Abstract

D'SOUZA, ERWIN. Automating the enumeration of sequences defined by digraphs. (Under the direction of Carla D. Savage.)

We consider sequences of nonnegative integers $S = (s_1, s_2, \ldots, s_n)$ defined by systems of constraints represented as weighted directed graphs in which edge (s_a, s_b) of weight w indicates constraint $s_a \geq s_b + w$. We propose a set of seven rules and a decomposition technique for obtaining multi-variable and single-variable generating functions for families of such graphs. Our method compares to existing techniques by offering an elegant and intuitive approach to obtaining generating functions and recurrences, albeit only for a subset of the partition and composition enumeration problems addressed by other techniques. The decomposition technique we propose remains relevant, nevertheless, to a wide range of applications, including several well-known ones. Moreover, our objective is to obtain recurrences for generating functions so as to assist the formulation and proof of their closed-form solutions. For integer sequences defined by directed graphs with $w \in \{0, 1\}$, we prove that our technique holds sufficient.

We describe the formulation of finite-variable generating function recurrences from multi-variable ones and provide a set of rules to determine the variables chosen. The construction tree is introduced as the tree representation of the construction of a generating function from the decomposition of a weighted directed graph. Given such a construction tree, automation of the process of building the multi-variable and finite-variable recurrences is possible, and we implement it as a computer program.

Finally, we apply our techniques and tools to a wide range of famous problems, including 2-rowed plane partitions, up-down compositions and hexagonal plane partitions, as well as some new problems, and obtain recurrences to each. We find that our methods are not only effective but also easy and simple to use.

AUTOMATING THE ENUMERATION OF SEQUENCES DEFINED BY DIGRAPHS

BY

Erwin D'Souza

A THESIS SUBMITTED TO THE GRADUATE FACULTY OF
NORTH CAROLINA STATE UNIVERSITY
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

COMPUTER SCIENCE

RALEIGH

September 2005

APPROVED BY:

CHAIR OF ADVISORY COMMITTEE

To my loving parents

Biography

Erwin D'Souza was born in Udupi, India in 1979. After a memorable schooling in the State of Bahrain, the Kingdom of Saudi Arabia and hometown Brahmavar, he obtained a Bachelor of Engineering in Computer Science degree at the Manipal Institute of Technology in 2001. Following his life-long passion for computer programming, he entered the software industry in 2002, working for Infosys Technologies Ltd., India, for a year before his academic interests took him half way across the world. He obtained his Master of Science in Computer Science degree from North Carolina State University, U.S.A., in 2005.

He currently looks forward to conquering the world at Google Inc. in the Silicon Valley.

Acknowledgments

I am enormously indebted to my research advisor, Dr. Carla Savage, for having guided me along this path that I am very pleased to have taken. The wisdom and inspiration that she imparted on me was as much of a gift to me as her constant support and encouragement. There is much that I will take away from this wonderful experience.

I am very thankful to Dr. Erich Kaltofen and Dr. Jon Doyle for their helpful reviews and insights that helped enhance this thesis. I am grateful as well as proud of their being part of my advisory committee.

I would do grave injustice not to mention Dr. Mehmet Ozturk whose support and guidance through these two unforgettable years I would have been handicapped without. I am undoubtedly lucky to have walked in the shadow of his kindness. I thank him also for being a part of my advisory committee.

For their unflagging faith in me and their never-ending support, love and prayers, I give special thanks to my parents. The suitcase full of snacks and food items played a big role in the speedy completion of this thesis.

Table of Contents

List of Figures

Chapter 1

Introduction

Integer partitions hold a special place in the vast field of Combinatorics. Despite a very straightforward definition, that of being an unordered collection of integers summing up to some integer n, they are surrounded by several interesting and not-so-trivial problems. Applications of integer partitions are widespread, ranging from the theory of symmetric functions to Statistical Physics. Euler made a breakthrough in enumerating partitions when he used *generating functions* for presenting their solutions [27], thereby introducing a reasonable way of counting them, though not quite directly. The generating function approach has gained much momentum, evolving into enumeration problems not merely of integer partitions but also of sequences of integers satisfying arbitrary systems of constraints. Recent work has involved the automation of the construction of generating functions for a given set of inequalities that produce only integer solutions (*diophantine inequalities*) and the derivation of recurrences for assisting not only in the guessing of solutions but also in the proofs of them. In this thesis, we restrict our attention to those systems of constraints defined by inequalities of the form $s_a \geq s_b + w$ where s_a and s_b belong to the sequence under consideration, $(s_1, s_2, \ldots, s_m)$, and w is an integer. We devise a simple set of rules for obtaining generating function recurrences for these restricted systems and find that our technique stays elegant and simple while still being applicable to a wide range of well-studied problems. We demonstrate also that the process is mechanical enough to be automated.

Euler's investigations in the enumeration of partitions led to easy-to-explain generating functions of the form

$$F(x) = \frac{1}{(1-x)(1-x^2)\ldots(1-x^m)}$$

wherein the coefficient of x^n gives the number of partitions of n into m parts or fewer. MacMahon introduced Partition Analysis [27] as a technique for obtaining the generating function of a sequence of integers defined by a set of diophantine inequalities. The method based itself upon the $\underset{\geq}{\Omega}$ operator

that sought to remove all invalid solutions from a generating function. An accompanying collection of twelve 'Omega rules' helped perform this task. The Partition Analysis approach lost steam, however, and was only recently reintroduced by Andrews, Paule and Riese in a succession of papers [3, 4, 6, 10, 7, 8, 9, 5, 1, 2] that solved several interesting partition enumeration problems. As part of these efforts, the `Omega` package [6, 7] was designed to automate the Omega rules so as to take advantage of computer algebra methods. Despite its being undoubtedly powerful, a limitation of the `Omega` package was its inflexibility in handling *families* of constraint systems. The Five Guidelines approach proposed by Corteel, Lee and Savage in [18] attempts to get around this disadvantage by focusing on obtaining *recurrences* for the generating functions. This technique built upon what was reasoned to be the core of the Omega rules and was successful in constructing recurrences for several well-known problems. We find, however, that the technique can be simplified to make the solving of simpler systems of constraints more intuitive.

Restricting the type of inequalities allowable to only those between two integer parts, i.e., those that could be expressed as

$$s_a \geq s_b + w \tag{1.1}$$

for some integer w, allows us to greatly simplify the procurement of generating function recurrences for systems defined on them. Moreover, these constraints can be expressed in the form of a directed graph wherein the integer parts are represented by vertices and each constraint (1.1) by an edge (s_a, s_b) of weight w. As we shall see, this representation allows us to devise a technique whose application is not merely intuitive but also, arguably, fun. Potential applications of such a technique include problems that have been well studied in the past such as, for example, 2-rowed plane partitions [28, 4].

1.1 Our contribution

We restrict our attention to those systems of constraints that can be represented as directed graphs with weighted edges, as already described. A set of six simple rules is proposed, along with a technique for applying the rules upon a directed graph in a way that allows construction of the generating function for the system it represents. The rules we propose derive from the Five Guidelines and are simplified for the restricted problems we consider. We prove the sufficiency of these rules for certain classes of directed graphs.

We then take the technique a step further by incorporating an additional rule that enables us to treat sequences of directed graphs and obtain multi-variable recurrences for their generating functions. Defining finite-variable versions for these recurrences involves deciding which variables to keep and which to discard. We show how to overcome this non-trivial stumbling-block.

Several aspects of the technique we propose are noticeably mechanical, therefore suggesting an automated alternative. We develop a Maple package that constructs generating function recurrences for given families of directed graphs.

Finally, we approach several well-known problems and some new ones with the new set of techniques and tools we have thus acquired. Famous problems we consider include 2-rowed plane partitions (with or without diagonals) [28, 4, 1], up-down sequences [14], 1-compositions [26], plane partition hexagonals (with or without diagonals) [2] and plane partition diamonds [9].

1.2 Organization

This chapter gave an brief introduction to the ideas and motivation behind this thesis. In Chapter 2, we give an overview of the basic concepts that form the theme for this thesis, including enumerative combinatorics, generating functions, formal grammars and parsing techniques. Chapter 3 details prior work conducted in the field of enumerating partitions and compositions, and forms the basis of the research presented in the rest of the thesis. In Chapter 4, we propose seven rules for decomposition of graphs and describe the constraint graph technique. We also prove sufficiency of the rules and investigate the creation of finite-variable generating function recurrences from multi-variable ones. Chapter 5 describes the structure and functioning of the program we developed to automate construction of generating functions, and also serves as a manual for future users. In Chapter 6, we use the tools and techniques developed to obtain generating functions for several well-known problems as well as a few new ones. Chapter 7 summarizes our contributions, formulates some of the open problems and concludes the thesis.

Chapter 2

Background

This chapter constitutes a broad overview of subjects relevant to this thesis, with the intention of putting the work into context. We begin in Section 2.1 with an overview of problems in combinatorics that deal with enumeration and counting. Section 2.2 introduces generating functions and emphasizes the innate power of the concept. We also demonstrate how generating functions can be tailored to solve combinatorics problems such as that of enumerating partitions and compositions. Finally, in preparation for the automation of the constraint graph technique in Chapter 5, Section 2.4 presents an overview of formal grammars and parsing techniques.

2.1 Enumeration problems

In this section, we provide an overview of those problems in combinatorics that deal with enumeration and counting. This includes an introduction to integer compositions and partitions, which will form the basis of this thesis.

Combinatorics is the field of mathematics that deals with orderings and arrangements of objects. There are three basic problems of combinatorics [31]. *Enumeration problems* seek to count or list out all possible solutions, *existential problems* pose the question of whether or not there exists a solution, and, *optimization problems* deal with finding the best one. In this section, we are interested only in enumeration problems and we provide a series of examples illustrating them. Some problems we encounter have no direct formula, thereby setting the stage for generating functions, which we explore in Section 2.2. Note that we use the term *enumeration* in this thesis to refer to the listing out of solutions, as opposed to the term *counting* which we use to refer to the determination of the number of them.

Permutations of a set

Consider the problem of enumerating and counting all orderings of a set of n distinct objects. The first object in every ordering can be selected in n ways, the second in $n-1$ ways, and so on. Since the number of ways of selecting each object is independent of the objects already picked, there are a total of $n!$ orderings.

Permutations of a multiset

Consider the ordering of a collection of n objects that contains r_1 similar objects of one type, r_2 of another, and so on up to r_k, where $r_1 + r_2 + \ldots + r_k = n$. The number of solutions is

$$\frac{n!}{r_1!r_2!\ldots r_k!}$$

and can be represented as $\binom{n}{r_1, r_2, \ldots r_k}$, called the *multinomial coefficient* [29]. This equation can be explained as the taking away of the number of permutations of objects for each type, $r_i!$, from the number of permutations of n distinct objects, $n!$.

Combinations

Consider the problem of counting the number of possible subsets of a set of n distinct objects. We see that there must be exactly 2^n of them by observing that every object is either present or absent from each subset, independent of other objects.

Now consider the same problem with the difference that every subset must contain exactly r objects. This can be equated to the problem of ordering a multiset of n labels, with r elements labelled 'Y' and $n-r$, 'N', such that 'Y' corresponds to the selection of the object at that location in the ordering and 'N' corresponds to its non-selection. This can be obtained from the multinomial coefficient of n, as

$$\binom{n}{r, n-r} = \frac{n!}{r!(n-r)!},$$

represented as $C(n,r)$.

r-Permutations

Consider the problem of finding all r-subsets of a set of n elements, where order is important. Observe that if order is irrelevant, there are $C(n,r)$ such subsets. Since each of these can be permuted in $r!$ ways, we find that the number of solutions to the problem is

$$\begin{aligned} P(n,r) &= r!C(n,r) \\ &= \frac{n!}{(n-r)!}. \end{aligned}$$

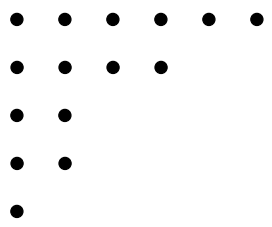


Figure 2.1: Ferrer's diagrams for a partition of 15 as (6,4,2,2,1)

Selection with replacement

The number of ways of making ordered selections of r objects from n wherein every object may be selected more than once is n^r, since every element of the ordering can be independently selected in n different ways. If order is irrelevant, however, the number of ways of doing this is $C(r+n-1, r)$ [29].

Integer compositions

Define a *composition* of an integer n into m parts as a sequence of positive integers $(s_1, s_2, \ldots, s_m)$ where $s_1 + s_2 + \ldots + s_m = n$. For example, the compositions of 5 into 3 parts are $(1,1,3)$, $(1,2,2)$, $(1,3,1)$, $(2,1,2)$, $(2,2,1)$ and $(3,1,1)$. The number of compositions of n into m parts is $C(n-1, m-1)$, obtained as follows. Represent the integer n as a series of n dots. These dots create $n-1$ spaces in between them. Filling $m-1$ of these $n-1$ spaces with "walls" splits the series of n dots into m smaller series. The values for $s_1, s_2, \ldots s_m$ can be read off from these m series, thereby giving a valid composition of n. Observing that there are exactly $C(n-1, m-1)$ ways of filling $n-1$ spaces with $m-1$ walls leads directly to the result.

The number of compositions of n into *any* number of positive parts can be shown to be 2^{n-1}, using a similar approach as above.

Partitions

Define a *partition* of an integer n into m parts as a sequence of positive integers $(s_1, s_2, \ldots, s_m)$ where $s_1 + s_2 + \ldots + s_m = n$ and the order is not relevant. Since order is ignored, an arbitrary ordering can be applied, for convenience. By convention, partitions are represented in a non-increasing order, i.e., $s_1 \geq s_2 \geq \ldots \geq s_m$.

A useful representation of a partition $(s_1, s_2, \ldots, s_m)$ is its *Ferrer's diagram*, depicted as m rows of dots, the ith row containing s_i of them. For example, Figure 2.1 shows the Ferrer's diagrams for one possible partition of 15 into 5 parts. An interesting result can be obtained from the Ferrer's diagram of a partition of an integer n into m parts. If we "transpose" this diagram so that the rows

become columns and the columns become rows, the resulting Ferrer's diagram represents a partition of n with largest part m. This one-to-one correspondence indicates that there are as many partitions of n into m parts as there are of n with largest part m.

We see in Section 2.2 that generating functions prove very useful in enumerating and counting solutions to partition problems.

Set partitions

The Stirling number of the second kind, $S(n,m)$, is defined as the number of ways of partitioning a set of n distinct elements into m non-empty subsets. In general, $S(n,m)$ can be computed as

$$S(n,m) = \frac{1}{m!}\sum_{i=1}^{m-1}(-1)^i\binom{m}{i}(m-i)^n.$$

Some special cases are $S(n,1)=1$, $S(n,2)=2^{n-1}-1$, $S(n,n-1)=\binom{n}{2}$ and $S(n,n)=1$.

2.2 Generating functions

A generating function is a power series whose coefficients encode information about a sequence. Given a sequence of integers $\{a_0, a_1, a_2, \ldots\}$, the generating function is defined as

$$\begin{aligned} g(x) &= a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + \ldots \\ &= \sum_{n=0}^{\infty} a_n x^n. \end{aligned}$$

This representation of the sequence is especially useful when no direct formula for a_n is available but a closed-form representation for $g(x)$ is. Often, this closed-form generating function can be obtained from the recurrence that defines the sequence, if such a recurrence is known. The following is a sampling of the other advantages of generating functions.

- The closed-form representation of a generating function provides a compact representation for the sequence.
- A generating function may lead to a direct formula for the elements of the sequence.
- Sequences of two different problems can be proved equal by showing that their generating functions are the same.
- Several generating functions may be combined or multiplied together in a meaningful way, often to provide simpler solutions to problems.
- Expressing a problem as a generating function may provide new insights into it.

We now offer several examples of the applications of generating functions. In Section 2.3, we will demonstrate their utility in dealing with integer compositions and partitions.

Placeholder for a sequence

Consider a sequence whose elements are defined by a closed formula that is already known, say, $a_n = 2^n$, $n \geq 0$. The generating function for the sequence $(a_1, a_2, \ldots, a_n)$ is then

$$g(x) = 1 + 2x + 4x^2 + 8x^3 + 16x^4 + \ldots$$

where the coefficient of x^n is a_n. Using the algebraic identity

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + x^4 + \ldots,$$

we can represent g(x) in a closed form, as

$$g(x) = \frac{1}{1-2x}.$$

This example demonstrates how generating functions can be treated simply as a placeholder for a sequence of integers. As Wilf articulated in [39], "a generating function is a clothesline on which we hang up a sequence of numbers for display."

A two-term recurrence

We now consider a sequence whose closed form solution is unavailable, but whose elements are defined by the recurrence

$$a_{n+1} = 2a_n + 1 \tag{2.1}$$

where $n \geq 0$ and $a_0 = 0$. We seek a generating function of the form

$$g(x) = \sum_{n \geq 0} a_n x^n.$$

Multiplying (2.1) with x^n and summing over the values of n for which the recurrence is valid, we get

$$\begin{aligned} \sum_{n \geq 0} a_{n+1} x^n &= \sum_{n \geq 0} (2a_n x^n + x^n) \\ \frac{g(x) - a_0}{x} &= 2g(x) + \frac{1}{1-x} \\ g(x) &= \frac{x}{(1-x)(1-2x)}. \end{aligned}$$

The unknown numbers a_n are now available in $g(x)$ as the coefficients of x^n. After expanding the generating function and observing the coefficients, we can additionally obtain an explicit formula for the sequence as $a_n = 2^n - 1$, $n \geq 0$.

Fibonacci series

Consider the well-known Fibonacci series $1, 1, 2, 3, 5, 8, 13, \ldots$ defined by the three-term recurrence,

$$F_{n+1} = F_n + F_{n-1}$$

where, $n \geq 1$, $F_0 = 0$ and $F_1 = 1$. Following an approach similar to the one above, we obtain the generating function as

$$g(x) = \frac{x}{1 - x - x^2}.$$

The explicit formula can then be obtained by expanding this generating function in partial fractions. Thus, the closed formula for the nth Fibonacci number is found to be

$$F_n = \frac{1}{\sqrt{5}}(r_+^n - r_-^n)$$

where $r_\pm = (1 \pm \sqrt{5})/2$ and $n \geq 0$.

2.2.1 Analytical and formal power series

We now address the issue of convergence of generating functions, since we have defined them as power series. In this regard, we can treat generating functions either as analytical power series or as formal power series.

Treating a generating function $g(x)$ as an analytical object involves defining the region of x over which $g(x)$ is convergent. This is usually done by assuming that the generating function is defined over a positive radius of convergence, so that $g(x)$ is convergent at points close to 0.

The question of convergence can be avoided altogether if the generating function is considered to be a formal power series. A formal power series has no analytical significance and is thus considered simply as a placeholder for the coefficients of its terms. However, treating generating functions as formal power series necessitates definition of various algebraic operations and identities, as explained in [30].

In this thesis, we treat generating functions as formal power series.

2.3 Partitions and compositions

In this section, we tackle the combinatorial problems of integer compositions and partitions with the algebraic tool of generating functions. We first consider the role of generating functions in enumerating integer compositions and then explore its role with integer partitions.

2.3.1 Compositions

Introduced in Section 2.1, we saw that a composition of an integer n into m positive parts is a sequence of positive integers $(s_1, s_2, \ldots, s_m)$ such that $s_1 + s_2 + \ldots + s_m = n$. Consider the generating function

$$\begin{aligned} g(x) &= \frac{x}{(1-x)}\frac{x}{(1-x)} \\ &= (x + x^2 + x^3 + x^4 + \ldots)(x + x^2 + x^3 + x^4 + \ldots) \end{aligned}$$

In order to determine the coefficient of, say, x^5 in this generating function, we observe that it can be obtained by exactly the four products x^1x^4, x^2x^3, x^3x^2 and x^4x^1. The coefficient of x^5 in $g(x)$ is thus the number of ways of adding two positive integers such that the sum is 5, i.e., the number of compositions of 5 into 2 positive parts. In general, the number of compositions of n into m positive parts is the coefficient of x^n, a_n, in the generating function

$$g(x) = \frac{x^m}{(1-x)^m}.$$

Using Newton's generalized binomial theorem, we find that

$$\begin{aligned} g(x) &= x^m \sum_{i \geq 0} \binom{m+i-1}{m-1} x^i \\ &= \sum_{i \geq 0} \binom{m+i-1}{m-1} x^{m+i} \\ &= \sum_{n \geq m} \binom{n-1}{m-1} x^n. \end{aligned}$$

Therefore, $a_n = \binom{n-1}{m-1}$ for $n \geq m > 0$. Similarly, we can find ([27]) that the generating function for the total number of compositions is

$$\sum_{m \geq 1} (x + x^2 + x^3 + x^4 + \ldots)^m = \frac{x}{1-2x},$$

in which the coefficient of x^n, $a_n = 2^{n-1}$.

