T(s) + C(s) \ge k + 1 + 1 - \ell(\lambda) \ge m + 1$ and consequently, $x_{T(s)} + y_{T(s)+C(s)} = 0$. Hence, both sides of (5.9) are given by the same combinatorial formulae. □

Theorem 5.4 *The super-Schur polynomial $s_\lambda(x_1, \ldots, x_k; y_1, \ldots, y_m)$ can be computed by a subtraction-free arithmetic circuit of size $O((k + m)^3)$, assuming that $k \ge \lambda_1 + \ell(\lambda) - 2$.*

Proof Denote $k^* = m + \ell(\lambda) - 1$. If $k \geq k^*$, then (5.9) holds, and we can compute $s_\lambda(x_1, \ldots, x_k; y_1, \ldots, y_m)$ using the subtraction-free algorithm for a double Schur function, in time $O((k + \lambda_1)^3)$.

From now on, we assume that $k \leq k^* - 1$. We can still use (5.9) with k replaced by k^*, and then specialize the extra variables to 0:

$$s_\lambda(x_1, \ldots, x_k; y_1, \ldots, y_m) = s_\lambda(x_1, \ldots, x_{k^*}|y)\Big|_{\substack{y_{m+1}=y_{m+2}=\cdots=0,\\ x_{k+1}=\cdots=x_{k^*}=0}}. \tag{5.10}$$

The plan is to compute the right-hand side using the algorithm described above for the double Schur functions, with some of the x and y variables specialized to 0:

$$y_{m+1} = y_{m+2} = \cdots = 0, \quad x_{k+1} = \cdots = x_{k^*} = 0. \tag{5.11}$$

In order for this version of the algorithm to work, we need to make sure that the initial flag minors (5.5)—and consequently all chamber minors computed by the algorithm—do not vanish under (5.11). Note that we do not have to worry about the vanishing of denominators in (5.1) since the algorithm does not rely on the latter formula (The specialization as such is always defined since $s_\lambda(x_1, \ldots, x_{k^*}|y)$ is a polynomial).

The algorithm that computes $s_\lambda(x_1, \ldots, x_{k^*}|y)$ works with (specialized) flag minors of a square matrix of size

$$n^* = k^* + \lambda_1 = m + \ell(\lambda) - 1 + \lambda_1.$$

In the case of an initial flag minor, we have the formula

$$s_{[\ell,\ell+s-1]}(x_1, \ldots, x_s|y) = \prod_{1\leq j\leq s} \prod_{1\leq b<\ell-1} (x_j + y_b) \tag{5.12}$$

(cf. (5.5)); here, $\ell + s - 1 \leq n^*$, the size of the matrix. We see that such an initial minor vanishes (identically) under the specialization (5.11) if and only if the factor $x_s + y_{\ell-1}$ vanishes, or equivalently $s \geq k + 1$ and $\ell - 1 \geq m + 1$. This, however, cannot happen since it would imply that

$$m + k + 2 \leq \ell + s - 1 \leq n^* = m + \ell(\lambda) - 1 + \lambda_1$$

which contradicts the condition $k \geq \lambda_1 + \ell(\lambda) - 2$ in the theorem. □

We expect the condition $k \geq \lambda_1 + \ell(\lambda) - 2$ in Theorem 5.4 to be unnecessary. Note that one could artificially increase the number of x variables to satisfy this condition, then specialize the extra variables to 0. Such a specialization, however, is not included among the operations allowed in arithmetic circuits.

6 Skew Schur Functions

In this section, we use the Jacobi–Trudi identity to reduce the problem of subtraction-free computation of a skew Schur function to the analogous problem for the ordinary Schur functions. This enables us to deduce Theorem 3.5 from Theorem 3.1.

In accordance with usual conventions [21,31], we denote by $h_m(x_1, \ldots, x_k)$ the complete homogeneous symmetric polynomial of degree m. For $m < 0$, one has $h_m = 0$ by definition.

Let $\lambda = (\lambda_1, \ldots, \lambda_k)$ and $\nu = (\nu_1, \ldots, \nu_k)$ be partitions with at most k parts. The *skew Schur function* $s_{\lambda/\nu}(x_1, \ldots, x_k)$ can be defined by the *Jacobi–Trudi formula*

$$s_{\lambda/\nu}(x_1, \ldots, x_k) = \det(h_{\lambda_i - \nu_j - i + j}(x_1, \ldots, x_k)). \tag{6.1}$$

The polynomial $s_{\lambda/\nu}(x_1, \ldots, x_k)$ is nonzero if and only if $\nu_i \leq \lambda_i$ for all i; the latter condition is abbreviated by $\nu \subset \lambda$.

Formula (6.1) can be rephrased as saying that $s_{\lambda/\nu}$ is the $k \times k$ minor of the infinite Toeplitz matrix (h_{i-j}) that has row set $I(\lambda)$ (see (4.3)) and column set $I(\nu)$.