Generating functions thus take advantage of the circumstance that the product of algebraic terms in x corresponds to the sum of the powers of the terms. Additionally, the use of generating functions has hereby effectively converted the combinatorial problem of counting compositions into an algebraic one.

2.3.2 Partitions of integers

Integer partitions were introduced in Section 2.1, where we defined them similar to compositions except that their order was irrelevant. We now demonstrate how generating functions can be used to enumerate them.

Consider the generating function

$$\frac{1}{1-x^i} = 1 + x^i + (x^i)^2 + (x^i)^3 + (x^i)^4 + \ldots$$

where each term $(x^i)^r$ is considered to contribute r parts of size i. Then, the coefficient of x^n in

$$\prod_{0<i\leq k} \frac{1}{1-x^i} = (1 + x + (x)^2 + \ldots)(1 + x^2 + (x^2)^2 + \ldots)\ldots(1 + x^k + (x^k)^2 + \ldots) \tag{2.2}$$

is the number of ways of putting together, heedless of order, parts of size $1, 2, 3, \ldots k$ so that they sum to n. Equation (2.2) therefore represents the generating function for the partitions of n into parts no greater than k.

Now consider, instead of a restriction on the largest part, a restriction on the number of parts. In the generating function

$$g(x,y) = \prod_{i>0} \frac{1}{(1-yx^i)}, \tag{2.3}$$

the coefficient of $y^m x^n$ gives the number of partitions of n into exactly m parts, with no restriction on the size of parts. Furthermore, the coefficient of y^m in $g(x,y)$ gives us the generating function for partitions into exactly m parts. MacMahon shows in [27] how the fraction of (2.3) can be expanded in ascending powers of y to obtain

$$g(x,y) = 1 + \frac{yx}{1-x} + \frac{y^2x^2}{(1-x)(1-x^2)} + \ldots + \frac{y^ix^i}{(1-x)(1-x^2)\ldots(1-x^i)} + \ldots,$$

from which it is clear that the coefficient of y^m is

$$g(x) = \frac{x^m}{(1-x)(1-x^2)\ldots(1-x^m)}.$$

In this generating function, the coefficient of x^n gives the number of partitions of n into exactly m parts. As MacMahon points out in [27], it is interesting to note that this value is the same as the coefficient of x^{n-m} in

$$\frac{1}{(1-x)(1-x^2)\ldots(1-x^m)}, \tag{2.4}$$

which suggests that n has as many partitions into m parts as $n-m$ has with no part exceeding m. This becomes evident once the Ferrer's diagram of the partition of n into exactly m parts is considered. When the first column, containing exactly m dots, is removed, we see that the resulting Ferrer's diagram represents a partition of $n-m$ into at most m parts. If transposed, this diagram represents a partition of $n-m$ with no part greater than m, thus explaining the equivalence. We similarly observe that (2.4) is in fact the generating function for partitions into no more than m parts.

2.3.3 Finite-variable and multi-variable generating functions

The generating functions presented until now are all defined on a constant number of variables that is independent of the number of parts of the partition. We therefore refer to them as *finite-variable generating functions.* If such a generating function is defined on a single variable, it can also be called a *single-variable generating function*, or a *counting generating function* since the coefficient of x^n counts the number of solutions with sum n.

Generating functions can also prove very useful when they are defined on as many variables as there are parts. Let $S = (s_1, s_2, s_3, \ldots, s_k)$ be a composition of k parts whose *weight* is defined as $|S| = s_1 + s_2 + s_3 + \ldots + s_k$. Let S_C be the set containing all possible solutions for S. Consider the generating function

$$g(x_1, x_2, x_3, \ldots, x_k) = \sum_{S \in S_C} x_1^{s_1} x_2^{s_2} x_3^{s_3} \ldots x_k^{s_k},$$

from which we pick all terms $x_1^{s_1} x_2^{s_2} x_3^{s_3} \ldots x_k^{s_k}$ with $|S| = n$. These terms enumerate the compositions of n into k parts, the size of the ith part being the value of the exponent of x_i. Such generating functions we call *multi-variable generating functions* or *enumerating generating functions.* They serve to represent the enumeration of all solutions to a given problem.

The multi-variable generating function for partitions of n into at most m parts can be obtained as follows. Consider the Ferrer's diagram of a partition $(s_1, s_2, s_3, \ldots, s_m)$, modified so that the dots are labelled by the parts they belong to. Since $s_1 \geq s_2 \geq s_3 \geq \ldots \geq s_m$, for every dot labelled m there is at least one labelled i for $1 \leq i < m$. These s_m columns of dots can be represented in the generating function by the factor $1/(1 - x_1 x_2 x_3 \ldots x_m)$. Similarly, for every dot labelled $m-1$ there is at least one labelled i for $1 \leq i < m-1$, thereby contributing a factor of $1/(1 - x_1 x_2 x_3 \ldots x_{m-1})$ and so on. The multi-variable generating function for partitions into at most m parts is therefore

$$g(x_1, x_2, x_3, \ldots, x_m) = \frac{1}{(1 - x_1)(1 - x_1 x_2)(1 - x_1 x_2 x_3) \ldots (1 - x_1 x_2 x_3 \ldots x_m)} \tag{2.5}$$

wherein the terms $x_1^{s_1} x_2^{s_2} x_3^{s_3} \ldots x_m^{s_m}$ with $|S| = n$ enumerate the partitions of n into at most m parts.

Note that replacing all x_i in (2.5) by x results in the single-variable generating function,

$$g'(x) = g(x, x, x, \ldots, x) = \frac{1}{(1 - x)(1 - x^2)(1 - x^3) \ldots (1 - x^m)}$$

which is the same as (2.4). As another example, replacing all odd-numbered x_i by x and even by y in (2.5) gives us the finite-variable generating function,

$$g''(x, y) = g(x, y, x, y, \ldots, x, y) = \sum x^{s_o} y^{s_e}$$

where the coefficient of $x^{s_o} y^{s_e}$ represents the number of ways of partitioning n such that the sum of the odd parts is s_o and that of the even parts is s_e and $s_o + s_e = n$. Note that $g''(x, x) = g'(x)$.

2.4 Grammars and parsing techniques

In preparation for the automation of the constraint graph technique in Chapter 5, we present here an overview of formal grammars and parsing techniques.

Parsing is the process of structuring an input sequence according to a fixed set of guidelines. The input sequence is called a *sentence* and the set of guidelines a *grammar*. A *language* is the set of all possible sentences that a given grammar can represent. A *parser* for a given grammar is a program that parses any sentence that the grammar can represent.

Formally, a grammar is a 4-tuple (V_N, V_T, R, S) where

- V_N is the finite set of *non-terminal symbols*,
- V_T is the finite set of *terminal symbols*,
- R is the set of *production rules* or pairs (P, Q), where P is a sequence of one or more symbols from $V_N \cup V_T$ and Q is a sequence of zero or more symbols from $V_N \cup V_T$, and
- $S \in V_N$ is the *start symbol*.

To illustrate, we define the grammar G by $V_N = \{A, B\}$, $V_T = \{2, +, \times\}$, $S = B$ and R as

$$\begin{aligned} B &\rightarrow B + B \\ B &\rightarrow A \\ A &\rightarrow A \times A \\ A &\rightarrow 2. \end{aligned} \tag{2.6}$$

Some of the sentences this grammar can parse are '2', '$2 + 2 + 2$' and '$2 \times 2 + 2 \times 2 \times 2$'.

As we see here, terminal symbols are what constitute the sentences of a grammar. Non-terminal symbols, however, may not be a part of a sentence. The start symbol S indicates that the first rule applied must replace the specified non-terminal symbol. The set of production rules, R, dictates how a sequence of symbols can be replaced by a different sequence of symbols. Grammars are classified into several types based on the restrictions imposed on these production rules. We describe the Chomsky hierarchy and other classifications in Section 2.4.2.

2.4.1 Parsing

Parsing is the process of structuring an input sequence according to a given grammar. This is done in order to construct a structure that can be processed according to the semantics of the grammar. A *parse tree* of an input sentence I is a tree representation of the structure of I after it is parsed according to the rules of some grammar G. The root node of the parse tree is the start symbol of G. The leaves are all terminal symbols, and if read from left to right, form the sentence I. Consider

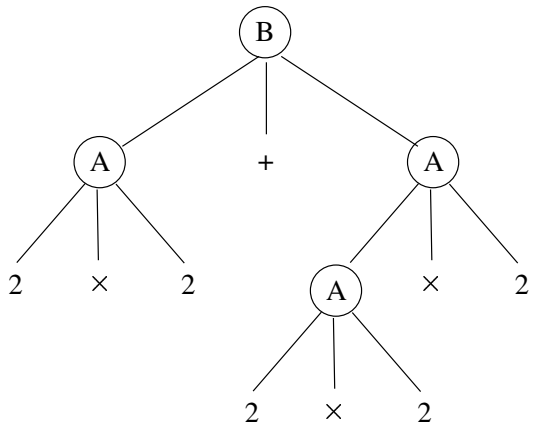


Figure 2.2: Parse tree for $2 \times 2 + 2 \times 2 \times 2$

the grammar G and rules R defined in (2.6). The parse tree for the input sequence '$2 \times 2 + 2 \times 2 \times 2$' is shown in Figure 2.2.

There are several ways in which parsers and parsing techniques can be classified. We discuss the prominent classifications below.

Top-down and bottom-up parsing

In *top-down parsing*, the parse tree is constructed from the root downwards. Beginning with the start symbol, rules are selected and applied in such a way that the end product is the input sentence. In *bottom-up parsing*, the parse tree is constructed from the leaves upwards. Starting from the input sentence, right-hand sides of rules are matched and replaced with the left-hand sides in such a way that the end product is the start symbol.

Directionality

A parser may either be *directional* or *non-directional*. Directional parsers process the input in a fixed direction, either from left to right or from right to left. Non-directional parsers, on the other hand, process the input sentence in any order that seems fit. The disadvantage is that the entire input needs to be available to the parser before parsing can begin. There are several ways to implement non-directional parsers, most notably the Unger method [37] and the CYK method [36].

Directional parsers can further be classified into *deterministic* and *non-deterministic* parsers.

Deterministic and non-deterministic parsers

For some classes of grammars, it is possible to build parsers that do not need to perform a search in order to determine which rule to apply next. Such parsers are therefore deterministic. Non-deterministic parsers are those that do need to include the search functionality. Deterministic parsers are generally more efficient than non-deterministic ones.

Search technique

For non-deterministic parsers, the process of searching for the rule to apply at any given point can be done in two ways, *depth-first search* and *breadth-first search*. Depth-first search begins by applying the first matching rule it encounters. If this rule does not lead to the desired sequence, it backtracks and applies the next matching rule, and so on. In breadth-first search, the parser applies all matching rules simultaneously, thereby producing several states for the parse tree. These are then further processed in a similar manner, simultaneously, to obtain additional parse tree states, and so on, until the desired sequence is reached.

2.4.2 Grammars

There are several ways in which grammars can be classified. We discuss here the Chomsky hierarchy and some other important classifications.

The Chomsky hierarchy

In [16], Chomsky defines four classes of grammars, each subsequent class more restrictive than the previous one. We define each of these grammars below.

- *Type 0 grammars* generate all languages that can be recognized by Turing machines. They have no restrictions and may convert any sequence of symbols into any other.
- *Type 1 grammars*, also called *context-sensitive grammars*, are Type 0 grammars with the restriction that when a rule is applied, only one non-terminal on the left hand side may be replaced, the remaining terminals and non-terminals staying in the same relative order.
- *Type 2 grammars* are like Type 1 grammars, except that the left hand side of a rule may contain only a single symbol, a non-terminal symbol. These grammars are also known as *context-free grammars* because the application of a rule is independent of the context in which the non-terminal appears.
- *Type 3 grammars* are like Type 2 grammars, with the additional constraint that the right hand side of a rule may contain no more than one non-terminal symbol, and this symbol

may only occur as the last one.

It is impossible to construct an algorithm to recognize sentences of arbitrary Type 0 grammars in finite time [25]. Although some algorithms may work for certain classes of Type 0 grammars, a generalized algorithm can never be built. For Type 1 grammars, it is always possible to build such an algorithm, though building an efficient one is not straightforward [23]. Several efficient algorithms exist for Type 2 grammars, which are the most common in practice. Top-down parsing and bottom-up parsing can both be implemented easily for these grammars. Type 3 grammars are the simplest, and are best handled using top-down parsers.

Other classifications

Semantic grammars are grammars that are associated with semantic information. This information allows a parser to extract the meaning of a sentence from its parse tree. In such grammars, every rule is associated with a semantic clause that defines the meaning of that rule.

Leftmost derivation is a parsing technique wherein at any given point, the non-terminal symbol chosen for replacement is always the leftmost one. If all sentences of a grammar can be parsed from left to right with leftmost derivation and a look-ahead of at most k tokens (terminal symbols), it is called an $LL(k)$ grammar.

Chapter 3

Literature review

In this chapter, we explore and review the research that has been conducted in the field of partition and composition enumeration. In Section 3.1, we give an overview of the method of Partition Analysis. Section 3.2 introduces the constraint matrix technique. Finally, we review the Five Guidelines technique in Section 3.2, upon which our methods are largely based.

The treatment of partition enumeration problems using generating functions was first carried out by Euler. His investigations led to intuitive generating functions of the form

$$F(x) = \frac{1}{(1-x)(1-x^2)\dots(1-x^m)}$$

wherein the coefficient of x^n gives the number of partitions of n into parts no greater than m (see Section 2.3.2). A famous result due to Euler is the proof of the number of partitions into odd parts being equinumerous with the number of partitions into distinct parts.

3.1 Partition Analysis

In [27], MacMahon takes a different approach towards partitions than that taken by Euler. He begins by treating a partition of an integer n as a sequence of integers that is explicitly defined by a set of diophantine inequalities and that sum to n, i.e., if $S = \{s_1, s_2, s_3, \dots, s_m\}$ is the integer sequence, the diophantine relations defining an ordinary partition are

$$\begin{aligned} s_1 &\geq s_2 \\ s_2 &\geq s_3 \\ &\vdots \\ s_{n-1} &\geq s_m. \end{aligned} \tag{3.1}$$

He then considers the generating function

$$\frac{1}{(1-\lambda_1 x)(1-\frac{\lambda_2}{\lambda_1}x)(1-\frac{\lambda_3}{\lambda_2}x)\dots(1-\frac{\lambda_m}{\lambda_{m-1}}x)}$$

and notes that it expands to have the general term

$$\lambda_1^{s_1-s_2}\lambda_2^{s_2-s_3}\dots\lambda_{m-1}^{s_{m-1}-s_m}\lambda_m^{s_m}x^{s_1+s_2+\dots+s_m}.$$

If this term is to represent a solution to the diophantine inequalities, then no λ_i in the term should have a negative exponent. He consequently defines the *omega* operator $\underset{\geq}{\Omega}$ as the operator that eliminates all such terms from the generating function. Subsequent application of

$$\lambda_1 = \lambda_2 = \lambda_3 = \dots = \lambda_m = 1$$

would result in the desired generating function. For the ordinary partitions of (3.1), for example, we can define the generating function as

$$\sum x^{s_1+s_2+s_3+\dots+s_m} = \underset{\geq}{\Omega}\frac{1}{(1-\lambda_1 x)(1-\frac{\lambda_2}{\lambda_1}x)(1-\frac{\lambda_3}{\lambda_2}x)\dots(1-\frac{\lambda_m}{\lambda_{m-1}}x)}. \tag{3.2}$$

MacMahon provides a series of identities that can be used for performing the Omega operation, such as

$$\underset{\geq}{\Omega}\frac{1}{(1-\lambda_1 A_1 x^{p_1})(1-\frac{A_2}{\lambda_1}x^{p_2})} = \frac{1}{(1-A_1x^{p_1})(1-A_1A_2x^{p_1+p_2})} \tag{3.3}$$

intended to help implement the operator. Applying (3.3) iteratively on (3.2) results in the generating function

$$\sum x^{s_1+s_2+s_3+\dots+s_m} = \frac{1}{(1-x)(1-x^2)(1-x^3)\dots(1-x^m)},$$

which is the same as that reached by Euler.

Some of the ordinary partition problems for which MacMahon uses this *Partition Analysis* technique include ordinary partitions with only odd parts, ordinary partitions with largest part constrained, and ordinary partitions with each part constrained.

Significant strides were made in Partition Analysis by a series of papers [3, 4, 6, 10, 7, 8, 9, 5, 1, 2] by Andrews, Paule and Riese that used the technique to obtain generating functions for a large range of interesting problems. Andrews first applied Partition Analysis in [3] to address the lecture hall partition problem that was first defined and solved by Bousquet-Mélou and Eriksson in [11] and further refined in [13]. This problem can be considered a finite version of Euler's classical partition theorem. They defined lecture hall partitions as a sequence of integers $S = (s_1, s_2, s_3, \dots, s_m)$ constrained by the inequalities

$$0 \leq \frac{s_1}{1} \leq \frac{s_2}{2} \leq \dots \leq \frac{s_m}{m}.$$

In [13] they prove that the generating function for these partitions is

$$\prod_{i=0}^{n-1} \frac{1}{1-q^{2i+1}},$$

which is the same as that of partitions into odd parts no greater than $2m-1$. In his first paper [3] on Partition Analysis, Andrews demonstrates that Partition Analysis techniques can be applied to the lecture hall partition problem. This involves the use of two new omega rules, both derived from MacMahon's original rules. (Lecture hall partitions were subsequently further generalized by Bousquet-Mélou and Eriksson in [12].)

Andrews' second paper on Partition Analysis [4] deals with integer-sided triangles and partitions in which both first and second difference of parts are non-negative, proving that they are equinumerous. Andrews also derives MacMahon's generating function for 2-rowed plane partitions, a problem area which MacMahon was unable to solve using Partition Analysis techniques beyond the trivial case.

In [6], Andrews, Paule and Riese treat solid partitions on a cube, a generalization of Hermite's problem and k-gon partitions. A bijective proof of the refined lecture hall partition is presented in [10]. Also offered is a treatment of Cayley's compositions [15], which are defined as sequences of integers such that "the first part is unity, and that no part is greater than twice the preceding part." They obtain a recurrence for the problem and offer a proof of Cayley's theorem but are unable to obtain solve for the generating function. In [7], the authors apply Partition Analysis techniques to the enumeration problem of magic squares and magic pentagrams. In [8], they restrict their attention to instances where the number of diophantine inequalities equals the number of variables. Andrews, Paule and Riese further apply these techniques to solve plane partition diamonds in [9], introduce new results for k-gon partitions in [5] and obtain generating functions for two-rowed and hexagonal plane partitions with diagonals in [1, 2]. In most of these investigations, they were aided by the computer algebra package `Omega` that they developed.

3.1.1 The `Omega` package

From their investigations in MacMahon's Partition Analysis, Andrews, Paule and Riese note that the process of obtaining generating functions for a given set of diophantine inequalities using the Omega rules can be automated using computer algebra software. They thus build the `Omega` package as an implementation of these rules.

Judging from the Omega rules and the Partition Analysis technique, the authors observe that a general algorithm for evaluating

$$\underset{\geq}{\Omega} \frac{\lambda^a}{(1-x_1\lambda)(1-x_2\lambda)\dots(1-x_n\lambda)(1-\frac{y_1}{\lambda})(1-\frac{y_2}{\lambda})\dots(1-\frac{y_m}{\lambda})}$$

is all that is necessary. This is done with the help of the *Fundamental Recurrence* that they establish in [6]. They describe the Mathematica implementation, and demonstrate with straight-forward as well as more involved applications the potential of the package. In [7], the authors introduce an improvement to the package by providing a faster reduction algorithm. Further possible improvements are suggested in [8].

3.2 Constraint matrices

In [21], Savage, Corteel and Wilf introduce a simple technique, a refinement of that proposed in [20], of constructing generating functions for the enumeration of integer sequences defined by linear inequalities.

Consider a sequence of integers $S = \{s_1, s_2, s_3, \ldots, s_m\}$ defined by a set of linear constraints of the form $s_i \geq \sum_{j=i+1}^{m} A[i,j]s_j$ where $A[1 \ldots m, 1 \ldots m]$ is any matrix of integers. If $B = A^{-1}$ has only nonnegative integer entries, then the generating function is obtained as

$$\prod_{i=1}^{m} \frac{1}{1-q^{b_i}}$$

where b_i is the sum of entries in column i of B. This technique also produces a natural bijection between solutions for S and partitions into parts from $\{b_1, b_2, \ldots, b_n\}$.

In [21], the constraint matrix technique is applied successfully to a wide range of problems, including lecture hall partitions (Section 3.1), Hickerson partitions [24] and Santos' interpretation of Euler [32, 35, 34]. Despite its simplicity and its many successful applications, there are many simple systems of constraints that the technique cannot address. We see an example in the next section.

3.3 Five guidelines

In [18], Corteel, Lee and Savage present a new approach to obtaining generating functions for partition enumeration problems which overcomes the limitations of the constraint matrix technique and that proves as powerful as the Partition Analysis techniques. They propose five guidelines that can be strategically applied on a system of homogenous inequalities to build a generating function for the sequences of integers constrained by them. In particular, their emphasis is on obtaining a *recurrence* for the generating function when applicable. These guidelines can be viewed as a generalization of the constraint matrix technique or as a simplification of Partition Analysis. We present here a brief overview of the five guidelines.

Consider the sequence of integers $S = \{s_1, s_2, s_3, \ldots, s_m\}$ defined by the set of linear constraints, C, each constraint $c \in C$ of the form $a_0 + \sum_{i=1}^{m} a_i s_i \geq 0$ where a_i are integer constants (possibly

negative). Assume, in addition, that C always contains the constraints $s_i \geq 0$ for $1 \leq i \leq m$. We wish to find the multi-variable generating function

$$F_C = \sum_{S \in S_C} x_1^{s_1} x_2^{s_2} \dots x_n^{s_n},$$

where S_C is the set of all solutions to the integer sequences subject to the constraints in C.

The Five Guidelines proposed in [18] are

1. If C contains only the single constraint $s_1 \geq t$, for integer $t \geq 0$, then

$$F_C(x_1) = x_1^t + x_1^{t+1} + x_1^{t+2} + x_1^{t+3} + \ldots = \frac{x_1^t}{1 - x_1}.$$

2. If C_1 is a set of constraints on variables $s_1, \ldots, s_j$ and C_2 is a set of constraints on variables $s_{j+1}, \ldots, s_n$, then

$$F_{C_1 \bigcup C_2}(x_1, \ldots, x_n) = F_{C_1}(x_1, \ldots, x_j) F_{C_2}(x_{j+1}, \ldots, x_n).$$

3. If a constraint of the form $s_i - a s_j \geq 0$ is implied by C for any integer a, then

$$F_C(X_n) = F_{C_{s_i \leftarrow s_i + a s_j}}(X_n; x_j \leftarrow x_j x_i^a).$$

4. For any constraint c with the same variables as the set C,

$$F_C(X_n) = F_{C \bigcup \{c\}}(X_n) + F_{C \bigcup \{\neg c\}}(X_n).$$

5. For any constraint $c \in C$,

$$F_C(X_n) = F_{C - \{c\}}(X_n) - F_{C - \{c\} \bigcup \{\neg c\}}(X_n).$$

Here, X_n represents the variables $x_1, x_2, \ldots, x_n$.

Guideline 1 gives a trivial generating function that allows s_1 to be assigned any integer value greater than or equal to t. Guideline 2 indicates that if two generating functions are based on mutually exclusive sets of variables, then the generating function for the combined set of constraints can be obtained by multiplying the two together. Intuitively, this guideline encourages breaking the problem into several independent sub-problems, solving them separately and finally combining their results together. Guideline 3 allows the uniform modification of a variable across all constraints in C with the intention of obtaining a simplified set of constraints, and also dictates the effect this modification has on the generating function. Guideline 4 enables us to define a new constraint c and split the set of solutions for the system into two, one containing solutions that satisfy this constraint,

and the other containing those that do not. Guideline 5 is similar, allowing us to choose a constraint c and split the set of solutions into two, one containing solutions that are not constrained by c and the other containing those that are constrained by $\neg c$ but not c. The difference between these two sets represents the required set of solutions. Intuitively, this guideline allows us to convert a 'troublesome' constraint into two others that are preferred.

The following is a simple example of the application of the five guidelines technique. The example can be solved easily using Partition Analysis but, incidentally, not by the constraint matrix technique.

Consider the sequence of integers $S = \{s_1, s_2, s_3\}$ defined by the constraint set C,

$$\begin{aligned} s_1 &\geq s_2 \\ s_1 &\geq s_3 \\ s_2, s_3 &\geq 0. \end{aligned} \tag{3.4}$$

By guideline 5, we can define $F_C(x_1, x_2, x_3) = G(x_1, x_2, x_3) - H(x_1, x_2, x_3)$, where $G(x_1, x_2, x_3)$ is the generating function for

$$\begin{aligned} s_1 &\geq s_2 \\ s_2, s_3 &\geq 0 \end{aligned} \tag{3.5}$$

and $H(x_1, x_2, x_3)$ is that for

$$\begin{aligned} s_1 &\geq s_2 \\ s_3 &\geq s_1 + 1 \\ s_2, s_3 &\geq 0. \end{aligned}$$

Applying guideline 3 in (3.5) as $s_1 \leftarrow s_1 + s_2$ gives us constraints

$$s_1, s_2, s_3 \geq 0 \tag{3.6}$$

and we get $G(x_1, x_2, x_3) = G'(x_1, x_1x_2, x_3)$. Generating function $G'(x_1, x_2, x_3)$ can be obtained from guidelines 1 and 2 as

$$G'(x_1, x_2, x_3) = \frac{1}{(1 - x_1)(1 - x_2)(1 - x_3)}.$$

Therefore,

$$G(x_1, x_2, x_3) = \frac{1}{(1 - x_1)(1 - x_1x_2)(1 - x_3)}.$$

Similarly, $H(x_1, x_2, x_3)$ can be shown to be

$$H(x_1, x_2, x_3) = \frac{x_3}{(1 - x_1x_2x_3)(1 - x_1x_3)(1 - x_3)}.$$

Subtracting H from G gives

$$F_C(x_1, x_2, x_3) = \frac{1 - x_1^2 x_2 x_3}{(1 - x_1)(1 - x_1 x_2)(1 - x_1 x_3)(1 - x_1 x_2 x_3)}$$

the generating function for (3.4).

The advantage of the five guidelines technique over Partition Analysis and the `Omega` package is that it is tailored for obtaining recurrences for generating functions, thereby providing not only a blueprint for a computer implementation of the generating function but also, significantly, a means to prove a closed-form solution of the generating function, if found.

In this thesis, we consider a subset of the partition and composition enumeration problems addressed by the Five Guidelines. In particular, we consider only those systems of constraints that contain inequalities of the form $s_a \geq s_b + w$ where s_a and s_b belong to the sequence of integers under examination and w is any integer.

Chapter 4

Sequences defined by directed graphs

In this chapter, we describe how certain sequences defined by inequalities can be expressed as directed graphs of constraints. In Section 4.2, we propose a technique for constructing the generating functions of sequences defined by these graphs. This method involves strategically decomposing the graph in stages, choosing from a set of simple rules at each stage. We present six rules that can be used for this decomposition and provide examples to illustrate their application. We further prove that the six rules are sufficient for the construction of generating functions of certain families of sequences. In Section 4.3, we consider sequences of constraint graphs and cite the utility of defining them recursively. An additional rule is introduced in order to support the recursive decomposition of these constraint graphs. We then introduce sequence descriptions, explore multi-variable recurrences of generating functions and propose a technique to derive their finite-variable versions. Finally, in preparation for automating the procedure in Chapter 5, we formulate in Section 4.4 the construction tree as a representation of the decomposition process of a constraint graph.

4.1 Directed graphs and sequences

While the Five Guidelines technique (Section 3.3) and the methods of Partition Analysis (Section 3.1) are well suited to obtaining generating functions for any system of homogeneous diophantine inequalities, we find that reducing the scope of investigation to simpler systems enables us to devise simpler techniques that are intuitive yet powerful. In this section, we propose six rules that can be used to construct generating functions for a restricted system of homogenous diophantine inequalities.

Consider a sequence of integers $S = (s_1, s_2, s_3, \ldots, s_m)$ and a set of linear constraints C containing constraints of the form

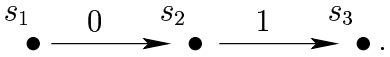
$$s_i \geq s_j + w$$

where $1 \leq i, j \leq m$ and w is an integer, and also constraints $s_i \geq 0$ for every i, $1 \leq i \leq m$. This system of constraints can be represented as a weighted directed graph $G = (V, E)$ where $V = \{s_1, s_2, \ldots, s_m\}$ and every vertex s_i in V corresponds to the variable s_i in the sequence S. In this graph, a directed edge $(s_i, s_j) \in E$ if and only if the constraint $s_i \geq s_j + w$ exists in C and its weight is w. Additionally, every vertex s_i in G implies the constraint $s_i \geq 0$.

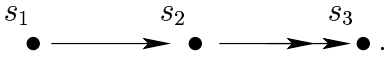
In this way, we can express certain sequence enumeration problems as directed graphs which we refer to as *constraint graphs*. For example, the constraints

$$\begin{aligned} s_1 &\geq s_2 \\ s_2 &\geq s_3 + 1 \\ s_1, s_2, s_3 &\geq 0 \end{aligned}$$

can be represented by the constraint graph



For convenience, an unlabelled edge is considered to have weight 0, and a double-arrowed edge, of weight 1. The above constraints can therefore also be represented by the graph



As another example, the constraints

$$\begin{aligned} s_2 &\geq s_1 \\ s_3 &\geq s_1 \\ s_4 &\geq s_2 + 2 \\ s_4 &\geq s_3 + 1 \\ s_5 &\geq s_2 + 1 \\ s_5 &\geq s_3 + 3 \\ s_6 &\geq s_4 \\ s_6 &\geq s_5 \\ s_1, s_2, s_3, s_4, s_5, s_6 &\geq 0 \end{aligned}$$

on the sequence $S = (s_1, s_2, s_3, s_4, s_5, s_6)$ can be represented by the constraint graph of Fig. 4.1.

Despite the restrictions that constraint graphs have on the type of inequalities they can represent, their scope is large enough to encompass several famous problems, such as two-rowed plane-partitions

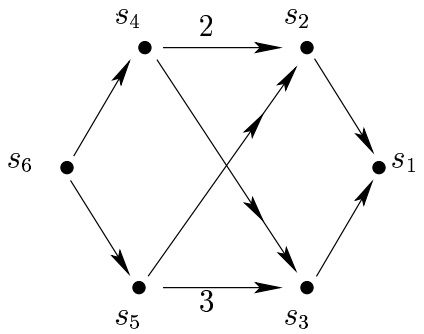


Figure 4.1: Example of a constraint graph representing a given system of constraints.

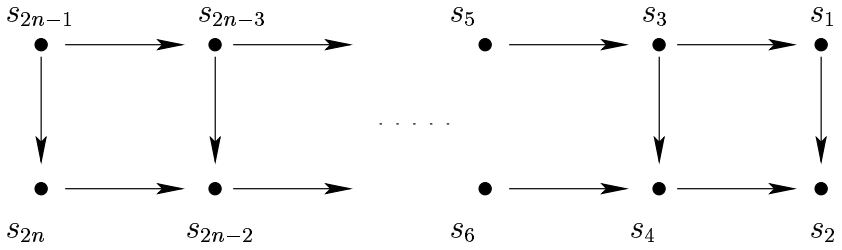


Figure 4.2: Constraint graph for 2-rowed plane partitions [4]

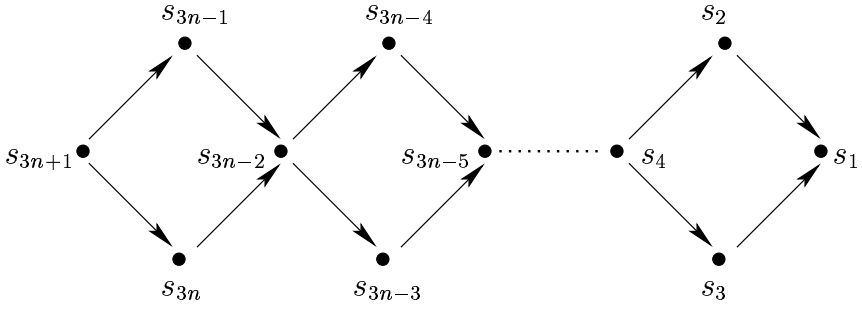


Figure 4.3: Constraint graph for plane partition diamonds [9]

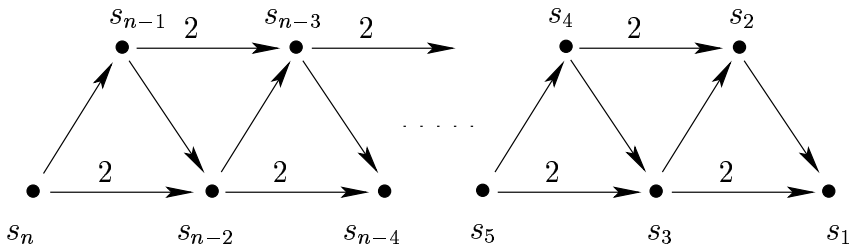


Figure 4.4: Constraint graph for Gordon partitions [22]

[28] of Figure 4.2, plane-partitions diamonds [9] of Figure 4.3 and Gordon partitions [22] of Figure 4.4.

We next show how the generating function of a sequence enumeration problem may be derived from its constraint graph by systematic application of the simple rules that we propose.

4.2 Constructing generating functions for constraint graphs

In this section, we present a technique for the construction of generating functions for sequences defined by constraint graphs and propose six rules, consisting of two terminal graph cases and four graph operators, as the tools that can be applied in conjunction with this technique.

We begin by describing the technique. Let G be a constraint graph whose generating function we wish to obtain. To construct the generating function of G, we follow a recursive approach. If G corresponds to one of the terminal graph cases, its generating function can be constructed directly. Otherwise, we strategically apply one of the four graph operators to G, obtain the generating functions of the resulting simpler constraint graphs, and then use these generating functions to build the generating function of G.

In this approach, our goal at each step is to strategically select the graph operator to be applied, in such a way that each of the resulting constraint graphs is in some way closer to being a terminal graph. This process of reducing a constraint graph into simpler forms we call *decomposing* of the constraint graph. There may be more than one way of decomposing a given constraint graph, each of which may differ in complexity or elegance but eventually produce the same generating function. The two terminal graph cases we propose are

- single vertex graph, and
- inconsistent graph.

The four graph operators we propose are

- independent vertex operator,

- redundant edge operator,
- incoming edge operator, and
- inclusion-exclusion operator.

In the next subsection, we define the two terminal graphs and describe how to handle them. In Subsection 4.2.2, we introduce the four graph operators that we propose for decomposition. We then illustrate with an example in Subsection 4.2.3 how generating functions can be constructed using these tools. Finally, in Subsection 4.2.4 we prove that the six rules are sufficient for obtaining the generating function of any constraint graph with edges all of weight either 0 or 1.

4.2.1 Terminal graph cases

The generating functions for constraint graphs corresponding to the following two cases can be constructed directly without need for a recursive approach.

Single vertex graph

Constraint graph $G = (V, E)$ is a single vertex graph if and only if $|V| = 1$ and $E = \emptyset$. The only constraint that G represents, therefore, is $s_a \geq 0$ for the single vertex s_a. Guideline 1 of the Five Guidelines (Section 3.3) gives us the generating function for G directly as

$$F_G(x_a) = \frac{1}{1 - x_a}.$$

Inconsistent graph

Define an *inconsistent cycle* C in a constraint graph G as a directed cycle in which the sum of the weights of edges is positive. An *inconsistent graph* is a constraint graph that contains at least one inconsistent cycle. For example, the constraint graph of Fig. 4.5 is inconsistent because it contains inconsistent cycle $((s_2, s_3), (s_3, s_4), (s_4, s_5), (s_5, s_7), (s_7, s_2))$ of total weight 1.

Theorem 1 *The generating function of an inconsistent graph G is*

$$F_G(x_1, \ldots, x_m) = 0.$$

Proof. Let $C = ((s_1, s_2), (s_2, s_3), \ldots, (s_{k-1}, s_k), (s_k, s_1))$ be an inconsistent cycle in G with corresponding edge weights $(w_1, w_2, \ldots, w_k)$. Then, in terms of constraints, $s_i \geq s_{i+1} + w_i$ $(1 \leq i < k)$ and $s_k \geq s_1 + w_k$. Adding all constraints, we get

$$\begin{aligned} (s_1 + s_2 + s_3 + \ldots + s_k) &\geq (s_1 + s_2 + s_3 + \ldots + s_k) + (w_1 + w_2 + w_3 + \ldots + w_k) \\ 0 &\geq w_1 + w_2 + w_3 + \ldots + w_k, \end{aligned} \tag{4.1}$$

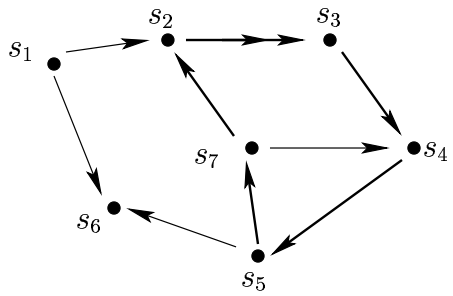


Figure 4.5: An inconsistent graph

which is impossible, since by definition of an inconsistent cycle,

$$w_1 + w_2 + w_3 + \ldots + w_k > 0.$$

Since no sequence $s_1, s_2, \ldots, s_m$ can satisfy these constraints,

$$F_G(x_1, \ldots, x_m) = 0.$$

■

We show in Section 4.2.4 how an inconsistent graph may be introduced during the decomposition of some constraint graph.

4.2.2 Graph operators

We propose the following graph operators for application on a constraint graph G when none of the terminal graph cases apply.

Independent vertex operator

For a directed edge $e = (u, v)$, u is called the *tail* of e and v the *head* of e. For a constraint graph G and a vertex v of G, the *outdegree* of v is the number of edges with tail v and the *indegree* of v is the number of edges with head v. The *degree* of v is the sum of its outdegree and indegree.