Let $n > k$. We fix the partition ν, and let λ vary over all partitions satisfying $I(\lambda) \subset \{1, \ldots, n\}$, or equivalently $k + \lambda_1 \leq n$. Let us denote by H_ν the $n \times k$ matrix

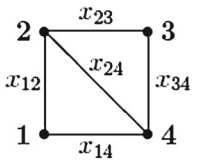
$$H_\nu = (h_{i-j}(x_1, \ldots, x_k))_{\substack{1 \leq i \leq n \\ j \in I(\nu)}}.$$

The maximal (i.e., $k \times k$) minors of H_ν are the (possibly vanishing) skew Schur polynomials $s_{\lambda/\nu}(x_1, \ldots, x_k)$. More generally, a $p \times p$ flag minor of H_ν is a skew Schur polynomial of the form $s_{\lambda/\nu(p)}$ where λ is a partition with at most p parts satisfying $p + \lambda_1 \leq n$, and $\nu(p) = (\nu_{k-p+1}, \ldots, \nu_k)$ denotes the partition formed by p smallest (possibly zero) parts of ν. Such a flag minor does not vanish if and only if $\nu(p) \subset \lambda$.

Our algorithm computes a skew Schur polynomial $s_{\lambda/\nu}(x_1, \ldots, x_k)$ (equivalently, a maximal minor of $H(\nu)$) using the same approach as before: We first compute the initial flag minors corresponding to intervals (4.9), then proceed via recursive cluster transformations.

The problem of calculating the interval flag minors of H_ν (in an efficient and subtraction-free way) turns out to be equivalent to the (already solved) problem of computing ordinary Schur polynomials. This is because $I(\lambda)$ is an interval if and only if λ has *rectangular shape*, i.e., all its nonzero parts are equal to each other. For such a partition, the nonzero skew Schur polynomial $s_{\lambda/\nu}$ is well known to coincide with an ordinary Schur polynomial s_θ where θ is the partition formed by the differences $\lambda_i - \nu_i$.

We then proceed, as before, with a recursive computation utilizing cluster transformations. However, substantial adjustments have to be made due to the fact that many flag minors of H_ν vanish. (Also, H_ν is not a square matrix, but this issue is less important.) Our recipe is as follows. Suppose that we need to perform a step of our algorithm that involves, in the notation of Fig. 3, expressing f in terms of a, b, c, d, e (It is easy to see that we never have to move in the opposite direction, i.e., from a, b, c, d, f to e,



$$f_G = x_{12}x_{14}x_{23} + x_{12}x_{14}x_{34} + x_{12}x_{23}x_{24} + x_{12}x_{23}x_{34} + x_{12}x_{24}x_{34} + x_{14}x_{23}x_{24} + x_{14}x_{23}x_{34} + x_{14}x_{24}x_{34}$$

Fig. 6 A weighted graph G and the spanning tree generating function f_G

while moving away from the special arrangement A° using the algorithms described above). If $e \neq 0$ (and we shall know beforehand whether this is the case or not), then set $f = (ac + bd)/e$ as before. If, on the other hand, $e = 0$, then set $f = 0$.

In order to justify this algorithm, we need to show that the skew Schur polynomials at hand have the property $e = 0 \Rightarrow f = 0$, in the above notation (Also, it is not hard to check in the process of computing a flag minor of size k, we never need to compute a flag minor of larger size which would not fit into H_v). This property is a rather straightforward consequence of the criterion for vanishing/nonvanishing of skew Schur functions. Let $p < q < r$ denotes the labels of the lines shown in Fig. 3, and let J denotes the set of lines passing below the shown fragment. Then, $e = s_{J\cup\{q\}}$ and $f = s_{J\cup\{p.r\}}$. Since $p < q$, the vanishing of e implies the vanishing of $s_{J\cup\{p\}}$, which in turn implies the vanishing of $f = s_{J\cup\{p.r\}}$. We omit the details.

The complexity of the algorithm is dominated by the initialization stage, which involves computing $O(n^2)$ ordinary Schur polynomials; each of them takes $O(n^3)$ operations to compute. The bit complexity is accordingly $O(n^5 \log^2 n)$.

7 Generating Functions for Spanning Trees

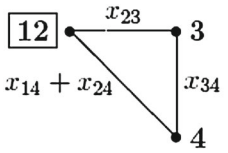
In this section, we present a polynomial subtraction-free algorithm for computing the generating function for *spanning trees* in a graph with weighted edges (a *network*). While this algorithm is going to be improved upon in Sect. 8, we decided to include it because of its simplicity, in order to highlight the connection to the theory of electric networks (equivalently, discrete potential theory). An impatient reader can go straight to Sect. 8.

Let G be an undirected connected graph with vertex set V and edge set E. We associate a variable x_e to each edge $e \in E$ and consider the generating function f_G (a polynomial in the variables x_e) defined by

$$f_G = \sum_T x^T$$

where the summation is over all spanning trees T for G, and x^T denotes the product of the variables x_e over all edges e in T. An example is given in Fig. 6.