Define an *independent vertex* of a graph G to be a vertex v with degree 0. The *independent vertex operator* allows us to remove an independent vertex s_a from the graph G, obtain the generating function $F_{G'}$ for the resulting graph G', and then derive the generating function for G, F_G by suitably modifying $F_{G'}$. Formally, the independent vertex operator, V, takes a vertex s_a and a graph G' not containing s_a as operands, and returns a graph G containing s_a. It can be expressed as

$$G = V(s_a, G').$$

The generating function for the single vertex graph of s_a is

$$F_a = \frac{1}{1 - x_a}$$

as shown in Section 4.2.1. Using Guideline 2 (Section 3.3) to combine F_a and $F_{G'}$, we get the generating function of G as

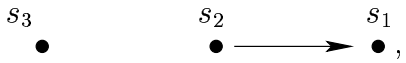
$$F_G(x_1, \ldots, x_n, x_a) = \frac{F_{G'}(x_1, \ldots, x_n)}{1 - x_a}.$$

As an example, assume we know the generating function of the following constraint graph G',

to be

$$F_{G'}(x_1, x_2) = \frac{1}{(1 - x_2)(1 - x_1 x_2)}.$$

To find the generating function of the graph G,



we apply the independent vertex operator as

$$G = V(s_3, G')$$

and thus obtain generating function as follows.

$$\begin{aligned} F_G(x_1, x_2, s_3) &= F_{G'}(x_1, x_2, x_3)\frac{1}{(1 - x_3)} \\ &= \frac{1}{(1 - x_2)(1 - x_1 x_2)(1 - x_3)}. \end{aligned}$$

Redundant edge operator

If, for an edge $e = (s_a, s_b)$ in a constraint graph G, the set of sequences satisfying $G = (V, E)$ is the same as the set of sequences satisfying $G' = (V, E - e)$, e is considered *redundant.* For example, in the constraint graph of Fig. 4.6, the edges (s_3, s_2) and (s_3, s_4) are redundant.

The *redundant edge operator* allows us to *remove* a redundant edge (s_a, s_b) from a graph G without modifying its generating function. Formally, the redundant edge operator, R, takes an edge (s_a, s_b), its weight w and a graph G' containing s_a and s_b but not (s_a, s_b) as operands, and produces a graph G containing (s_a, s_b) as the result. It can be expressed as

$$G = R((s_a, s_b), w, G').$$

Since the removal of a redundant edge from a constraint graph does not in any way change its set of solutions, the generating function is not affected, i.e.,

$$F_G(x_1, \ldots, x_n) = F_{G'}(x_1, \ldots, x_n)$$

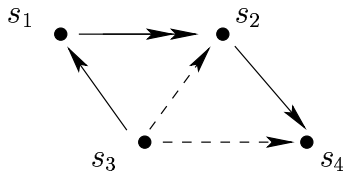


Figure 4.6: Constraint graph with redundant edges (s_3, s_2) and (s_3, s_4)

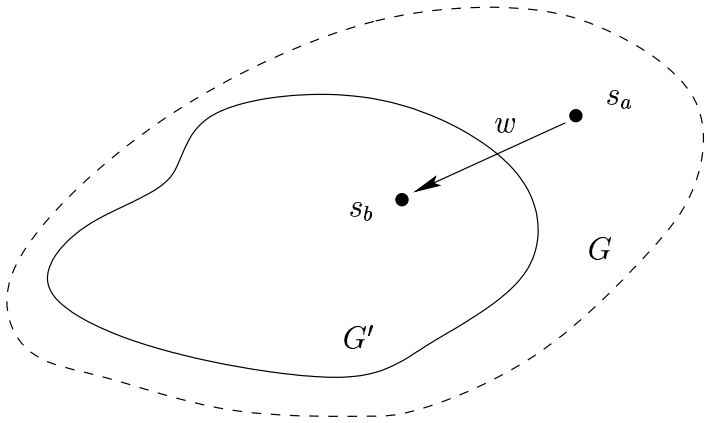


Figure 4.7: Incoming edge (s_a, s_b)

Incoming edge operator

Define an *incoming edge* of a graph G to be an edge (s_a, s_b) where vertex s_a has outdegree 1 and indegree 0. In Fig. 4.7, edge (s_a, s_b) of weight w is an incoming edge.

The incoming edge operator allows us to *remove* any incoming edge (s_a, s_b) of nonnegative weight w and vertex s_a from the graph G, obtain the generating function $F_{G'}$ for the resulting graph G', and then derive the generating function for G, F_G, by suitably modifying $F_{G'}$.

Formally, the incoming edge operator, I, takes an edge (s_a, s_b), its weight w, and a graph G' containing s_b but not s_a and (s_a, s_b), and vertex s_a, as operands and produces a graph G containing s_a and (s_a, s_b). It can be expressed as

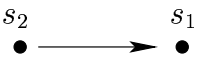
$$G = I((s_a, s_b), w, G').$$

As seen in Section 3.3, the generating function for G is

$$F_G(x_1, \ldots, x_b, \ldots, x_n, x_a) = \frac{F_{G'}(x_1, \ldots, x_b x_a, \ldots, x_n) x_a^w}{1 - x_a}.$$

Thus, once $F_{G'}$ has been obtained, F_G can be obtained by replacing all occurrences of x_b in $F_{G'}$

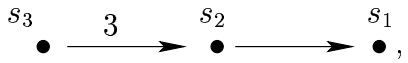
by $x_b x_a$ and then multiplying the result by $x_a^w/(1-x_a)$. For example, assume that we know the generating function of the following constraint graph G'



to be

$$F_{G'}(x_1, x_2) = \frac{1}{(1-x_2)(1-x_1x_2)}.$$

We can find the generating function of the graph G,



by applying the incoming edge operator as

$$G = I((s_3, s_2), 3, G')$$

and thus obtain the generating function as

$$\begin{aligned} F_G(x_1, x_2, s_3) &= F_{G'}(x_1, x_2x_3)\frac{x_3^3}{(1-x_3)} \\ &= \frac{x_3^3}{(1-x_3)(1-x_2x_3)(1-x_1x_2x_3)}. \end{aligned}$$

Inclusion-exclusion operator

Let (s_i, s_j) be a directed edge in G of weight w. As defined in Section 4.1, the constraint c represented by this edge is

$$s_i \geq s_j + w.$$

Consider the complement of this constraint, c',

$$s_i < s_j + w$$

or equivalently,

$$s_i \geq s_j + 1 - w.$$

This new constraint c' can be represented by the edge (s_j, s_i) of weight $(1-w)$. Therefore, replacing a constraint c by its complement c' is equivalent to replacing the edge (s_i, s_j) of weight w by the edge (s_j, s_i) of weight $(1-w)$. We call this the *reversal* of an edge in a constraint graph G.

Let (s_a, s_b) be any arbitrary edge in constraint graph G and let w be its weight. Let G' be the graph obtained by removing (s_a, s_b) from G. Let G'' be the graph obtained from G by reversing (s_a, s_b). The *inclusion-exclusion operator* allows us to derive from G the graphs G' and G'', obtain their generating functions $F_{G'}$ and $F_{G''}$ respectively, and then define the generating function of G,

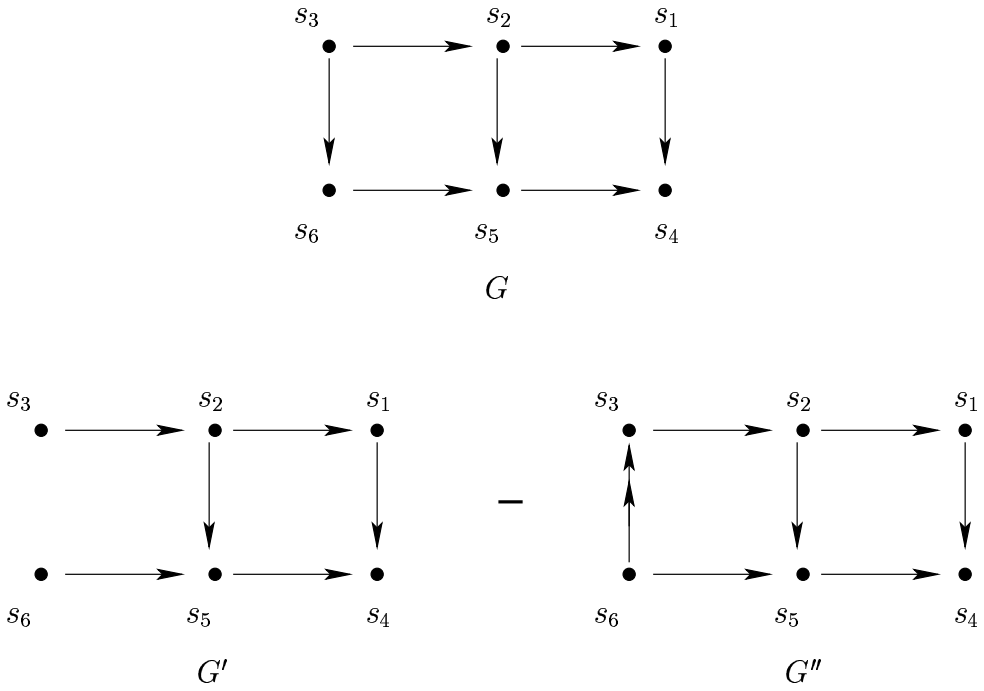


Figure 4.8: Application of the inclusion-exclusion operator to define G in terms of G' and G''

F_G, in terms of $F_{G'}$ and $F_{G''}$. Formally, the inclusion-exclusion operator, $-$, takes an edge (s_a, s_b), its weight w, a graph G' containing s_a and s_b but not (s_a, s_b), and a graph G'' containing (s_b, s_a) of weight $1 - w$ but not (s_a, s_b), as operands and produces a graph G containing (s_a, s_b). It can be expressed as

$$G = -((s_a, s_b), w, G', G'').$$

From Guideline 5 (Section 3.3), since constraint c represented by (s_a, s_b) and weight w is present in G but not in G' or G'' and since G'' additionally contains constraint $\neg c$ in the form of (s_b, s_a), the generating function of G can be obtained as

$$F_G(x_1, \ldots, x_n) = F_{G'}(x_1, \ldots, x_n) - F_{G''}(x_1, \ldots, x_n).$$

For instance, in Fig. 4.8, the generating function for G can be obtained from that of G' and G'' by application of the inclusion-exclusion operator,

$$G = -((s_3, s_4), 0, G', G'')$$

and the generating function is thus

$$F_G(x_1, x_2, x_3, x_4, x_5, x_6) = F_{G'}(x_1, x_2, x_3, x_4, x_5, x_6) - F_{G''}(x_1, x_2, x_3, x_4, x_5, x_6).$$

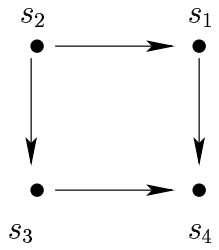


Figure 4.9: Constraint graph G

We summarize the six rules in the following theorem.

Theorem 2 *Let G be a constraint graph. The* Six Rules *for constraint graph decomposition are as follows.*

- *Rule 1: If G is a single vertex s_a, $F_G = 1/(1-x_a)$.*
- *Rule 2: If G is inconsistent, $F_G = 0$.*
- *Rule 3: If $G = V(s_a, G')$ then $F_G = \frac{F_{G'}}{1-x_a}$.*
- *Rule 4: If $G = R((s_a, s_b), w, G')$, then $F_G = F_{G'}$.*
- *Rule 5: If $G = I((s_a, s_b), w, G')$, then $F_G(x_1, \ldots, x_b, \ldots, x_n, x_a) = \frac{F_{G'}(x_1, \ldots, x_b x_a, \ldots, x_n) x_a^w}{1-x_a}$.*
- *Rule 6: If $G = -((s_a, s_b), w, G', G'')$, then $F_G = F_{G'} - F_{G''}$.*

Every application of a rule therefore corresponds to some graph operation or terminal graph case. We define the *decomposition* of a graph G as a sequence of rule applications that decomposes G into one or more terminal graphs. In the next subsection, we demonstrate the technique decomposition with an example.

4.2.3 Construction of generating functions: example

Consider the constraint graph G in Figure 4.9 whose generating function we wish to obtain. Applying the inclusion-exclusion operator for the edge (s_2, s_3), we can represent the graph as

$$G = -((s_2, s_3), 0, G_1, G_2)$$

where G_1 and G_2 are the graphs in Figure 4.10. So, by Rule 6, the generating function for G can be expressed as

$$F_G(x_1, x_2, x_3, x_4) = F_{G_1}(x_1, x_2, x_3, x_4) - F_{G_2}(x_1, x_2, x_3, x_4). \tag{4.2}$$

Graph G_1 can be represented as the application of the incoming edge operator upon graph G_3 as

$$G_1 = I((s_3, s_4), 0, G_3)$$

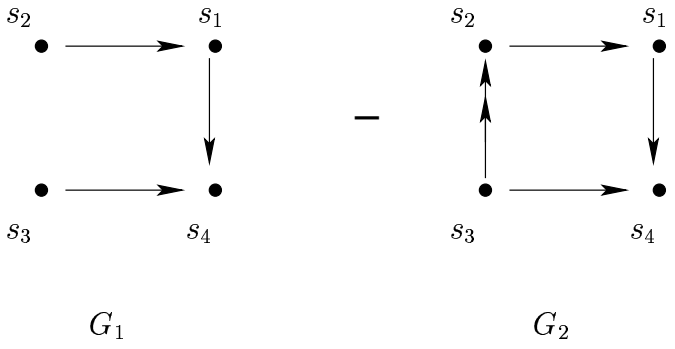


Figure 4.10: Application of the inclusion-exclusion operator on constraint graph G_1 and G_2

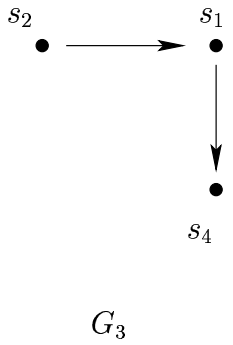


Figure 4.11: Constraint graph G_3

where G_3 is the constraint graph in Figure 4.11. By Rule 5,

$$F_{G_1}(x_1, x_2, x_3, x_4) = \frac{F_{G_3}(x_1, x_2, x_3 x_4)}{(1 - x_3)}. \tag{4.3}$$

Graph G_2 can be expressed as

$$G_2 = R((s_3, s_4), 0, G_4)$$

where G_4 is shown in Figure 4.12. By Rule 4, the generating function remains the same,

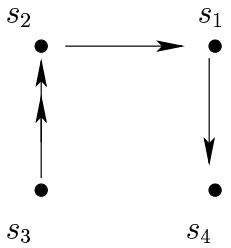
$$F_{G_2}(x_1, x_2, x_3, x_4) = F_{G_4}(x_1, x_2, x_3, x_4). \tag{4.4}$$

Constraint graph G_4 can be expressed as

$$G_4 = I((s_3, s_2), 1, G_3)$$

and its generating function, by Rule 5, is

$$F_{G_4}(x_1, x_2, x_3, x_4) = \frac{F_{G_3}(x_1, x_3 x_2, x_4) x_3}{(1 - x_3)}. \tag{4.5}$$



G_4

Figure 4.12: Constraint graph G_4

G_5

Figure 4.13: Constraint graph G_3

Graph G_3 can be obtained by applying the incoming edge operator for (s_2, s_1) upon the graph G_5 of Figure 4.13 since

$$G_3 = I((s_2, s_1), 0, G_5).$$

By Rule 5,

$$F_{G_3}(x_1, x_2, x_4) = \frac{F_{G_5}(x_2 x_1, x_4)}{(1 - x_2)}. \tag{4.6}$$

Graph G_5 can be described in terms of graph G_6 as

$$G_5 = I((s_1, s_4), 0, G_6)$$

where G_6 is the single vertex graph with vertex s_4. Rule 5 gives the generating function of G_5 as

$$F_{G_5}(x_1, x_4) = \frac{F_{G_6}(x_1 x_4)}{(1 - x_1)}. \tag{4.7}$$

and Rule 1 gives that of G_6 as

$$F_{G_6} = 1/(1 - x_4).$$

Figure 4.14: An empty graph

Replacing the value of F_{G_6} in (4.7), we get

$$F_{G_5}(x_1, x_4) = \frac{1}{(1-x_1)(1-x_1x_4)}.$$

Substituting the value of F_{G_5} in (4.6), we get

$$F_{G_3}(x_1, x_2, x_4) = \frac{1}{(1-x_2)(1-x_2x_1)(1-x_2x_1x_4)}.$$

Replacing F_{G_3} in (4.3) gives us

$$F_{G_1}(x_1, x_2, x_3, x_4) = \frac{1}{(1-x_3)(1-x_2)(1-x_2x_1)(1-x_2x_1x_3x_4)}$$

and, similarly, replacing F_{G_3} in (4.5) gives us

$$F_{G_4}(x_1, x_2, x_3, x_4) = \frac{x_3}{(1-x_3)(1-x_3x_2)(1-x_3x_2x_1)(1-x_3x_2x_1x_4)}.$$

Substituting F_{G_4} in (4.4), we get

$$F_{G_2}(x_1, x_2, x_3, x_4) = \frac{x_3}{(1-x_3)(1-x_3x_2)(1-x_3x_2x_1)(1-x_3x_2x_1x_4)}.$$

Finally substituting values of F_{G_1} and F_{G_2} in (4.2), we get the desired generating function

$$\begin{aligned} F_G(x_1, x_2, x_3, x_4) \quad = \quad & \frac{1}{(1-x_3)(1-x_2)(1-x_2x_1)(1-x_2x_1x_3x_4)} - \\ & \frac{x_3}{(1-x_3)(1-x_3x_2)(1-x_3x_2x_1)(1-x_3x_2x_1x_4)}, \end{aligned}$$

the multi-variable generating function for constraint graph G.

4.2.4 Sufficiency of Operators

We now prove that the Six Rules outlined in Theorem 2 are sufficient for constructing the generating function of any constraint graph with all edges of weights either 0 or 1.

Define an *empty graph* $G = (V, E)$ as a constraint graph in which the set E is empty. For example, the constraint graph of Figure 4.14 is an empty graph because it contains no edges.

Lemma 1 *The Six Rules (Theorem 2) are sufficient for constructing the generating function of any empty graph $G = (V, E)$. If $V = \{s_1, s_2, \ldots, s_n\}$, the generating function is*

$$F_G(x_1, \ldots, x_n) = \prod_{i=1}^{n} \frac{1}{(1 - x_i)}$$

where $n = |V|$.

Proof. We prove this by induction on n.

Let $n = 1$. The graph is a single vertex graph with vertex s_1. By Rule 1, its generating function is

$$F_G(x_1) = \frac{1}{(1 - x_1)}.$$

Assume Lemma 1 is true for all $n < k$. We now prove that the above proposition is true for $n = k$. Let $G_k = (V_k, E_k)$ be an empty graph with $V_k = \{s_1, s_2, \ldots, s_k\}$. It can be represented as

$$G_k = V(s_k, G_{k-1})$$

where $G_{k-1} = (V_{k-1}, E_{k-1})$ with $V_{k-1} = \{s_1, s_2, \ldots, s_{k-1}\}$. By Rule 3,

$$F_{G_k}(x_1, \ldots, x_{k-1}, x_k) = \frac{F_{G_{k-1}}(x_1, \ldots, x_{k-1})}{1 - x_k} \tag{4.8}$$

Observe that, by induction, the generating function of G' is

$$F_{G_{k-1}}(x_1, \ldots, x_{k-1}) = \prod_{i=1}^{k-1} \frac{1}{(1 - x_i)}. \tag{4.9}$$

Replacing this value of $F_{G_{k-1}}$ in 4.8,

$$\begin{aligned} F_{G_k}(x_1, \ldots, x_{k-1}, x_k) &= \frac{1}{1 - x_k} \prod_{i=1}^{k-1} \frac{1}{(1 - x_i)} \\ &= \prod_{i=1}^{k} \frac{1}{(1 - x_i)}. \end{aligned}$$

■

Define an *outgoing edge* in a constraint graph G as an edge (s_a, s_b) where the indegree of s_b is 1 and the outdegree is 0. Define the *underlying undirected graph* of a graph G, denoted by $U(G)$, to be the undirected graph that possesses the same set of vertices V and set of edges E as in G, except that the edges in $U(G)$ are undirected.

Lemma 2 *Let G be a constraint graph with edges of weights either 0 or 1, such that $U(G)$ contains a cycle C. Let E' be the set of directed edges in G that correspond to the edges in C. There exists a subset $E'' \subset E'$ such that reversing the direction of the edges in E'' creates an inconsistent (directed) cycle in G.*

Proof. Let $E'_1 \subset E'$ be an edge set such that reversing all edges in E'_1 makes E' a directed cycle. Let $E'_2 = E' - E'_1$. Define W_1 as

$$W_1 = w_{1,1} + w_{1,2} + \ldots + w_{1,r}$$

where $w_{1,i}$ is the weight of edge $e_i \in E'_1$, and $r = |E'_1|$. Similarly, define W_2 as

$$W_2 = w_{2,1} + w_{2,2} + \ldots + w_{2,n-r}$$

where $w_{2,i}$ is the weight of edge $e_i \in E'_2$ and $n = |E'|$. Note that $|E'_2| = n - r$.

If we reverse all edges in E'_1, the new sum of the weights of edges can be computed as

$$\begin{aligned} W'_1 &= (1 - w_{1,1}) + (1 - w_{1,2}) + \ldots + (1 - w_{1,r}) \\ W'_1 &= r - W_1. \end{aligned}$$

The sum of weights of edges in the directed cycle E' is then

$$\begin{aligned} W_{T_1} &= W'_1 + W_2 \\ W_{T_1} &= r - W_1 + W_2. \end{aligned}$$

If $W_{T_1} > 0$ the directed cycle is inconsistent, the required edge set $E'' = E'_1$ and the proof is done. Otherwise,

$$\begin{aligned} W_{T_1} &\leq 0 \\ r - W_1 + W_2 &\leq 0 \\ W_1 - W_2 &\geq r. \end{aligned} \tag{4.10}$$

If we reverse all edges in E'_2 instead of those in E'_1, the sum of the weights of edges in E'_2 is now

$$\begin{aligned} W'_2 &= (1 - w_{2,1}) + (1 - w_{2,2}) + \ldots + (1 - w_{2,n-r}) \\ W'_2 &= (n - r) - W_2. \end{aligned}$$

The sum of weights of edges in the directed cycle E' is

$$\begin{aligned} W_{T_2} &= W_1 + W'_2 \\ W_{T_2} &= W_1 + (n - r) - W_2. \end{aligned}$$

Substituting for $W_1 - W_2$ from equation 4.10,

$$\begin{aligned} W_{T_2} &\geq r + (n - r) \\ &\geq n. \end{aligned}$$

Since any cycle in G must contain at least 3 edges, $n \geq 3$. Hence, W_{T_2} is positive, and the resultant directed cycle is inconsistent. The required edge set $E'' = E'_2$ and the proof is done. ■

Theorem 3 *Let $G = (V, E)$ be a constraint graph with edges of weights either 0 or 1. The Six Rules (Theorem 2) are sufficient for the construction of the generating function for G.*

Proof. We prove the theorem by induction on the number of edges, $|E|$.

If $|E| = 0$, the graph is an empty graph. By Lemma 1, its generating function can be directly constructed using only the Six Rules. Hence, the proposition is true for $|E| = 0$.

Assume that the proposition is true for $0 \leq |E| < k$. We now show that the proposition holds true for $|E| = k$ as well.

Let $|E| = k$. If G is an inconsistent graph, then by Rule 2 its generating function can be directly inferred and the proposition is true.

Otherwise, if there exists a redundant edge (s_a, s_b) in G, by Rule 4, G can be expressed as

$$G = R((s_a, s_b), w, G')$$

where w is the weight of (s_a, s_b) and G' is a graph containing s_a and s_b but not (s_a, s_b). Since G' has $k - 1$ edges, its generating function $F_{G'}$ can be obtained inductively. Rule 4 defines the generating function of G in terms of $F_{G'}$ and the proposition is therefore true.

If no redundant edge exists in G, we look for an incoming edge (s_a, s_b) in G. If such an edge is present, by Rule 5, G can be expressed as

$$G = I((s_a, s_b), w, G')$$

where w is the weight of (s_a, s_b) and G' is a graph containing s_a and s_b but not (s_a, s_b), and G'' is a graph containing s_a, s_b and (s_b, s_a) of weight $(1 - w)$ but not (s_a, s_b).

Since G' has $k - 1$ edges, its generating function $F_{G'}$ can be obtained inductively. Rule 5 defines the generating function of G in terms of $F_{G'}$ and hence the proposition is true.

Otherwise, if there exists an outgoing edge (s_a, s_b) in G, by Rule 6, G can be expressed as

$$G = -((s_a, s_b), w, G', G'')$$

where w is the weight of (s_a, s_b), G' is a graph containing s_a and s_b but not (s_a, s_b), and G'' is a graph containing (s_b, s_a) of weight $1 - w$ but not (s_a, s_b). Since the number of edges in G' is $k - 1$, its generating function $F_{G'}$ can be inductively obtained using only the Six Rules. Edge (s_b, s_a) in G'' is an incoming edge. The generating function of G'' can be obtained using only the Six Rules, in a similar fashion as shown for the incoming edge case above. Finally, Rule 6 defines the generating function of G, F_G, in terms of $F_{G'}$ and $F_{G''}$ and hence the proposition is true.

If none of the above cases apply, the degree of every vertex in G must be at least 2. Graph $U(G)$ must therefore contain at least one cycle C [38]. Let E' be the set of directed edges in G

that correspond to the undirected edges in C. By Lemma 2, there exists a subset of E' such that reversing the direction of the edges in E'' creates an inconsistent cycle in G. Let E'' be one such subset.

If $|E''| > 0$, we remove an edge (s_a, s_b) from E'' and apply Rule 6,

$$G = -((s_a, s_b), w, G', G'')$$

where w is the weight of (s_a, s_b), G' is a graph containing s_a and s_b but not (s_a, s_b), and G'' is a graph containing (s_b, s_a) of weight $1 - w$ but not (s_a, s_b). Since the number of edges in G' is $k - 1$, its generating function $F_{G'}$ can be inductively obtained using only the Six Rules. If $|E''| > 0$, the generating function of G'', $F_{G''}$, can be obtained by repeating the process recursively, taking G'' instead of G as the graph under consideration. Otherwise, all edges have been reversed and G'' is inconsistent. By Rule 2, $F_{G''}$ can be obtained directly and the proposition is therefore true. ■

4.3 Sequences of constraint graphs

Let $G_1, G_2, \ldots$ be a sequence of graphs which can be defined recursively by first specifying G_1, some specific graph with b vertices, and then recursively specifying G_n as being obtained from G_{n-1} in a prescribed way by adding t new vertices and certain additional edges. Can we get the generating function for G_n if we know the generating function for G_1 and G_{n-1}? In this section, we extend the Six Rules of Theorem 2 by incorporating a new rule that facilitates handling of sequences of constraint graphs. We discuss multi-variable recurrences of generating functions and propose the Special Variable Rules for obtaining finite-variable recurrences.

While the ability to obtain the generating function of any graph in the sequence is clearly useful, the utility of dealing with constraint graph sequences mainly lies in the recurrences obtained for the generating functions, which can help obtain and prove a direct form of the generating function. In this section, therefore, we are primarily interested in sequences of constraint graphs that can be defined recursively.

For example, the 2-rowed plane partition of Figure 4.2 exhibits a simple pattern in which G_n is the graph G_{n-1} of Figure 4.15 with additional vertices s_{2n} and s_{2n-1} and edges (s_{2n}, s_{2n-2}), (s_{2n-1}, s_{2n-3}) and (s_{2n-1}, s_{2n}). In the next section, we introduce a new rule that allows us to specify the decomposition of G_n by taking advantage of this pattern and without having to decompose the entire graph.

Other examples of sequences with simple recursive patterns include those represented by figures 4.3, 4.4 and 4.16.

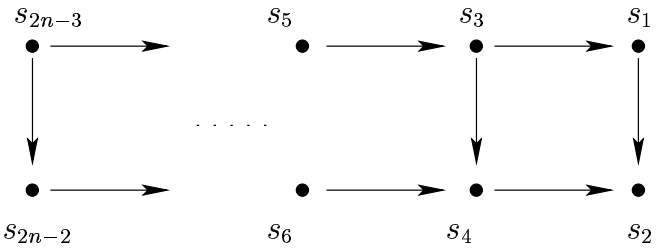


Figure 4.15: Constraint graph G_{n-1} for 2-rowed plane partitions

Figure 4.16: Linear partitions

4.3.1 Graph instance

Consider a constraint graph sequence $S_G = (G_1, G_2, \ldots)$. A constraint graph G' produced during the decomposition of a constraint graph G_n in S_G can be treated as a *graph instance* if it is identical to some graph G_{n-r} in S_G, where $1 \leq r < n$. The graph instance G' is represented simply as G_{n-r}, and is considered a terminal graph (see Section 4.2.1).

The generating function of the graph instance G_{n-r} is the same as that of the constraint graph $G_{n-r} \in S_G$, since the graphs are identical.

We summarize the seven rules of decomposition in the following theorem.

Theorem 4 *Let G be a constraint graph. The* Seven Rules *for constraint graph decomposition are as follows.*

- *Rule 1: If G is a single vertex s_a, $F_G = 1/(1 - x_a)$.*
- *Rule 2: If G is inconsistent, $F_G = 0$.*
- *Rule 3: If $G = V(s_a, G')$ then $F_G = \frac{F_{G'}}{1-x_a}$.*
- *Rule 4: If $G = R((s_a, s_b), w, G')$, then $F_G = F_{G'}$.*
- *Rule 5: If $G = I((s_a, s_b), w, G')$, then $F_G(x_1, \ldots, x_b, \ldots, x_n, x_a) = \frac{F_{G'}(x_1,\ldots,x_b x_a,\ldots,x_n)x_a^w}{1-x_a}$.*
- *Rule 6: If $G = -((s_a, s_b), w, G', G'')$, then $F_G = F_{G'} - F_{G''}$.*
- *Rule 7: If G is a graph instance G_{n-r}, $F_G = F_{G_{n-r}}$.*

The decomposition of a graph G_n in a constraint graph sequence $S_G = (G_1, G_2, \ldots)$ is said to be *recursive* if it involves at least one application of Rule 7.

Figure 4.17: Base case G_1 for 2-rowed plane partitions

4.3.2 Constraint graph sequence description

Let $D_n^{(1)}$ be a decomposition of $G_n \in S_G$ for all $n \in N^{(1)} = \{n_1, n_2, n_3, \ldots\}$. We call $N^{(1)}$ the *index set* of $D^{(1)}$. Since $N^{(1)}$ may not cover the whole range over which S_G is defined, there may additionally exist decompositions $D^{(2)}, D^{(3)}, D^{(4)}, \ldots$ applicable over index sets $N^{(2)}, N^{(3)}, N^{(4)}, \ldots$ respectively, such that the $N^{(j)}$, $j \geq 1$, are pairwise disjoint and their union completely covers S_G. This collection of decompositions and corresponding index sets we call the *constraint graph sequence description* or *sequence description* of S_G, and represent it as Z_{S_G}.

Given a sequence of constraint graphs, S_G, the goal of the decomposition process is then to obtain a sequence description Z_{S_G} for S_G. While the set of decompositions obtained need not necessarily contain a recursive decomposition, it is beneficial if it does so since recursive decompositions translate to generating function recurrences.

4.3.3 Example

We demonstrate the formulation of sequence descriptions for graph sequences by returning to the 2-rowed plane partitions of Figure 4.2. In order to obtain Z_{S_G} for the sequence $S_G = (G_1, G_2, G_3, \ldots)$, we first consider the decomposition of G_1 (Figure 4.17). This graph can be expressed as the application of the incoming edge operator upon graph $G_1^{(1)}$ as

$$G_1 = I((s_1, s_2), 0, G_1^{(1)})$$

where $G_1^{(1)}$ is the single-vertex graph in Figure 4.17. By Rule 5,

$$F_1(x_1, x_2) = \frac{F_1^{(1)}(x_1 x_2)}{(1 - x_1)}. \tag{4.11}$$

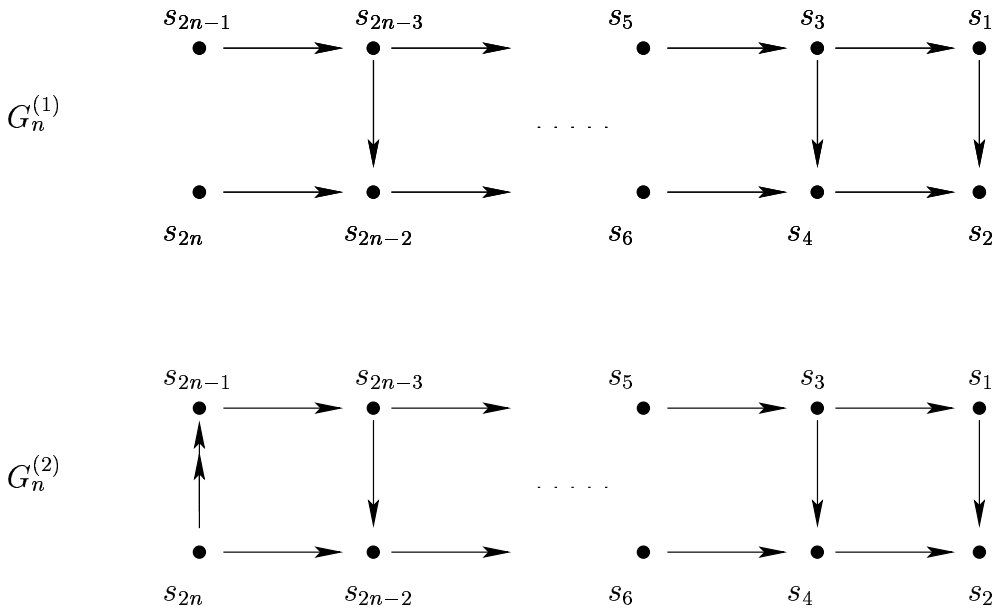


Figure 4.18: Graphs $G_n^{(1)}$ and $G_n^{(2)}$

Since, by Rule 6, $F_1^{(1)}(x_2) = 1/(1-x_2)$, we have in (4.11),

$$F_1(x_1, x_2) = \frac{1}{(1-x_1)(1-x_1x_2)}. \tag{4.12}$$

We next consider graph G_n (Figure 4.2) and attempt to decompose it into G_{n-1} (Figure 4.15) using the 7 Rules. Applying the inclusion-exclusion operator for the edge (s_{2n-1}, s_{2n}), G_n can be represented as

$$G_n = -((s_{2n-1}, s_{2n}), 0, G_n^{(1)}, G_n^{(2)})$$

where $G_n^{(1)}$ and $G_n^{(2)}$ are the graphs in Figure 4.18. So, by Rule 6, the generating function for G_n can be expressed as

$$F_n(x_1, x_2, \ldots, x_{2n-1}, x_{2n}) = F_n^{(1)}(x_1, x_2, \ldots, x_{2n-1}, x_{2n}) - F_n^{(2)}(x_1, x_2, \ldots, x_{2n-1}, x_{2n}). \tag{4.13}$$

Graph $G_n^{(1)}$ can be represented as the application of the incoming edge operator upon graph $G_n^{(3)}$ as

$$G_n^{(1)} = I((s_{2n}, s_{2n-2}), 0, G_n^{(3)})$$

where $G_n^{(3)}$ is the constraint graph in Figure 4.19. By Rule 5,

$$F_n^{(1)}(x_1, x_2, \ldots, x_{2n-1}, x_{2n}) = \frac{F_n^{(3)}(x_1, x_2, \ldots, x_{2n}x_{2n-2}, x_{2n-1})}{(1-x_{2n})}. \tag{4.14}$$

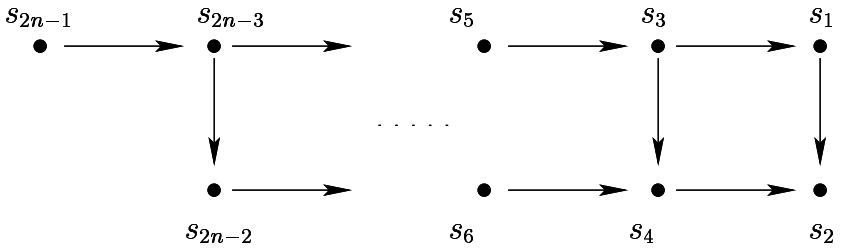


Figure 4.19: Graph $G_n^{(3)}$

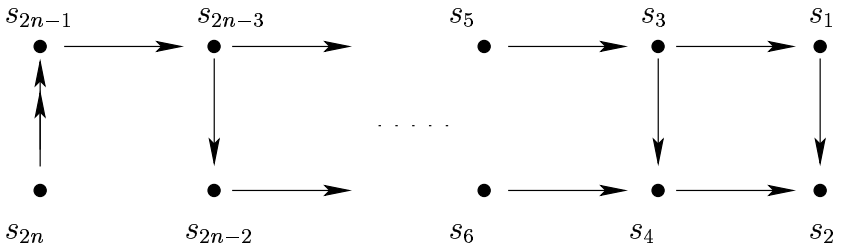


Figure 4.20: Graph $G_n^{(4)}$

Graph $G_n^{(2)}$ can be expressed as

$$G_n^{(2)} = R((s_{2n}, s_{2n-2}), 0, G_n^{(4)})$$

where $G_n^{(4)}$ is shown in Figure 4.20. By Rule 4, the generating function remains the same,

$$F_n^{(2)}(x_1, x_2, \ldots, x_{2n-1}, x_{2n}) = F_n^{(4)}(x_1, x_2, \ldots, x_{2n-1}, x_{2n}). \tag{4.15}$$

Constraint graph $G_n^{(4)}$ can be expressed as

$$G_n^{(4)} = I((s_{2n}, s_{2n-1}), 1, G_n^{(3)})$$

and its generating function, by Rule 5, is

$$F_n^{(4)}(x_1, x_2, \ldots, x_{2n-1}, x_{2n}) = \frac{F_n^{(3)}(x_1, x_2, \ldots, x_{2n-2}, x_{2n-1}x_{2n})x_{2n}}{(1 - x_{2n})}. \tag{4.16}$$

Graph $G_n^{(3)}$ can be obtained by applying the incoming edge operator for (s_{2n-1}, s_{2n-3}) upon the graph $G_n^{(5)}$ of Figure 4.21 since

$$G_n^{(3)} = I((s_{2n-1}, s_{2n-3}), 0, G_n^{(5)}).$$

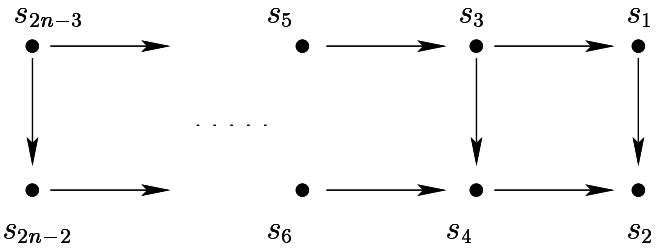


Figure 4.21: Graph $G_n^{(5)}$

By Rule 5,

$$F_n^{(3)}(x_1, x_2, \ldots, x_{2n-2}, x_{2n-1}) = \frac{F_n^{(5)}(x_1, x_2, \ldots, x_{2n-1}x_{2n-3}, x_{2n-2})}{(1 - x_{2n-1})}. \tag{4.17}$$

Graph $G_n^{(5)}$ is simply the graph instance G_{n-1}. By Rule 7,

$$F_n^{(5)}(x_1, x_2, \ldots, x_{2n-2}) = F_{n-1}(x_1, x_2, \ldots, x_{2n-3}, x_{2n-2}). \tag{4.18}$$

Substituting the value of $F_n^{(5)}$ in (4.17), we get

$$F_n^{(3)}(x_1, x_2, \ldots, x_{2n-2}, x_{2n-1}) = \frac{F_{n-1}(x_1, x_2, \ldots, x_{2n-1}x_{2n-3}, x_{2n-2})}{(1 - x_{2n-1})}.$$

Replacing $F_n^{(3)}$ in (4.14) gives us

$$F_n^{(1)}(x_1, x_2, \ldots, x_{2n-1}, x_{2n}) = \frac{F_{n-1}(x_1, x_2, \ldots, x_{2n-1}x_{2n-3}, x_{2n-2}x_{2n})}{(1 - x_{2n-1})(1 - x_{2n})}$$

and, similarly, replacing $F_n^{(3)}$ in (4.16) gives us

$$F_n^{(4)}(x_1, x_2, \ldots, x_{2n-1}, x_{2n}) = \frac{F_{n-1}(x_1, x_2, \ldots, x_{2n}x_{2n-1}x_{2n-3}, x_{2n-2})x_{2n}}{(1 - x_{2n})(1 - x_{2n}x_{2n-1})}.$$

Substituting $F_n^{(4)}$ in (4.15), we get

$$F_n^{(2)}(x_1, x_2, \ldots, x_{2n-1}, x_{2n}) = \frac{F_{n-1}(x_1, x_2, \ldots, x_{2n}x_{2n-1}x_{2n-3}, x_{2n-2})x_{2n}}{(1 - x_{2n})(1 - x_{2n}x_{2n-1})}.$$

Finally substituting values of $F_n^{(1)}$ and $F_n^{(2)}$ in (4.13), we get the desired generating function

$$\begin{aligned} F_n(x_1, x_2, \ldots, x_{2n-1}, x_{2n}) \quad = \quad & \frac{F_{n-1}(x_1, x_2, \ldots, x_{2n-1}x_{2n-3}, x_{2n-2}x_{2n})}{(1 - x_{2n-1})(1 - x_{2n})} - \\ & \frac{F_{n-1}(x_1, x_2, \ldots, x_{2n}x_{2n-1}x_{2n-3}, x_{2n-2})x_{2n}}{(1 - x_{2n})(1 - x_{2n}x_{2n-1})}. \end{aligned} \tag{4.19}$$

This generating function is recursively defined in terms of the generating function of smaller versions of the graph. Since it is defined over all variables $x_1, x_2, \ldots, x_{2n}$, it is called the *multi-variable recurrence* of the generating function. In conjunction with the generating function for the base case G_1, this recurrence defines the generating function for any graph G_n of S_G.