Remark 7.1 Without loss of generality, we may restrict ourselves to the case when the graph G is *simple*, that is, G has neither loops (i.e., edges with coinciding endpoints) nor multiple edges. Loops cannot contribute to a spanning tree, so we can throw them away without altering f_G. Furthermore, if say vertices v and w are connected by several



$$f_{G(1,2)} = (x_{14} + x_{24})\, x_{23} + (x_{14} + x_{24})\, x_{34} + x_{23} x_{34}$$

Fig. 7 The graph $G(1, 2)$ for the graph G in Fig. 6

edges $e_1, \ldots, e_\ell$, then we can replace them by a single edge of weight $x_{e_1} + \cdots + x_{e_\ell}$ without changing the generating function f_G.

Recall that the number of spanning trees in a complete graph on n vertices is equal to n^{n-2}, so the monomial expansion of f_G may have a superexponential number of terms. On the other hand, there is a well-known determinantal formula for f_G, due to Kirchhoff [17] (see, e.g., [4, Theorem II.12]), known as the (weighted) Matrix-Tree Theorem. This formula provides a way to compute f_G in polynomial time—but the calculation involves subtraction. Is there a way to efficiently compute f_G using only addition, multiplication, and division? Just like in the case of Schur functions, the answer turns out to be *yes*.

Theorem 7.2 *In a weighted simple graph G on n vertices, the spanning tree generating function f_G can be computed by a subtraction-free arithmetic circuit of size $O(n^4)$.*

This result is improved to $O(n^3)$ in Sect. 8.

The rest of this section is devoted to the Proof of Theorem 7.2, i.e., the description of an algorithm that computes f_G using $O(n^4)$ additions, multiplications, and divisions. The algorithm utilizes well-known techniques from the theory of electric networks (more precisely, circuits made of ideal resistors). In order to apply these techniques to the problem at hand, we interpret each edge weight x_e as the electrical *conductance* of e, i.e., the inverse of the resistance of e. We note that the rule, discussed in Remark 7.1, for combining parallel edges into a single edge is compatible with this interpretation.

Definition 7.3 (*Gluing two vertices*) Let v and v' be distinct vertices in a weighted simple graph G as above. We denote by $G(v, v')$ the weighted simple graph obtained from G by

(i) gluing together the vertices v and v' into a single vertex which we call $\boxed{vv'}$, then
(ii) removing the loop at $\boxed{vv'}$ (if any), and then
(iii) for each vertex u connected in G to both v and v', say by edges e and e', replacing e and e' by a single edge of conductance $x_e + x_{e'}$ between u and $\boxed{vv'}$.

In view of Remark 7.1, steps (ii) and (iii) do not change the spanning tree generating function of the graph at hand. An example is shown in Fig. 7.

Lemma 7.4 (Kirchhoff's effective conductance formula [17]; see, e.g., [36, Section 2]) *Let G be a weighted connected simple graph whose edge weights are interpreted as electrical conductances. The effective conductance between vertices v and v' of G is given by*

$$\operatorname{effcond}_G\left(v, v'\right) = \frac{f_G}{f_{G(v,v')}}.$$

To illustrate, the effective conductance between vertices 1 and 2 in the graph shown in Fig. 6 is equal to $\frac{f_G}{f_{G(1,2)}}$, where f_G and $f_{G(1,2)}$ are given in Figs. 6 and 7, respectively. This matches the formula

$$\operatorname{effcond}_G(1,2) = x_{12} + \cfrac{1}{\frac{1}{x_{14}} + \cfrac{1}{x_{24} + \cfrac{1}{\frac{1}{x_{23}} + \frac{1}{x_{34}}}}}$$

that can be obtained using the *series-parallel* property of this particular graph G.

Definition 7.5 Let G be a weighted connected simple graph on the vertex set $\{1, \ldots, n\}$. Define the graphs $G_1, \ldots, G_n$ recursively by $G_1 = G$ and

$$G_{i+1} = G_i\left(\boxed{1 \cdots i}, i+1\right)$$

where $\boxed{1 \cdots i}$ denotes the vertex obtained by gluing together the original vertices $1, \ldots, i$. In other words, G_i is obtained from G by collapsing the vertices $1, \ldots, i$ into a single vertex, removing the loops, and combining multiple edges into single ones while adding their respective weights, cf. Remark 7.1.

For example, if G is the graph in Fig. 6, then $G_1 = G$; G_2 is the graph shown in Fig. 7; G_3 is a two-vertex graph with a single edge of weight $x_{14} + x_{24} + x_{34}$; and G_4 (and more generally G_n) is a single-vertex graph with no edges (so $f_{G_n} = 1$).

The following formula is immediate from Lemma 7.4, via telescoping.

Corollary 7.6 *Let G be a weighted connected simple graph on the vertex set $\{1, \ldots, n\}$. Then*

$$f_G = \prod_{i=1}^{n-1} \operatorname{effcond}_{G_i}(i, i+1).$$

Corollary 7.6 reduces the computation of the generating function f_G to the problem of computing effective conductances. The latter can be done, both efficiently and in a subtraction-free way, using the machinery of *star–mesh transformations* developed by electrical engineers, see, e.g., [6, Corollary 4.21]. The technique goes back at least 100 years, cf. the historical discussion in [26].