We have thus obtained, from a simple set of decomposition rules and some clever graph manipulation, a formula for the enumerating generating function of any 2-rowed plane partition. This is clearly an improvement over earlier techniques that required separate derivations for different sizes of partitions. Moreover, the generating function recurrences thus obtained can be programmed on a computer to easily represent the generating function for any desired graph size. Although the derivation of the recurrence above seems tedious, it is in fact sufficiently mechanical for a computer program to perform. We describe in Chapter 5 a program we thus developed to construct generating function recurrences for any sequence description provided.

Returning to the multi-variable recurrence of (4.19), define recurrence $H_n(q, x_{2n-1}, x_{2n})$ as the recurrence obtained from F_n by replacing all x_i with q, except for x_{2n-1} and x_{2n}, and all recursive calls to F_{n-1} with calls to H_{n-1}, i.e.,

$$H_n(q, x_{2n-1}, x_{2n}) = \frac{H_{n-1}(q, qx_{2n-1}, qx_{2n})}{(1-x_{2n-1})(1-x_{2n})} - \frac{H_{n-1}(q, qx_{2n}x_{2n-1}, q)x_{2n}}{(1-x_{2n})(1-x_{2n}x_{2n-1})}. \tag{4.20}$$

Since $H_n(q, x_{2n-1}, x_{2n})$ is defined on the variables q, x_{2n-1} and x_{2n}, it allows us to compute $H_{n-1}(q, qx_{2n-1}, qx_{2n})$ and $H_{n-1}(q, qx_{2n}x_{2n-1}, q)$ using the same recurrence, (4.20).

Defining $H_n(q, x_{2n-1}, x_{2n})$ for the base case of (4.12) in a similar way, we get

$$H_n(q, x_{2n-1}, x_{2n}) = \frac{1}{(1-x_{2n-1})(1-x_{2n-1}x_{2n})} \tag{4.21}$$

for $n = 1$.

The recurrence H_n of (4.20), unlike F_n, is defined on a fixed set of variables and is therefore called the *finite-variable recurrence* for the generating function. In conjunction with the base case, it can be used to obtain the counting generating function of any G_n as $H_n(q, q, q)$. The next section describes a technique for deducing finite-variable recurrences from their multi-variable counterparts.

4.3.4 Finite variable recurrence

For most problems that involve obtaining generating functions for graphs in decomposable graph sequences $S_G = (G_1, G_2, \ldots)$ we are interested in finding explicit non-recursive formulae for their single-variable generating functions. A useful approach is to compute and compare single-variable generating functions of arbitrary graphs in the sequence, for example $F_6(q, q, q, \ldots)$ and $F_7(q, q, q, \ldots)$, and to then guess the solution. To assist this process, we would like to automate the computation of single-variable generating functions from the multi-variable recurrences. Finally, we would

also like to prove this conjecture, if true, by mathematical induction. Unfortunately, multi-variable recurrences are ill-equipped for both automation and inductive proofs, as we now explain.

Consider the 2-rowed plane partitions of Section 4.3.3 and some arbitrary graph in the sequence, say G_4. The single-variable generating function of G_4, can be obtained as $F_4(q,q,q,q,q,q,q,q)$. From (4.19), we see that

$$F_4(q,q,q,q,q,q,q,q) = \frac{F_3(q,q,q,q,q^2,q^2)}{(1-q)(1-q)} - \frac{F_3(q,q,q,q,q,q,q^3,q)q}{(1-q)(1-q^2)}.$$

Unfortunately, even though F_4 and F_3 use the same recursive definition, knowledge of computing $F_4(q,q,q,q,q,q,q,q)$ does not imply that of $F_3(q,q,q,q,q^2,q^2)$ or $F_3(q,q,q,q,q,q,q^3,q)$. Hence, it is not possible to use multi-variable recurrence as the inductive step for a proof of the solution. Finite variable recurrences overcome this shortcoming by defining the recurrences over a set of chosen variables, whereby every possible non-q value has a variable parameter to be assigned to.

An additional advantage of finite-variable recurrences over multi-variable ones is apparent when the storage requirements of each are compared. Implemented as a program, the multi-variable recurrences take up a lot of memory space when they are expanded entirely before substituting all variables by q. Finite variable recurrences take up comparatively less space, hence making single-variable generating function computation for larger n more feasible.

Define a *linear sequence description* of a decomposable graph sequence S_G as a sequence description Z_{S_G} in which the set of decompositions is finite and every rule-application references only vertices and edges of the form $x_{m(n)-a}$ and $(x_{m(n)-a_1}, x_{m(n)-a_2})$ for non-negative constants a, a_1 and a_2 independent of graph index n, and where $m(n) = |V(G_n)|$. In this section, we consider only decomposable graph sequences $S_G = (G_1, G_2, \ldots)$ that exhibit the following properties.

- Vertices of G_n are labelled sequentially as x_i, $1 \leq i \leq m(n)$, $m(n) = |V(G_n)|$.
- Graph G_i is a subgraph of G_n for $i < n$.
- There exists a linear sequence description Z_{S_G} for S_G.

We refer to these sequences S_G as *linear graph sequences.*

Consider a linear sequence description Z_{S_G} of a linear graph sequence S_G. The multi-variable generating function F_n of every graph G_n in S_G is defined upon a sequence of variables of the form $x_1, \ldots, x_{m(n)-1}, x_{m(n)}$ where $m(n) = |V(G_n)|$. Consider one such generating function, $F_a(x_1, \ldots, x_{m(a)-1}, x_{m(a)})$, that includes a recursive call to $F_b(x_1, \ldots, x_{m(b)-1}, x_{m(b)})$. Let x_c, $1 \leq c \leq m(a)$, be one of the variables upon which F_a is defined. In generating function F_a, we represent x_c as $x_{m(n)-c'}$ for $n = a$, where $c' = m(a) - c$. However, x_c cannot be represented as $x_{m(n)-c'}$ in F_b because $n = b$. Therefore, x_c is represented here as $x_{m(n)-c'+t}$ instead, where $t = m(a) - m(b)$. Hence, whenever a generating function F_n makes a recursive call to F_{n-r}, $r > 0$, with graph G_{n-r} possessing t fewer vertices than G_n, there occurs an implicit *shift* of all variables by t positions to

the right, relative to the position of the last variable $x_{m(n)}$. Also, after a recursive call, the shifted variables $x_{m(n)+1}, x_{m(n)+2}, \ldots, x_{m(n)+t}$ in F_{n-r} are disregarded because they correspond to vertices that belong to G_n but not to G_{n-r}. We refer to these variables as being *new* in G_n with respect to G_{n-r}.

Consider a linear sequence description Z_{S_G} and a finite set of integers $A = \{a_1, a_2, a_3, \ldots, a_l\}$. For every recursively defined generating function F corresponding to some decomposition D in Z_{S_G} applicable over some index set N, we construct a recursive function H defined upon the recursion F by replacing all variables $x_{m(n)-a'}$, $a' \notin A$, by q and all recursive calls to F_{n-r} by calls to H_{n-r}. We also define the parameter list for H as $q, x_{m(n)-a_1}, x_{m(n)-a_2}, \ldots, x_{m(n)-a_l}$ and the index set for H as N. We similarly construct, for every non-recursively defined generating function F in Z_{S_G}, a non-recursive function H defined upon F by replacing all variables $x_{m(n)-a'}$, $a' \notin A$, by q and setting the parameter list to $q, x_{m(n)-a_1}, x_{m(n)-a_2}, \ldots, x_{m(n)-a_l}$.

If A is defined such that all $H_n(q, x_{m(n)-a_1}, x_{m(n)-a_2}, \ldots, x_{m(n)-a_l})$ so constructed respect the property

$$H_n(q, q, q, \ldots, q, q) = F_n(q, q, q, \ldots, q, q)$$

for every G_n in S_G, we have obtained valid finite-variable versions for the multi-variable generating functions of Z_{S_G} and we call A the *special variable specification* for Z_{S_G}. We next propose the *Special Variable Rules* that specify the composition of such a set A.

Special Variable Rules

Let D_n be a decomposition in the sequence description Z_{S_G} of S_G. Consider the generating function F_n obtained from D_n, applicable over all $n \in N$ for some index set N. Let F_n be defined upon $m(n)$ variables and contain one or more recursive calls to smaller generating functions F_{n-r}, $r > 0$. Consider a recursive call to one such smaller generating function F_{n-r}, defined upon $m(n) - t$ variables. The special variable specification for Z_{S_G}, A, must satisfy the following Special Variable Rules with regards to this recursive call from F_n to F_{n-r}.

- SVR(1) – If the decomposition from G_n to G_{n-r} includes an incoming-edge operation of the form $I((s_{m(n)-a'}, s_{m(n)-a}), w, G')$ for some a', w, G', and $a \geq t$, then $a - t \in A$.
- SVR(2) – If $a \in A$ for some $a \geq t$, then $a - t \in A$.

The special variable specification A for Z_{S_G} is obtained by applying the above rules across all recursive calls F_{n-r} of all generating function recurrences F_n in Z_{S_G}.

Consider a linear sequence description Z_{S_G} and its special variable specification A obtained through the Special Variable Rules. For every decomposition D in Z_{S_G}, we observe the following.

- Rule SVR(1) implies only a finite number of integer constants $a_i \in A$ because D contains only a finite number of incoming-edge operations.

- For each recursive call in the generating function F of D causing a shift of t variables, SVR(2) implies as members of A the integers $a-t, a-2t, \ldots, a-ut$ for $a > ut$. Here, u must be finite because it is bounded by a and t, both of which are independent of n.

Since there are only a finite number of decompositions in Z_{S_G}, we see that the Special Variable Rules result in only a finite set A.

Theorem 5 *Let Z_{S_G} be a linear sequence description and $A = \{a_1, a_2, a_3, \ldots, a_l\}$ be the special variable specification inferred from the Special Variable Rules. For every recursively defined generating function F corresponding to some decomposition D in Z_{S_G} applicable over some index set N, construct a recursive function H defined upon the recursion F by replacing all variables $x_{m(n)-a'}$, $a' \notin A$, by q and all recursive calls to F_{n-r} by calls to H_{n-r}. Also define the parameter list for H as $q, x_{m(n)-a_1}, x_{m(n)-a_2}, \ldots, x_{m(n)-a_l}$ and the index set for H as N. Construct similarly, for every non-recursively defined generating function F in Z_{S_G}, a non-recursive function H defined upon F by replacing all variables $x_{m(n)-a'}$, $a' \notin A$, by q and setting the parameter list to $q, x_{m(n)-a_1}, x_{m(n)-a_2}, \ldots, x_{m(n)-a_l}$. The property*

$$H_n(q, q, q, \ldots, q) = F_n(q, q, q, \ldots, q)$$

is satisfied for all F_n in Z_{S_G}.

Proof. For non-recursive F, the proof is trivial, since all x_i in H are replaced by q anyway. For recursive F, the two essential differences between F and H are (1) the replacement of all $x_{m(n)-a'}$ in F, $a' \notin A$, by q in H and (2) the replacement of all recursive calls to F_{n-r} by calls to H_{n-r}. The second difference is merely cosmetic since it has no effect on the computations. We therefore focus only on the first. We demonstrate the validity of H by showing that no $x_{m(n)-a'}$, $a' \notin A$, could have a value other than q at any point during the computation of $F_n(q, q, q, \ldots, q)$.

Observe that the only two instances in which the value of a variable $x_{m(n)-a}$ may be modified from its default value of q are

1. when $s_{m(n)-a}$ participates in an incoming edge operation $I((s_{m(n)-a'}, s_{m(n)-a}), w, G')$ for some $s_{m(n)-a'}$, w and G', and
2. when $x_{m(n)-a}$ is assigned the value of variable $x_{m(n)-a-t}$ due to a shift of t positions for some recursive call from F_b to F_c.

The first instance is handled by SVR(1), and the second instance by SVR(2). In both cases, the rules ensure that $a \in A$. Since these are the only two possibilities, we see that no $x_{m(n)-a'}$, $a' \notin A$, could ever have a non-q value. ■

Example: 2-rowed plane partitions

In Section 4.3.3, we presented a finite-variable recurrence for 2-rowed plane partitions, H_n, by replacing all x_i in F_n by q, except for x_{2n-1} and x_{2n}. Although these two variables seem to have been selected arbitrarily, we now demonstrate that the Special Variable Rules suggest the same pair as well.

The decomposition of base case G_1 is not recursive and hence does not contribute to the special variable specification A. We summarize the decomposition of graph G_n as

$$\begin{aligned} G_n &= -((s_{2n-1}, s_{2n}), 0, G_n^{(1)}, G_n^{(2)}) \\ G_n^{(1)} &= I((s_{2n}, s_{2n-2}), 0, G_n^{(3)}) \\ G_n^{(2)} &= R((s_{2n}, s_{2n-2}), 0, G_n^{(4)}) \\ G_n^{(4)} &= I((s_{2n}, s_{2n-1}), 1, G_n^{(3)}) \\ G_n^{(3)} &= I((s_{2n-1}, s_{2n-3}), 0, G_n^{(5)}) \\ G_n^{(5)} &= G_{n-1}. \end{aligned} \tag{4.22}$$

Note that graphs $G_n^{(1)}$ and $G_n^{(4)}$ are both defined upon $G_n^{(3)}$. When $F_n^{(1)}$ and $F_n^{(4)}$ recursively call $F_n^{(3)}$, however, they do so with different sets of parameters, and the subsequent decompositions must therefore be treated separately. For simplicity, we consider instead of a single $G_n^{(3)}$, two disparate graphs $G_n^{(3a)}$ and $G_n^{(3b)}$, the former obtained from $G_n^{(1)}$ and the latter from $G_n^{(4)}$. We similarly adopt graphs $G_n^{(5a)}$, $G_n^{(5b)}$, $G_{n-1}^{(a)}$ and $G_{n-1}^{(b)}$ for the two separate decomposition sequences.

Identification of the special variables is now approached by analyzing both $G_{n-1}^{(a)}$ and $G_{n-1}^{(b)}$. Note that $m(n) = |V(G_n)| = 2n$. We first apply the Special Variable Rules to $G_{n-1}^{(a)}$, where the shift is $t = 2$. The incoming-edge operations leading to this graph instance are $G_n^{(1)} = I((s_{2n}, s_{2n-2}), 0, G_n^{(3a)})$ and $G_n^{(3a)} = I((s_{2n-1}, s_{2n-3}), 0, G_n^{(5a)})$. By SVR(1), s_{2n-2} indicates $2 - t = 0 \in A$ and s_{2n-3}, $3 - t = 1 \in A$. Rule SVR(2) does not offer any new members for A because $0, 1 < t$. We next consider $G_{n-1}^{(b)}$, where again the shift is $t = 2$. The only incoming-edge operations leading to this graph are $G_n^{(4)} = I((s_{2n}, s_{2n-1}), 1, G_n^{(3)})$ and $G_n^{(3)} = I((s_{2n-1}, s_{2n-3}), 0, G_n^{(5)})$. Rule SVR(1) does not apply to s_{2n-1} because $1 \not\geq t$, but does apply to s_{2n-3}, indicating $3 - t = 1 \in A$. Rule SVR(2) does not offer any new members for A because $1 < t$.

Thus, we find that $A = \{0, 1\}$, thereby suggesting x_{2n-1} and x_{2n} as the special variables.

4.4 Construction tree

The individual steps in the decomposition of a constraint graph G using the Seven Rules (Theorem 4) are specified in a recursive manner, with each graph operation decomposing a graph into one or more

other graphs, which are then further decomposed by other graph operations, and so on. This process can be represented as a tree with the root node corresponding to the first operation performed on G and its children corresponding to the subsequent operations performed on the resulting graphs, and so on. Since every rule application describes not only the decomposition of a constraint graph G but also the construction of its generating function F_G, the generating function construction process mirrors the decomposition process and can therefore be represented as a tree as well. We call this tree the *construction tree.* We describe in Section 4.4.2 how an application of a given rule can be represented as a node in this tree. Before we do that, however, we identify in Section 4.4.1 certain rules whose applications can be simplified in the construction tree or omitted from it altogether.

4.4.1 Decomposition preprocessing

Consider a decomposition D of a graph G. We first show that if G is not inconsistent, then D need contain no application of the inconsistent graph rule of the Seven Rules. We then describe how applications of the redundant edge rule can be omitted from the construction tree of D as well. Finally, we simplify applications of the incoming-edge rule in the construction tree.

Eliminating Rule 2 (Inconsistent graph)

Lemma 3 *Let $G = -((s_a, s_b), w, G', G'')$ be an application of the inclusion-exclusion rule (Rule 6) anywhere in the decomposition D_n of a graph G_n such that G'' is inconsistent and G is not. Then, edge (s_a, s_b) must be redundant in G.*

Proof. Let $w_1, w_2, \ldots, w_r, w'$ be the weights of the edges $e_1, e_2, e_3, \ldots, e_r, e'$ of an inconsistent cycle in G'', where e' is the edge (s_a, s_b) after reversal, i.e, $e' = (s_b, s_a)$ with weight $w' = 1 - w$. By definition of an inconsistent cycle,

$$\begin{aligned} w_1 + w_2 + \ldots + w_r + w' &> 0 \\ w_1 + w_2 + \ldots + w_r + 1 - w &> 0 \\ w_1 + w_2 + \ldots + w_r &\geq w. \end{aligned} \tag{4.23}$$

In graph G, the edge (s_a, s_b) implies the constraint

$$s_a \geq s_b + w \tag{4.24}$$

and the edges $e_1, e_2, \ldots, e_r$ together imply the constraint

$$s_a \geq s_b + w_1 + w_2 + \ldots + w_r. \tag{4.25}$$

From (4.25) and (4.23),

$$s_a \geq s_b + w$$

which is the same as (4.24). Thus, (s_a, s_b) is redundant in G. ■

Define a *consistent graph* as a graph that contains no inconsistent cycle.

Theorem 6 *Application of the inconsistent graph rule (Rule 2) of the Seven Rules in the decomposition of a consistent graph G_n is unnecessary.*

Proof. Let $(O_1, O_2, O_3, \ldots, O_d)$ be the sequence of rule applications that defines D. Let O_b, $1 \leq b \leq d$, be an application of Rule 2 in D for some inconsistent graph G'' such that there exists no O_i, $1 \leq i < b$, that applies Rule 2. We show that O_b can be eliminated from D.