Lemma 7.7 (Star–mesh transformation) *Let v be a vertex in a weighted simple graph G (viewed as an electric network with the corresponding conductances). Let $e_1, \ldots, e_k$ be the full list of edges incident to v; assume that they connect v to distinct vertices $v_1, \ldots, v_k$, respectively. Transform G into a new weighted graph G' defined as follows:*

- *remove vertex v and the edges $e_1, \ldots, e_k$ incident to it;*
- *for all $1 \le i < j \le k$, introduce a new edge e_{ij} connecting v_i and v_j, and assign*

$$x_{e_{ij}} \stackrel{\text{def}}{=} x_{e_i} x_{e_j} \sum_{\ell=1}^{k} \frac{1}{x_{e_\ell}} \tag{7.1}$$

  as its weight (=conductance);
- *in the resulting graph, combine parallel edges into single ones, as in Remark 7.1.*

Then the weighted graphs G and G' have the same effective conductances. More precisely, for any pair of vertices a, b different from v, we have

$$\operatorname{effcond}_G(a, b) = \operatorname{effcond}_{G'}(a, b).$$

Lemma 7.7 provides an efficiently way to compute an effective conductance between two given vertices a and b in a graph G, by iterating the star–mesh transformations (7.1) for all vertices $v \notin \{a, b\}$, one by one. Since these transformations are subtraction-free and require $O(n^2)$ arithmetic operations each, we arrive at the following result.

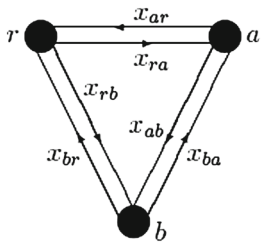
Corollary 7.8 *An effective conductance between two given vertices in an n-vertex weighted simple graph G can be computed by a subtraction-free arithmetic circuit of size $O(n^3)$.*

Combining Corollaries 7.6 and 7.8, we obtain a Proof of Theorem 7.2. The algorithm computes the effective conductances $\operatorname{effcond}_{G_i}(i, i+1)$ for $i = 1, \ldots, n-1$ using star–mesh transformations, then multiplies them to get the generating function f_G.

8 Directed Spanning Trees

In this section, we treat the directed version of the problem considered in Sect. 7, designing a polynomial subtraction-free algorithm that computes the generating function for directed spanning trees in a directed graph with weighted edges.

Similarly, to the unoriented case, our approach makes use of the appropriate version of star–mesh transformations. As before, they are local modifications of the network which transform the weights by means of certain subtraction-free formulas. There is also a difference: Unlike in Sect. 7, we apply these transformations directly to the computation of the generating functions of interest—rather than to "effective conductances" from which those generating functions can be recovered via telescoping. Adapting the latter technique to the directed case would require a thorough review of W. Tutte's theory of "unsymmetrical electricity" [33, Sections VI.4–VI.5] [34, Section 4]. This elementary but somewhat obscure theory goes back to the 1940s, see references in *loc. cit.*, and is closely related to Tutte's directed version of the Matrix-Tree Theorem [33, Theorem 6.27].



$$\varphi_G = x_{ar}x_{br} + x_{ar}x_{ba} + x_{br}x_{ab}$$

Fig. 8 The generating function φ_G for the directed spanning trees in G

Remark 8.1 The approach used in this section can be applied in the undirected case as well, bypassing the use of electric networks (cf. Sect. 7). Also, one can reduce the undirected case to the directed one by replacing each edge $a \overset{e}{—} b$ in an ordinary weighted graph by two oriented edges $a \to b$ and $b \to a$ each having the weight x_e of the original edge.

In this section, G is a *directed graph* with vertex set V and edge set E and with a fixed vertex $r \in V$ called the *root*. A *directed spanning tree* T in G (sometimes called an *in-tree*, an *arborescence*, or a *branching*) is a subgraph of G that spans all vertices in V and includes a subset of edges such that for any $v \in V$, there is a unique path in T that begins at v and ends at r. Equivalently, T is a spanning tree of G in which all edges are oriented toward r.

We assume that G has at least one such tree, or equivalently that there is a path from any vertex $v \in V$ to the root r.

We associate a variable x_e to each (directed) edge $e \in E$ and define the generating function φ_G by

$$\varphi_G = \sum_T x^T$$

where the summation is over all directed spanning trees T for G (rooted at r). As before, x^T denotes the product of the variables x_e over all edges e in T. Figure 8 shows the generating function φ_G for the *complete directed graph* on three vertices.

Without loss of generality, we may assume that G is a *simple* directed graph, i.e., it has no loops and no multiple edges, for the same reasons as in Remark 7.1. We certainly do allow pairs of edges connecting the same pair of vertices but oriented in opposite ways.

Theorem 8.2 *In a weighted simple directed graph G on n vertices, the generating function for directed spanning trees rooted at a given vertex r can be computed by a subtraction-free arithmetic circuit of size $O(n^3)$.*

In view of Remark 8.1, the analogue of Theorem 8.2 for undirected graphs follows, improving upon Theorem 7.2 and implying Theorem 3.8.

The algorithm that establishes Theorem 8.2 relies on the following lemma.

Lemma 8.3 (Star–mesh transformation in a directed network) *Let $v \neq r$ be a vertex in a weighted directed graph G as above. Let $v_1, \ldots, v_k$ be the full list of vertices directly connected to v by an edge (either incoming, or outgoing, or both). For $i = 1, \ldots, k$, let x_i (resp., y_i) denote the weight of the edge $v_i \to v$ (resp., $v \to v_i$); in the absence of such edge, set $x_i = 0$ (resp., $y_i = 0$). Transform G into a new weighted directed graph G'' as follows:*

- *remove vertex v and all the edges incident to it;*
- *for each pair $i, j \in \{1, \ldots, k\}$, $i \neq j$, $x_i y_j \neq 0$, introduce a new edge e_{ij} directed from v_i to v_j, and set its weight to be*

$$x_{e_{ij}} \stackrel{\text{def}}{=} x_i \, y_j \, (y_1 + \cdots + y_k)^{-1}; \tag{8.1}$$

- *in the resulting graph G', combine multiple edges (if any), adding their respective weights, to obtain G''. (Thus $\varphi_{G''} = \varphi_{G'}$.)*

Then

$$\varphi_G = (y_1 + \cdots + y_k)\, \varphi_{G''}. \tag{8.2}$$

We note that $y_1 + \cdots + y_k \neq 0$ since otherwise there is no path from v to r. (If that happens, we have $\varphi_G = 0$.)

It is easy to see that Lemma 8.3 implies Theorem 8.2. The algorithm computes the generating function φ_G by iterating the star–mesh transformations described in the lemma.

Example 8.4 Consider the weighted graph in Fig. 8. Choose $v = b$. The recipe in Lemma 8.3 asks us to remove the vertex b and the four edges incident to it, introducing instead two edges connecting r and a. According to the formula (8.1), the new edge in G' pointing from a to r has weight $x_{ab}\, x_{br}\, (x_{ba} + x_{br})^{-1}$. Adding this to the weight x_{ar} of the old edge $a \to r$, we obtain the combined weight of the edge going from a to r in the two-vertex graph G''. Thus,

$$\varphi_{G''} = x_{ar} + \frac{x_{ab}\, x_{br}}{x_{ba} + x_{br}} = \frac{x_{ar}\, x_{ba} + x_{ar}\, x_{br} + x_{ab}\, x_{br}}{x_{ba} + x_{br}}.$$

Then (8.2) gives

$$\varphi_G = (x_{ba} + x_{br})\, \varphi_{G''} = x_{ar}\, x_{ba} + x_{ar}\, x_{br} + x_{ab}\, x_{br},$$

matching the result of a direct calculation in Fig. 8.

It remains to prove Lemma 8.3. The proof uses a classical result (see, e.g., [31, Theorem 5.3.4], with $k = 1$) sometimes called "the Cayley–Prüfer theorem"; it is indeed immediate from Prüfer's celebrated proof of Cayley's formula for the number of spanning trees. We state this result in a version best suited for our purposes.

Lemma 8.5 *Let H be a complete directed graph on the vertex set W, with root $r \in W$. For $v \in W$, let z_v be a formal variable. Assign to every edge $a \to b$ in H the weight $z_b\, (\sum_v z_v)^{-1}$. Then substituting these weights into φ_H gives $z_r\, (\sum_v z_v)^{-1}$.*

Example 8.6 The case when H has three vertices is shown in Fig. 8. Substituting $x_{ij} = z_j\,(z_a + z_b + z_r)^{-1}$, we get

$$\begin{aligned}\varphi_H &= x_{ar}x_{br} + x_{ar}x_{ba} + x_{br}x_{ab} = \left(z_r^2 + z_r z_a + z_r z_b\right)(z_a + z_b + z_r)^{-2}\\ &= z_r\,(z_a + z_b + z_r)^{-1}\,.\end{aligned}$$

Proof of Lemma 8.3 The proof uses standard techniques of elementary enumerative combinatorics. As Eq. (8.2) is equivalent to

$$\varphi_G = (y_1 + \cdots + y_k)\,\varphi_{G'}, \tag{8.3}$$

we will be proving the latter identity.

The edge set E of G naturally splits into two disjoint subsets. The $2k$ edges $v_i \to v$ and $v \to v_i$ form Star_v (the *star* of v). The remaining edges form the set $\mathrm{Out}_v = E \setminus \mathrm{Star}_v$. Similarly, the edge set E' of G' is a disjoint union of $\mathrm{Mesh}_v = \{e_{ij}\}$ (the *mesh* of v) and Out_v.

We shall write φ_G (resp., $\varphi_{G'}$) as a sum of terms of the form AB where A is a polynomial expression in the weights of the edges in Star_v (resp., Mesh_v) while B only involves the weights of edges in Out_v. Each factor B will be a generating function for a certain class of directed forests in Out_v. (Think of those forests as leftover chunks of a directed tree after its edges in Star_v (resp., Mesh_v) have been removed.) More specifically, the factors B in our formulas will be of the following kind. Let $\mathcal{P} = \{P_a\}$ be an (unordered) partition of the set

$$K = \{v_1, \ldots, v_k\} \cup \{r\}$$

into nonempty subsets P_a (called *blocks*) where in each block P_a, one vertex a has been designated as the *root* of the block. If $r \in P_a$ (i.e., if the block contains the root of G), then we require that $a = r$; moreover, P_r must contain at least one of the elements $v_1, \ldots, v_k$. We denote by $B(\mathcal{P})$ the generating function for the directed forests F which span the vertex set $V \setminus \{v\}$ and have the property that the vertices in K are distributed among the connected components of F as prescribed by $\mathcal{P}$. More precisely, each connected component C of F is a directed tree whose vertex set includes all vertices from some block P_a of $\mathcal{P}$ (and no vertices from other blocks), with a serving as the root of C. (In particular, C contains at least one of the vertices $v_1, \ldots, v_k$.) The weight of F is the product of the weights of its edges.