Observe that b cannot be 1 because G_n is consistent. There therefore exists a graph operation O_a, $1 \leq a < b$, that decomposes some graph G into one or more graphs, of which one is G''. Note that O_a can only be of the form $G = -((s_a, s_b), w, G', G'')$ for some s_a, s_b, w and G', because no other rule application can introduce an inconsistent cycle in a consistent graph. From Lemma 3, we find that (s_a, s_b) must be redundant in G. Notice that the graph obtained by applying the redundant edge rule on G for this edge is G'. Thus, O_a can be replaced in D by the operation $G = R((s_a, s_b), w, G')$. This causes O_b to become irrelevant in D and we can hence eliminate it.

By induction, every application of the inconsistent graph rule (Rule 2) in the decomposition of G_n can be safely eliminated. ■

Eliminating Rule 4 (Redundant edge)

From the definition of the Seven Rules in Theorem 4, we know that for an application of Rule 4 of the form $G = R((s_a, s_b), w, G')$, the resulting generating function is $F_G = F_{G'}$. From the point of view of constructing a generating function, Rule 4 thus makes no contribution and can be ignored. We consequently do not include applications of Rule 4 in the construction tree, and instead replace each $G = R((s_a, s_b), w, G')$ by the subsequent rule application on G'.

Simplifying Rule 6 (Inclusion-exclusion)

From the definition of the Seven Rules in Theorem 4, we know that for an application of Rule 6 of the form $G = -((s_a, s_b), w, G', G'')$, the resulting generating function is $F_G = F_{G'} - F_{G''}$. Since the computation of F_G does not depend on s_a, s_b or w, we can eliminate that information from all applications of Rule 6 in the construction tree.

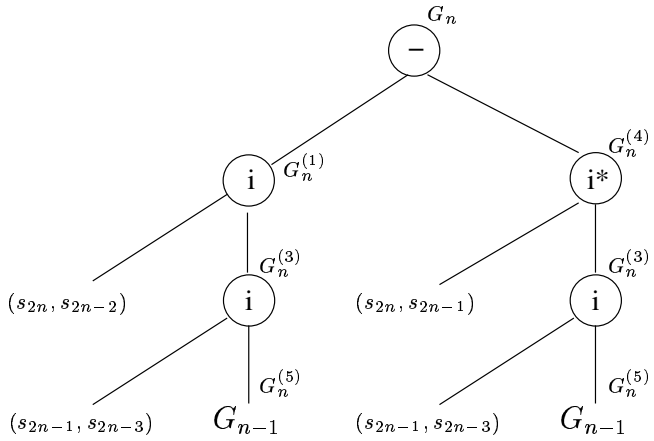


Figure 4.22: Construction tree for the decomposition of G_n for 2-rowed plane partitions

4.4.2 Construction tree creation

Let G_n be a consistent graph and D_n a decomposition for G_n that contains no application of the inconsistent graph rule (Rule 2). The description of the construction tree T_G for any rule application defining G in D_n is dependant on the rule applied, as specified below.

- If G is a single vertex s_a (Rule 1), then T_G is a leaf node with label s_a.
- If $G = V(s_a, G')$ (Rule 3), then the root node of T_G is a node labelled 'V', with first child a node labelled 's_a' and second child the construction tree of G'.
- If $G = R((s_a, s_b), w, G')$ (Rule 4), then T_G is $T_{G'}$, the construction tree of G'.
- If $G = I((s_a, s_b), w, G')$ (Rule 5), then the root node of T_G is labelled 'I', with first child a node labelled '(s_a, s_b)', second child a node labelled 'w' and third child the construction tree of G'. For convenience, the second child can be eliminated if $w = 0$ and the root node is labelled 'i', or if $w = 1$ and the root node is labelled '$i*$'.
- If $G = -((s_a, s_b), w, G', G'')$ (Rule 6), then the root node of T_G is a node labelled '$-$', with first child $T_{G'}$ and second child $T_{G''}$, where $T_{G'}$ and $T_{G''}$ are the construction trees of G' and G'' respectively.
- If G is a graph instance G_{n-r} (Rule 7), T_G is a leaf node with label 'G_{n-r}'.

Example

To illustrate how the decomposition process can be represented as a construction tree, consider the decomposition of G_n in the 2-rowed plane partitions example of Section 4.3.3 as summarized in (4.22). This decomposition can be represented by the construction tree of Figure 4.22.

Construction trees play an important role in the automation of the construction of generating functions, as we will see in the next chapter.

Chapter 5

Automation of generating function construction

The decomposition technique introduced in Chapter 4 enables us to obtain generating functions for a wide range of enumeration problems dealing with sequences defined by directed graphs. The technique, as we saw, also extends itself to sequences of constraint graphs and can produce generating function recurrences that emerge powerful not only for solving any arbitrary constraint graph in the sequence but also for solving and proving a direct form for the generating function. We noted in Section 4.3.3 that once the decompositions for a constraint graph sequence have been established, obtaining the generating functions and recurrences is mechanical and can be automated. In this chapter, we describe the `GFPartitions` package that we developed for this purpose. We begin by giving an overview of the package in Section 5.1. We then define, in Section 5.2, the input requirements of the package and demonstrate with an example. The output generated by the package is presented and explained in Section 5.3. Finally, in Section 5.4, we present a detailed description of the working of the package.

5.1 Overview of the package

The `GFPartitions` package is designed essentially to simplify the task of constructing generating functions. It does this by automating those portions of the construction process that are relatively mechanical and time-consuming. By doing so, the user is able to concentrate on the more interesting and challenging aspects of the problem, such as the decomposition process and the guessing of solutions to the recurrences. The package also automates definition of the finite-variable versions of multi-variable recurrences, thereby providing the user with inductive steps for proofs of the solution.

The `GFPartitions` package expects as input an encoding of the sequence description for the given problem, each decomposition being specified by a construction tree. Using this information, the package builds and outputs multi-variable and finite-variable recurrences of the generating functions.

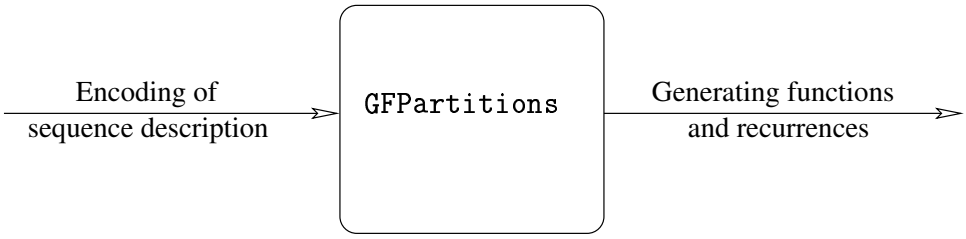


Figure 5.1: Flow of data into and out of the `GFPartitions` package.

The package also outputs a procedure that constructs the complete generating function for any desired constraint graph in the sequence. Figure 5.1 demonstrates the flow of data in the system. Detailed descriptions of the input and the output are provided in the next section.

The `GFPartitions` package handles only those sequence descriptions Z_{S_G} for constraint graph sequences $S_G = (G_1, G_2, G_3, \ldots)$ that satisfy the following.

- Sequence description Z_{S_G} is linear.
- No constraint graph in S_G is inconsistent.
- Every constraint graph G_i, $i \geq 2$, contains exactly a vertices more than G_{i-1}, for some fixed $a \geq 1$.

The package is implemented as a Maple `module` and consists of several procedures. The procedure `Recurrence`, though, is the only procedure that is exposed to the user, since all other procedures are used only internally.

We next describe the input specification for the package.

5.2 Input

In order to be able to construct the generating function recurrences and the complete generating function for a constraint graph sequence S_G, the `GFPartitions` package expects as input the construction tree of every decomposition in the sequence description Z_{S_G} of S_G. In this section, we define construction tree lists and description lists and give examples of how they can be formed.

5.2.1 Construction tree list

As described in Section 4.4, the construction of the generating function or recurrence defined by a given decomposition can be represented as a construction tree. For convenience of input to the `GFPartitions` package, we define a linear representation of this tree as follows.

Define a *proper subtree* of a tree T as any subtree of T other than T itself. The *traversal* of a tree T is the action of visiting every node in T. *Preorder traversal* of a tree T is accomplished by first visiting the root node of T and then making a preorder traversal of each of its proper subtrees from left to right. The *preorder representation* of a tree T is the linear sequence of all nodes of T arranged in the order in which the nodes are visited in a preorder traversal of T.

Define the *construction tree list* of a construction tree T as the preorder representation of T. This is the representation in which we will feed construction trees as input to procedure `GFPartitions`. Consistent with the above definition, construction tree lists can be defined by the grammar

$$\begin{aligned}
\mathit{Graph} &\rightarrow X && (5.1)\\
\mathit{Graph} &\rightarrow \texttt{"."}\ X\ \mathit{Graph} && (5.2)\\
\mathit{Graph} &\rightarrow \texttt{"I"}\ X\ X\ N\ \mathit{Graph} \mid \texttt{"i"}\ X\ X\ \mathit{Graph} \mid \texttt{"i*"}\ X\ X\ \mathit{Graph} && (5.3)\\
\mathit{Graph} &\rightarrow \texttt{"-"}\ \mathit{Graph}\ \mathit{Graph} && (5.4)\\
\mathit{Graph} &\rightarrow \texttt{"\%\%"}\ N \mid \texttt{"\%"} && (5.5)\\
X &\rightarrow \texttt{"x[an+b]"} && (5.6)\\
N &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid \ldots && (5.7)
\end{aligned}$$

where

- vertex s_{an+b} is labelled as a function of n and is represented as `"x[an+b]"`, the variable it is associated with in the generating function,
- edge (s_{an+b}, s_{cn+d}) is represented as two adjacent symbols, `"x[an+b]"` and `"x[cn+d]"`,
- (5.1) corresponds to Rule 1 (single vertex) of the Seven Rules,
- (5.2) corresponds to Rule 3 (independent vertex) of the Seven Rules,
- (5.3) corresponds to Rule 5 (incoming edge) of the Seven Rules,
- (5.4) corresponds to Rule 6 (inclusion-exclusion) of the Seven Rules,
- (5.5) corresponds to Rule 7 (graph instance) of the Seven Rules and graph instance G_{n-r} is represented as two symbols, '`"%%"`' and the integer value of r, and graph instance G_{n-1} is represented simply as '`"%"`'.

In this grammar, the set of non-terminal symbols is $\{\mathit{Graph}, X, N\}$, the set of terminal symbols is $\{\texttt{"I"}, \texttt{"i"}, \texttt{"i*"}, \texttt{"."}, \texttt{"-"}, \texttt{"\%"}, \texttt{"\%\%"}, \texttt{"x[an+b]"}, 0, 1, 2, 3, \ldots\}$ and the start symbol is Graph.

The construction tree list is input into the `Recurrence` procedure of the `GFPartitions` package in the form of a Maple `list`, each terminal symbol occurring as a separate item in the list. We demonstrate with an example.

In Section 4.4.2, we represented the decomposition of G_n of 2-rowed plane partitions as a construction tree (Figure 4.22). We can now convert this tree into a construction tree list, as

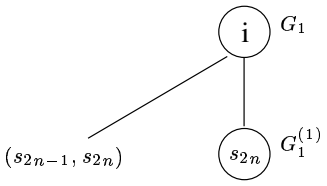


Figure 5.2: Construction tree for G_1 of 2-rowed plane partitions

```
["-", "i", x[2*n], x[2*n-2], "i", x[2*n-1], x[2*n-3], "%%", 1,
      "i*", x[2*n], x[2*n-1], "i", x[2*n-1], x[2*n-3], "%%", 1].
```

5.2.2 Specifying the sequence description

While a construction tree sufficiently describes how to build the generating function for a given decomposition, a sequence description Z_{S_G} may possess more than one decomposition, such as for example, a decomposition for even indices, another for odd and a third for the base case. This necessitates specification of several construction trees, one for each decomposition, in order to completely describe the construction of generating functions and recurrences for S_G. The `GFPartitions` package therefore accepts as input a `list` of d construction tree lists along with a specification of the index set for each. This *description list* can be specified as

$$[tree_{default}, cond_1, tree_1, cond_2, tree_2, \ldots, cond_{d-1}, tree_{d-1}]$$

where $tree_i$ is a construction tree list applicable over all indices n that satisfy $cond_i$ but not $cond_j$, $1 \leq j < i$, and $tree_{default}$ is the construction tree list applicable over all indices that satisfy no $cond_j$, $1 \leq j < d$.

We demonstrate formulation of the description list by returning to the 2-rowed plane partitions example. In Section 4.4.2, we obtained the construction tree of G_n, and in Section 5.2.1, converted it into a construction tree list. Before we build the description list, we first need to obtain the construction tree list for the base case G_1. The construction tree corresponding to the decomposition of G_1 performed in Section 4.3.3 is shown in Figure 5.2 and its construction tree list is therefore

```
["i", x[2*n-1], x[2*n], x[2*n]]
```

for $n = 1$.

The description list can now be formed as

```
[  ["-", "i", x[2*n], x[2*n-2], "i", x[2*n-1], x[2*n-3], "%",
        "i*", x[2*n], x[2*n-1], "i", x[2*n-1], x[2*n-3], "%"],
```

```
    n=1,
    ["i", x[2*n-1], x[2*n], x[2*n]]   ]
```

and can be used as the first parameter for input. The package also expects a second parameter, a list of variable names to replace the "ugly" names of the special variables in the finite-variable recurrences finally output. In this example, we conjecture that there may only be 2 special variables and thereby specify arbitrarily the list `[s, t]` as the second parameter for input.

Thus equipped with all required input parameters, we finally call the `Recurrence` procedure of `GFPartitions` as

```
GFPartitions[Recurrence]([
    ["-", "i", x[2*n], x[2*n-2], "i", x[2*n-1], x[2*n-3], "%",
         "i*", x[2*n], x[2*n-1], "i", x[2*n-1], x[2*n-3], "%"],
    n=1,
    ["i", x[2*n-1], x[2*n], x[2*n]]   ],
    [s, t])
```

We describe the output in the next section.

5.3 Output

The `Recurrence` procedure in the `GFPartitions` package displays as output the multi-variable and finite-variable recurrences of the generating function as well as the complete generating function.

For the 2-rowed plane partitions example of Section 5.2.2, the output generated is

---

```
========================  Multi-variable recurrence  ========================
```

```
Case "default"
```

$$\frac{R_8}{(1-x_{2\,n-1})\,(1-x_{2\,n})} - \frac{R_{15}x_{2\,n}}{(1-x_{2\,n-1}x_{2\,n})\,(1-x_{2\,n})}$$

```
        R[15] -- F(n-1)
                              x[2*n-3] <--- x[2*n-3]*x[2*n-1]*x[2*n]
        R[8] -- F(n-1)
                              x[2*n-2] <--- x[2*n-2]*x[2*n]
                              x[2*n-3] <--- x[2*n-3]*x[2*n-1]
Case n = 1
```

$$\frac{1}{(1-x_{2\,n-1}x_{2\,n})\,(1-x_{2\,n-1})}$$

```
========================  Finite-variable recurrence  ========================
            x[2*n-1] --> s
            x[2*n] --> t
```

$$H(n,q,s,t)$$

```
if n = 1 then
```

$$\frac{1}{(1-st)\,(1-s)}$$

```
else
```

$$\frac{H\,(n-1,q,qs,qt)}{(1-s)\,(1-t)} - \frac{H\,(n-1,q,qst,q)\,t}{(1-st)\,(1-t)}$$

```
fi;
```

The displayed output thus contains both the multi-variable recurrence F, and its finite-variable version H. The recurrences are presented separately for each case or set of applicable indices. In the multi-variable recurrence output, every recursive call is replaced by a placeholder R_p, which is described immediately in the subsequent lines. For each placeholder R_p, the generating function F_{n-k} to be invoked as a recursive call is specified as '`F(n-k)`', and is followed immediately by a specification of values for parameters of F_{n-k}, each parameter x_i cited only if its value is not simply x_i itself.

The finite-variable recurrence, on the other hand, is displayed in a processed form that is ready for implementation in Maple. Note that the variables we provided as input to the program, `s` and `t`, have been used to replace variables `x[2*n-1]` and `x[2*n]` respectively as the special variables in H. Also observe that H, as defined here, is identical to the definition (4.20) of H that we obtained in Sections 4.3.3 and 4.3.4, except that variables x_{2n-1} and x_{2n} have been replaced by s and t respectively. Simple copy-and-paste from the finite-variable output allows us to define the procedure `H(n,q,s,t)` in Maple. A call to `H(n,q,q,q)` for any $n \geq 1$ gives us the single-variable generating function for 2-rowed plane partitions of size n.

Besides the multi-variable and finite-variable recurrences displayed, the program also outputs the complete enumerating generating function for the problem, albeit indirectly. The return value of the program is a procedure `GFInstantiator(n)` that can be used to obtain the enumerating generating function for any desired size of the problem, n. If, for instance, the return value from the example above is assigned to a variable `GF2Rowed`, the call to `GF2Rowed(2)` returns the Maple function

$$(x_1,x_2,x_3,x_4) \rightarrow \frac{x_3^2 x_4 x_1 - 1}{(-1+x_3x_4)(-1+x_3x_4x_1)(-1+x_3)(-1+x_3x_1)(-1+x_2x_4x_3x_1)}$$

which is the complete generating function for 2-rowed plane partitions of size 2.

A quick comparison between the finite-variable recurrence `H` and the complete generating function `GF2Rowed` can be made by executing

```
simplify(H(4,q,q,q) / GF2Rowed(4)(q,q,q,q,q,q,q,q)).
```

The value returned is 1, thereby showing that they both result in the same single-variable (counting) generating function.

5.4 Working

In this section, we describe the inner-workings of the `GFPartitions` package. We first give an overview of the structure of the module along with descriptions of its main procedures. Then, in Sections 5.4.2 and 5.4.3, we describe how these procedures determine and display the multi-variable and finite-variable recurrences. Finally, in Section 5.4.4, the construction of the multi-variable generating function procedure is explained.

5.4.1 Overview

The `GFPartitions` package is implemented as a Maple `module` and consists of the procedures `Recurrence`, `G` and `ListWrap`. Of the three, `Recurrence` is the only procedure that is exposed to the user, and therefore plays a central role in the operations of the package. Procedures `G` and `ListWrap` are internal procedures used by `Recurrence`. We now describe `Recurrence` and `G` in further detail. The role of `ListWrap` is minor and will be explained later in Section 5.4.4.

Procedure `Recurrence`

This procedure accepts as input from the user a description list for the problem and a variable list for the special variables. For each construction tree list T in the description list, `Recurrence` calls `G` to parse and derive the generating function and recurrences for T, as well as obtain other valuable information about it. The procedure then displays the multi-variable recurrences, with placeholders replacing recursive calls and a list of parameter assignments for each such placeholder. The special variables are determined next and the finite-variable recurrences are subsequently constructed and displayed. While in the process of determining the above multi-variable and finite variable output, `Recurrence` also builds, in conjunction with `G` and `ListWrap`, the Maple procedure `GFInstantiator` and returns it to the user in the end.

Procedure `G`

This procedure is responsible for parsing a construction tree list and building the generating function defined by its construction tree. We presented the grammar of construction tree lists in Section 5.2.1. We now identify what type of parser `G` should implement in order to handle sentences of this grammar.

As discussed in Section 2.4.2, $LL(k)$ grammars are those grammars that generate sentences that can be parsed deterministically with a lookahead of no more than k symbols. They can be handled by simple deterministic top-down parsers. An $LL(1)$ grammar is one in which a look-ahead of one token is sufficient to parse any sentence. A *simple* $LL(1)$ grammar (or $SLL(1)$ grammar) has the additional restriction that the right-hand side of every production rule must begin with a different terminal symbol. This ensures that there is always only one possible choice of rule that matches, thereby making $SLL(1)$ parsers very efficient. Observe that the production rules (5.1) through (5.7) abide by this clause, therefore indicating that the grammar for construction tree lists is of type $SLL(1)$. Since the grammar also contains semantic information (generating function specifications) associated with its production rules it can also be classified as an attribute grammar. The semantic attributes in this grammar are derived-attributes, since the generating function information travels up the tree.