To complete the proof, we are going to write formulas of the form

$$\varphi_G = \sum_{\mathcal{P}} A(\mathcal{P})B(\mathcal{P}) \tag{8.4}$$

$$\varphi_{G'} = \sum_{\mathcal{P}} A'(\mathcal{P})B(\mathcal{P}) \tag{8.5}$$

(sums over rooted set partitions $\mathcal{P}$ as above) and demonstrate that for any $\mathcal{P}$, we have

$$A(\mathcal{P}) = (y_1 + \cdots + y_k)\, A'(\mathcal{P}). \tag{8.6}$$

Let $\mathcal{P} = \{P_a\}$ be a partition of K as above. For each block P_a, denote $Y_a = \sum_{v_i \in P_a} y_i$, the sum of the weights of the edges $v \to v_i$ entering the block P_a. The edges of each directed tree in G contributing to φ_G split into those contained in Star_v and those belonging to Out_v. The latter edges form a directed spanning forest in $V \setminus \{v\}$ whose connected components, with their roots identified, correspond to a partition $\mathcal{P}$ as above. Direct inspection shows that combining the terms in φ_G corresponding to each $\mathcal{P}$ yields the formula (8.4) with

$$A(\mathcal{P}) = Y_r \prod_{a \neq r} x_a.$$

An analogous—if less straightforward—calculation for the graph G', with Star_v replaced by Mesh_v, results in the formula (8.5) with

$$A'(\mathcal{P}) = \sum_T \prod_{P_a \to P_b} x_a\, Y_b\, (y_1 + \cdots + y_k)^{-1},$$

where the sum is over all directed trees T on the vertex set $\{P_a\}$, with root P_r (i.e., the vertices of T are the blocks of $\mathcal{P}$), and the product is over all directed edges $P_a \to P_b$ in T. We note that $\sum_{P_a} Y_a = y_1 + \cdots + y_k$. Thus, Lemma 8.5 applies, and we get

$$A'(\mathcal{P}) = Y_r\, (y_1 + \cdots + y_k)^{-1} \prod_{a \neq r} x_a,$$

implying (8.6). □

9 Subtraction-Free Complexity Versus Ordinary Complexity

In this section, we exhibit a sequence of rational functions (f_n) whose ordinary arithmetic circuit complexity is linear in n (or even $O(1)$ if one allows arbitrary constants as inputs), while their subtraction-free complexity grows exponentially in n.

Lemma 9.1 *Let F be a rational function (in one or several variables) representable as a ratio of polynomials with nonnegative coefficients. Assume that in any such representation $F = P/Q$, the (total) degree of P is greater than 2^m. Then, the subtraction-free complexity of F is greater than m.*

Proof Let $\mathcal{D}_k$ denotes the class of rational functions f which can be written in the form $f = p/q$ where both p and q have nonnegative coefficients and have degrees at most k. It is easy to see that if $f_1, f_2 \in \mathcal{D}_k$, then each of the functions $f_1 + f_2$, $f_1 f_2$, and f_1/f_2 lie in $\mathcal{D}_{2k}$. It follows that if F has subtraction-free complexity l, then

$F \in \mathcal{D}_{2^l}(x)$. On the other hand, the conditions in the lemma imply that $F \notin \mathcal{D}_{2^m}$. Hence, $l > m$. □

Lemma 9.2 *For a positive integer N, the quadratic univariate polynomial*

$$F_N(x) = (x-1)^2 + \frac{1}{N^2}$$

can be written as a subtraction-free expression. Furthermore, if $F_N(x)Q(x) = P(x)$ where $P(x)$ is a polynomial with nonnegative coefficients, then $\deg(P) > N$.

Proof By a classical theorem of Pólya [24], the fact that $F_N(x) > 0$ for any $x \geq 0$ (actually, any $x \in \mathbb{R}$) implies that we can write $F_N(x) = p(x)/(1+x)^r$ for r a sufficiently large integer, and $p(x)$ a polynomial with nonnegative coefficients. (It can be shown that $r > 9N^2$ suffices, cf. [25, p. 222].)

Let us prove the second statement. Assume that on the contrary, $\deg(P) \leq N$, and denote $P(x) = \sum_{k=0}^{N} p_k x^k$. Let $u = 1 + \sqrt{-1}/N$ and $v = 1 - \sqrt{-1}/N$ be the roots of F_N. Then,

$$u^k + v^k = 2\left(1 + \frac{1}{N^2}\right)^{k/2} \cos\left(k \cdot \tan^{-1}\left(\frac{1}{N}\right)\right).$$

If $0 \leq k \leq N$, then $0 \leq k \tan^{-1}\left(\frac{1}{N}\right) \leq \frac{k}{N} \leq 1 < \frac{\pi}{2}$, implying that $u^k + v^k > 0$. Consequently,

$$0 = F_N(u)Q(u) + F_N(v)Q(v) = P(u) + P(v) = \sum_{k=0}^{N} p_k \left(u^k + v^k\right) > 0,$$

a contradiction. □

Proposition 9.3 *The subtraction-free complexity of the univariate polynomial*

$$G_n(x) = F_{2^n}(x) = (x-1)^2 + 2^{-2^{n+1}},$$

while finite, is greater than 2^n. The ordinary arithmetic circuit complexity of $G_n(x)$ is $O(1)$ if arbitrary constants are allowed as inputs. If 1 is the only input constant allowed, the ordinary complexity of $G_n(x)$ is $O(n)$.

Proof By Lemma 9.2, the subtraction-free complexity of G_n is finite, and for any representation $G_n = P/Q$ where P and Q are polynomials with nonnegative coefficients, we have $\deg(P) > 2^{2^n}$. Now, Lemma 9.1 implies that subtraction-free complexity of G_n is greater than 2^n. Finally, the last statement of the proposition follows from the fact that 2^{2^n} can be computed by iterated squaring. □

The reader might feel uncomfortable about the fact that the polynomial $G_n(x)$ in Proposition 9.3 has a coefficient whose binary notation has exponential length. To alleviate those concerns, we present a closely related example that does not have

this drawback. In doing so, we use a modification of the well-known Lazard–Mora–Philippon trick, cf., e.g., [14].

Proposition 9.4 *Define the homogeneous polynomials* $H_n(t, x_1, \ldots, x_n)$ *by*

$$H_n(t, x_1, \ldots, x_n) = (x_1 - t)^4 + (x_1 - 2x_2)^4 + \left(x_2^2 - tx_3\right)^2 + \left(x_3^2 - tx_4\right)^2 + \cdots + \left(x_{n-1}^2 - tx_n\right)^2 + 4(x_1 - t)^2 x_n^2 + 2x_n^4.$$

Then, the subtraction-free complexity of H_n*, while finite, is greater than* 2^{n-2}*. By contrast, the ordinary arithmetic circuit complexity of* H_n *is linear in* n*.*

Proof Since $H_n(t, x_1, \ldots, x_n)$ is positive for any nonnegative (in fact, any real) vector $(t, x_1, \ldots, x_n) \neq (0, 0, \ldots, 0)$, Pólya's theorem [24] tells us that we can write

$$H_n(t, x_1, \ldots, x_n) = p(t, x_1, \ldots, x_n)/(t + x_1 + \cdots + x_n)^r,$$

for some polynomial p with nonnegative coefficients, and some positive integer r. So the subtraction-free complexity of H_n is finite.

Assume that $H_n = P/Q$ where P and Q are polynomials with nonnegative coefficients. Substituting $t = 1, x_2 = 2^{-1}, x_3 = 2^{-2}, \ldots, x_n = 2^{-2^{n-2}}$, we get:

$$\begin{aligned}\frac{P\left(1, x_1, 2^{-1}, 2^{-2}, \ldots, 2^{-2^{n-2}}\right)}{Q\left(1, x_1, 2^{-1}, 2^{-2}, \ldots, 2^{-2^{n-2}}\right)} &= H_n\left(1, x_1, 2^{-1}, 2^{-2}, \ldots, 2^{-2^{n-2}}\right)\\ &= (x_1 - 1)^4 + (x_1 - 1)^4\\ &\quad + 4(x_1 - 1) \cdot 2^{-2^{n-1}} + 2 \cdot 2^{-2^n}\\ &= 2\left(F_{2^{n-2}}(x_1)\right)^2.\end{aligned}$$

Since $P(1, x_1, 2^{-1}, 2^{-2}, \ldots, 2^{-2^{n-2}})$ is a polynomial with nonnegative coefficients, we can apply Lemma 9.2 to conclude that $\deg(P) \geq \deg_{x_1}(P) > 2^{2^{n-2}}$. Now, Lemma 9.1 implies that the subtraction-free complexity of H_n is greater than 2^{n-2}. □

Acknowledgments We thank Leslie Valiant for bringing the paper [15] to our attention.

References

1. A. Aho, J. Hopcroft, and J. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley, 1975.
2. A. Berenstein, S. Fomin, and A. Zelevinsky, Parametrizations of canonical bases and totally positive matrices, *Adv. Math.***122** (1996), 49–149.
3. P. Bürgisser, M. Clausen, and M. A. Shokrollahi, *Algebraic complexity theory*, Springer-Verlag, 1997.
4. B. Bollobás, *Modern graph theory*, Springer, 1998.
5. C. Chan, V. Drensky, A. Edelman, R. Kan, and P. Koev, On computing Schur functions and series thereof, preprint, 2008.

6. W.-K. Chen, *Graph theory and its engineering applications*, World Scientific, 1997.
7. V. I. Danilov, A. V. Karzanov, and G. A. Koshevoy, Systems of separated sets and their geometric models, *Uspehi Mat. Nauk*, **65** (2010), 67-152 (in Russian); English translation in *Russian Math. Surveys***65** (2010), no. 4, 659–740.
8. J. Demmel, M. Gu, S. Eisenstat, I. Slapničar, K. Veselić, and Z. Drmač, Computing the singular value decomposition with high relative accuracy, *Linear Algebra Appl.***299** (1999), no. 1-3, 21–80.
9. J. Demmel and P. Koev, Accurate and efficient evaluation of Schur and Jack functions, *Math. Comp.***75** (2006), no. 253, 223–239.
10. S. Fomin, Total positivity and cluster algebras, *Proceedings of the International Congress of Mathematicians. Volume II*, 125–145, Hindustan Book Agency, 2010.
11. S. Fomin and A. Zelevinsky, Cluster algebras I: Foundations, *J. Amer. Math. Soc.*, **15**, (2002), 497–529.
12. I. Goulden and C. Greene, A new tableau representation for supersymmetric Schur functions, *J. Algebra***170** (1994), 687–703.
13. D. Grigoriev, Lower bounds in algebraic complexity, *J. Soviet Math.***29** (1985), 1388–1425.
14. D. Grigoriev and N. Vorobjov, Complexity of Null- and Positivstellensatz proofs, *Ann. Pure Appl. Logic***113** (2002), 153–160.
15. M. Jerrum and M. Snir, Some exact complexity results for straight-line computations over semirings, *J. Assoc. Comput. Mach.***29** (1982), 874–897.
16. P. W. Kasteleyn, Graph theory and crystal physics, in: *Graph theory and theoretical physics*, 43–110, Academic Press, 1967.
17. G. Kirchhoff, Über die Auflösung der Gleichungen, auf welche man bei der Untersuchungen der linearen Vertheilung galvanischer Ströme geführt wird, *Ann. Phys. Chem.***72** (1847), 497–508.
18. P. Koev, Accurate computations with totally nonnegative matrices, *SIAM J. Matrix Anal. Appl.***29** (2007), 731–751.
19. G. L. Litvinov, Idempotent and tropical mathematics; complexity of algorithms and interval analysis. *Comput. Math. Appl.***65** (2013), 1483–1496.
20. I. G. Macdonald, Schur functions: theme and variations, *Séminaire Lotharingien de Combinatoire (Saint-Nabor, 1992)*, 5–39, Publ. Inst. Rech. Math. Av., 498, Univ. Louis Pasteur, Strasbourg, 1992.
21. I. G. Macdonald, *Symmetric functions and Hall polynomials*, Oxford Mathematical Monographs, 1999.
22. A. I. Molev, Comultiplication rules for the double Schur functions and Cauchy identities, *Electron. J. Combin.***16** (2009), no. 1, Research Paper 13, 44 pp.
23. H. Narayanan, On the complexity of computing Kostka numbers and Littlewood-Richardson coefficients, *J. Algebraic Combin.***24**, (2006), 347–354.
24. G. Pólya, Über positive Darstellung von Polynomen, in: *Vierteljschr. Naturforsch. Ges. Zrich***73** (1928), 141–145; see: Collected Papers, vol. 2, MIT Press, Cambridge, 1974, pp. 309–313.
25. V. Powers and B. Reznick, A new bound for Pólya's theorem with applications to polynomials positive on polyhedra, *J. Pure Appl. Algebra***164** (2001), 221–229.
26. J. Riordan, Review MR0022160 (9,166f), *Math. Reviews*, AMS, 1948.
27. G. Rote, Division-free algorithms for the determinant and the Pfaffian: algebraic and combinatorial approaches. *Computational discrete mathematics*, 119–135, Lecture Notes in Comput. Sci. **2122**, Springer, Berlin, 2001.
28. C. P. Schnorr, A lower bound on the number of additions in monotone computations, *Theor. Comput. Sci.***2**, (1976), 305–315.
29. E. Shamir and M. Snir, Lower bounds on the number of multiplications and the number of additions in monotone computations, *Technical Report RC-6757*, IBM, 1977.
30. A. Shpilka and A. Yehudayoff, Arithmetic circuits: a survey of recent results and open questions, *Found. Trends Theor. Comput. Sci.***5** (2009), no. 3–4, 207–388 (2010).
31. R. P. Stanley, *Enumerative combinatorics, vol. 2*, Cambridge University Press, 1999.
32. V. Strassen, Vermeidung von Divisionen, *J. Reine Angew. Math.***264** (1973), 184–202.
33. W. T. Tutte, *Graph theory*, Addison-Wesley, 1984.
34. W. T. Tutte, *Graph theory as I have known it*, Oxford University Press, 1998.
35. L. G. Valiant, Negation can be exponentially powerful, *Theor. Comput. Sci.***12**, (1980), 303–314.
36. D. G. Wagner, Matroid inequalities from electrical network theory, *Electron. J. Combin.***11** (2004/06), no. 2, Article 1, 17 pp.