Since the grammar is $SLL(1)$, we implement `G` as a simple top-down $SLL(1)$ parser, as follows. Procedure `G` takes as input a construction tree list, considers the first symbol, and determines the production rule that applies. There can only one such rule, say r. The procedure then knows the structure of the next few symbols because of the right hand side of r and therefore proceeds to process them as per the semantics attached to r. During the parsing of the list in `G`, every terminal symbol encountered is handled immediately and is never processed again, i.e., the symbols are *consumed* once processed. When `G` encounters a non-terminal in the right hand side of r, it makes a recursive call to itself, providing as input the unconsumed portion of the construction tree list for further parsing. If the construction tree list originally provided as input satisfies the grammar, the entire list will have been consumed when the `G` finishes parsing.

While procedure `G` parses the construction tree list, it also constructs portions of the generating function at each stage so that sufficient information about the generating function is available to the calling procedure when it returns. We discuss the generating function semantics in further detail in the next section.

x[1]	x[1]
x[2]	x[2]
x[3]	x[3]
⋮	⋮
x[19]	x[17]x[19]
x[20]	x[18]x[19]x[20]

Figure 5.3: Example of the contents of `vRtable`.

5.4.2 Multi-variable generating function recurrences

This section describes how procedures `Recurrence` and `G` work together to output the multi-variable generating functions and recurrences for a given input. As we saw in Section 5.4.1, procedure `Recurrence` iterates through the input description list, sending each construction tree list to `G` to assemble information about the generating function for the corresponding construction tree. We describe `G`'s role in composing the required information first, and then explain how `Recurrence` outputs the information to screen.

Information composed by `G`

During the process of parsing the input construction tree list, procedure `G` collects useful information about the generating functions that later helps `Recurrence` display the multi-variable output. Central to the collection of this information is the table `vRtable` that keeps track of the variable substitutions that occur while traversing down the construction tree. For example, the incoming edge operation '$I((s_a, s_b), w, G')$' causes the variable x_b to be assigned the new value of $x_a x_b$, if s_i is represented by x_i in the generating function. The modified value of a variable x_i remains in effect during the traversal of the entire subtree of that operation and not beyond. This is implemented in `G` as follows. When a variable var_i is first referenced, it is assigned the value var_i in table `vRtable`. Whenever a graph operation needs to modify the value of var_i, it does so in `vRtable` against the entry of var_i, and the modification remains in effect for the scope of the subtree of that operation. When the subtree has been completely traversed, the modification is reverted. Figure 5.3 shows a possible snapshot of the contents of `vRtable` at some point in the parsing of the construction tree.

The actual generating function or recurrence is constructed in `G` as follows. The generating function is defined directly if the applicable production rule contains no non-terminals in the right hand side ((5.1) and (5.5)). Otherwise it is obtained by first calling `G` recursively for every such non-terminal and then performing the required operation upon the generating functions returned by `G`.

In the case of production rule (5.5) (corresponding to Rule 7 of the Seven Rules), the generating function implied is a recursive call of the form F_{n-k} but `G` does not return it as such. Instead, `G` simply returns a unique placeholder of the form 'R_p'. Accompanying details of the recursive call are stored in a table called `substs` under an entry for 'R_p'. These details include a copy of table `vRtable`, which contains information about variable substitutions that have occurred until this point, and the integer value k. The end result is a multi-variable generating function of the form shown in Section 5.3, where all recursive calls are replaced by placeholders.

Processing and displaying by `Recurrence`

Procedure `Recurrence` iterates through the description list provided and sends each construction tree list $T^{(i)}$ to `G`. For every such $T^{(i)}$ it sends, `Recurrence` receives back a generating function $F^{(i)}$ and a table `substs` that contains information relating to each placeholder in $F^{(i)}$.

Next, the procedure iterates through the variable substitutions of every recursive call in every generating function to find the variable with the largest index. For example, in the set $\{x_{2n-1}$, x_{2n+1}, x_{2n}, $x_{2n-2}\}$, variable x_{2n+1} is considered the one with the largest index. This index is stored in variable `max`.

Procedure `Recurrence` finally displays, for every condition $c^{(i)}$ in the description list, the condition $c^{(i)}$ itself, followed by its corresponding generating function $F^{(i)}$. Every $F^{(i)}$ is followed immediately by a description of the recursive call F_{n-k} that each placeholder R_p in the equation represents. The recursive call is first specified as '`F(n-k)`' and is then followed by a list of the variable substitutions recorded in its `vRtable`. Note that, since every F_n is defined over a more variables than F_{n-1}, every variable can be expressed as x_{an+b}. A jump of index from n to $n-k$ therefore causes a shift of ka positions. The variables x_{an+b}, where b is such that $(an+b)+ak >$ `max`, are therefore new variables with respect to F_{n-k}, thus rendering their substitutions irrelevant and thence disregarded. After each of the conditions and corresponding recurrences are displayed, the same is done for the default case as well.

5.4.3 Finite-variable generating function recurrences

We now describe how finite-variable generating functions and recurrences are determined and output. The only information required for this process is that already obtained prior to the display of the

multi-variable recurrences. Consequently, procedure `G` does not play any additional role here and can be ignored for the present. All of the following processing takes place in procedure `Recurrence`.

We introduced the Special Variable Rules in Section 4.3.4, along with a technique for obtaining finite variable recurrences. Procedure `Recurrence` implements essentially the same rules and technique. To begin with, the special variables are identified in two stages, mirroring the two Special Variable Rules, as follows. In the first stage, every variable that occurs on the left hand side of any substitution is identified as a special variable unless its index $a(n+k)+b >$ `max`, as in the previous section. This is equivalent to rule SVR1, since $m =$ `max`, $t = ka$ and the incoming-edge rule always results in a substitution. In the second stage, for every special variable x_{an+b} identified in the first stage, all $x_{a(n+i)+b}$ are made special variables, where $i > 0$ and $a(n+i)+b \leq$ `max`. This is equivalent to SVR2.

Once the special variables have been identified, the rest of the processing is straightforward. Procedure `Recurrence` sorts the special variables in ascending order of index value. This is the order in which they will appear in the parameter list of the finite-variable recurrence. Next, for each generating function recurrence, the placeholders are replaced by recursive calls to function "`H`", providing as parameters the values of the special variables identified above. Following this, all variables in the generating functions that are not special variables are replaced one at a time by the generic common variable q. As a finishing touch, each of the special variables is replaced by a prettier one from the variable list provided at input (see Section 5.2.2). For sake of reference, these replacements are noted in the output.

Finally, the procedure displays for each of the conditions $cond^{(i)}$, the condition itself, followed by the corresponding finite-variable generating function. When all the conditions are exhausted, the generating function for the default case is displayed. This output is tailored to allow easy implementation of the finite variable recurrence `H` in Maple by copying and pasting.

5.4.4 Complete generating function

While the focus in sections 5.4.2 and 5.4.3 was on obtaining recurrences for generating functions applicable over any index n, our focus in this section is on building the complete enumerating generating function for only one given index n.

For the sake of clarity, we choose not to include index n as a parameter to procedure `Recurrence` and decide instead to have `Recurrence` output a procedure `GFInstantiator` that takes as input a value for n and returns as output the complete generating function for that index value. This new procedure can be thought of as an implementation of the multi-variable recurrence for the complete generating function. The essence of our method is then to build a function `f` that mimics the way `G` builds the generating function recursively, along with another function `CompleteGF` that mimics

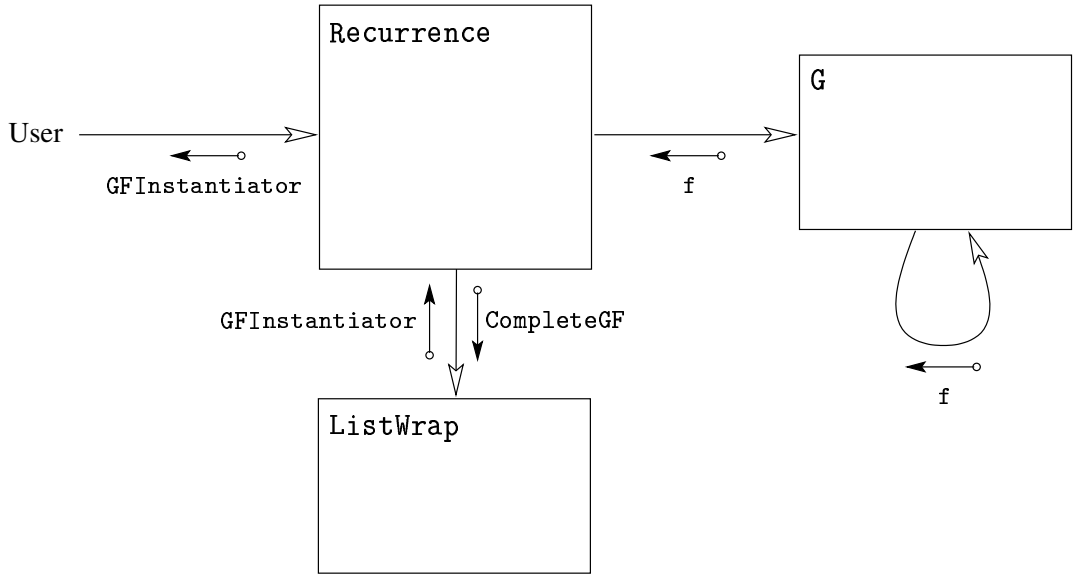


Figure 5.4: Program control flow

`Recurrence` but does more than just display the generating functions, instead connecting the various `f` together programmatically. Absent from both these new functions will be any aspect of parsing, since that is handled by `G` and `Recurrence`.

As we will see, `GFInstantiator` uses the procedures `CompleteGF` and `f` as two mutually-recursive procedures. Procedure `G` builds a hierarchy of functions `f`, while procedure `Recurrence` builds the single procedure `CompleteGF`. The relation between these procedures is portrayed in figures 5.4 and 5.5.

We first describe the construction of `CompleteGF` in `Recurrence` and then the construction of the `f` hierarchy in `G`.

Procedures `Recurrence` and `CompleteGF`

As we saw in Section 5.4.2, procedure `Recurrence` iterates through the input description list, sending each construction tree list to `G`. Procedure `G` then compiles the required information and returns it to `Recurrence`. While this information is being collected, `G` and `Recurrence` simultaneously build portions of the complete generating function. One of the objects returned to `Recurrence` from every call to `G` is a Maple procedure `f` that represents the multi-variable recurrence for the construction tree under consideration. Once all such procedures `f` are collected, `Recurrence` proceeds to dynamically construct Maple procedure `CompleteGF`, defined as follows.

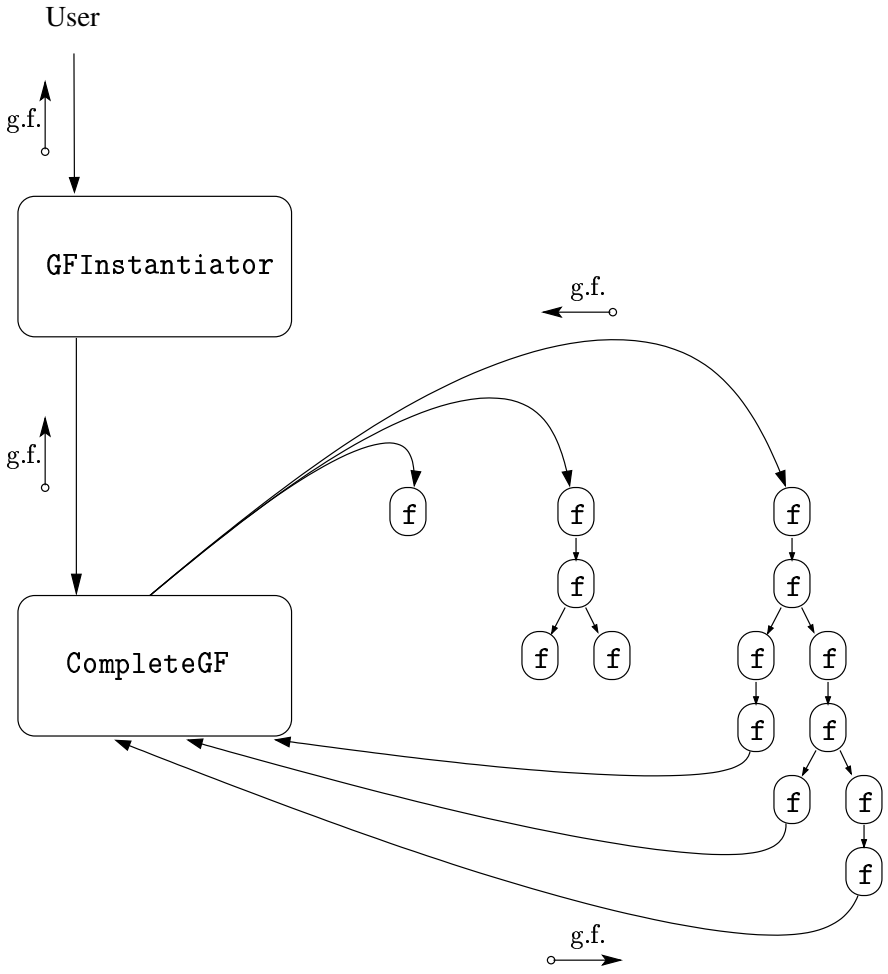


Figure 5.5: Interaction between `GFInstantiator`, `CompleteGF` and `f`

Procedure `CompleteGF` takes as input an index value `n` and a table `vtable` that represents the parameter list of variables. It identifies the first condition that `n` satisfies, and calls the corresponding procedure `f`, providing `n` and `vtable` as input. If none of the conditions are satisfied, the procedure `f` corresponding to the default case is called. Procedure `CompleteGF` thus behaves as a switching device, deciding for a given `n` the `f` it should call.

Procedures `G` and `f`

The main role of procedure `G`, as we have seen, is to parse the input construction tree list. We also saw the role `G` played in determining the generating functions and recurrences in Section 5.4.2. These generating functions, however, do not perform any variable substitutions and are therefore useful only for display purposes. We now investigate a new role for `G`, that of building a generating function recurrence that performs all the generating function manipulations required by the Seven Rules. This procedure, `f`, takes as input an index value `n` and a table `vtable` that represents the parameter list of variables, identical to `CompleteGF`.

Procedure `G` builds `f` in such a way that `f` mimics all the generating function manipulations that `G` performs, including variable substitutions. Internally, `f` may also call other versions of `f` and manipulate the returned generating functions, much in the same way that `G` does. The result is a hierarchy of procedures `f`, mirroring the construction tree of the list `G` parses and, consequently, the tree that `G` traverses. In the case of the graph instance rule, procedure `f` is defined to simply call `CompleteGF` with the new index value and the existing `vtable` that contains the variable substitutions.

Procedure `GFInstantiator`

The recurrence for the complete generating function is, as described above, defined by procedure `CompleteGF` and hierarchy `f`, which are mutually recursive. Although `CompleteGF` seems capable of handling any of the tasks that a multi-variable generating function recurrence is expected to, one of its parameters, `vtable`, is undesirable because it does not lend itself to ease of use. Moreover, as a collection of parameter variables, it is inconsequential when the recurrence is first called because there are no preexisting variable substitutions. The list of parameter variables can in fact be generated automatically based on the information in the construction tree lists provided to the parser `G`. We therefore dynamically construct another Maple procedure, `GFInstantiator`, that wraps around `CompleteGF`, accepting only index value n as input. Additionally, `GFInstantiator` converts the resulting complete generating function equation into a Maple function before returning it to the user. Thus, `GFInstantiator` presents the user with an interface that is easier to use. The construction of this procedure is done in `ListWrap`, which is called by `Recurrence` for this very purpose. The interaction between `GFInstantiator`, `CompleteGF` and `f` can be depicted by Figure 5.5.

Chapter 6

Examples

In this chapter, we approach several constraint graph problems using the Seven Rules of Theorem 4 and the `GFPartitions` module, and demonstrate the simplicity of the approach. Examples chosen include simple straightforward problems, well-known problems that have been solved previously, as well as new problems that we found interesting.

6.1 Ordinary partitions

We begin with the simplest of constraint graph sequences, the ordinary partitions, represented by Figure 6.1. These sequences correspond to simple integer partitions as introduced in Section 2.1. Integer partitions were first treated using generating functions by Euler, who presented several intuitive results. We described many of these generating functions in Section 2.3. We now demonstrate how to approach the problem using the Seven Rules of Theorem 4 and the `GFPartitions` module.

The decomposition of the constraint graph G_n of Figure 6.1 is fairly simple. The base case G_1 contains only a single vertex s_1, and so, G_1 can be defined by the application of Rule 1 for vertex s_1. The general case G_n can be obtained from $G_n^{(1)}$ (Figure 6.2) by an incoming-edge operation (Rule 5), and observing that $G_n^{(1)}$ is the graph instance G_{n-1}, i.e.,

$$\begin{aligned} G_n &= I((s_n, s_{n-1}), 0, G_n^{(1)}) \\ G_n^{(1)} &= G_{n-1}. \end{aligned}$$

Figure 6.1: Ordinary partitions; constraint graph G_n

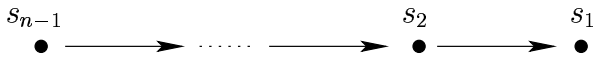


Figure 6.2: Constraint graph $G_n^{(1)}$, or G_{n-1}, for ordinary partitions

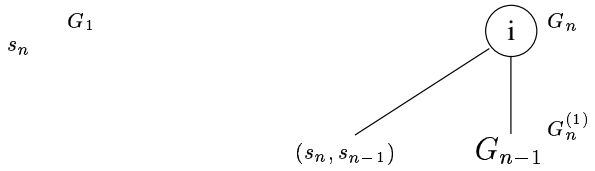


Figure 6.3: Construction trees for G_1 and G_n of ordinary partitions.

Thus, we obtain the construction trees of Figure 6.3 for the base case and the general case. The construction tree list for the base case is simply `[x[n]]` and that of the general case is

```
["i", x[n], x[n-1], "\%"].
```

Assuming that the finite variable recurrence needs no more than one special variable, we invoke procedure `Recurrence` of the `GFPartitions` package as

```
GFOrdinary := GFPartitions[Recurrence](
                    [["i", x[n], x[n-1], "%"], n=1, [x[n]]],
                    [r]).
```

The output generated is

```
========================  Multi-variable recurrence  ========================

Case "default"
```

$$\frac{R_4}{1 - x_n}$$

```
          R[4] -- F(n-1)
                              x[n-1] <--- x[n-1]*x[n]

Case n = 1
```

$$\frac{1}{1 - x_n}$$

```
========================  Finite-variable recurrence  ========================
            x[n] --> r
```

$$H(n,q,r)$$

```
if n = 1 then
```

$$\frac{1}{1-r}$$

```
else
```

$$\frac{H(n-1,q,qr)}{1-r}$$

```
fi;
```

In addition, the `GFInstantiator` procedure is provided as the return value, which we accept in variable `GFOrdinary`. We can thereby obtain the complete generating function for ordinary partitions of any size. For size $n = 5$, for instance, a call to `GFOrdinary(5)` returns the Maple function,

$$(x_1,x_2,x_3,x_4,x_5) \rightarrow \frac{1}{(1-x_5)(1-x_4x_5)(1-x_3x_4x_5)(1-x_2x_3x_4x_5)(1-x_1x_2x_3x_4x_5)}.$$

Returning to the displayed output, we see that the multi-variable recurrence for F_{G_n} is

$$F_{G_n}(x_1,x_2,\ldots,x_{n-1},x_n) = \begin{cases} \frac{1}{1-x_1} & n=1 \\ \frac{F_{G_{n-1}}(x_1,x_2,\ldots,x_{n-1}x_n)}{1-x_n} & otherwise. \end{cases}$$

Similarly, the finite variable recurrence $H_{G_n}(q,r) = F_{G_n}(q,q,\ldots,q,r)$ is

$$H_{G_n}(q,r) = \begin{cases} \frac{1}{1-r} & n=1 \\ \frac{H_{G_{n-1}}(q,qr)}{1-r} & otherwise. \end{cases} \tag{6.1}$$

We illustrate the usefulness of the above finite variable recurrence by using it to prove the following theorem, which allows us to then obtain the closed form generating function for ordinary partitions.

Theorem 7 *The solution to the finite-variable recurrence of 6.1 is*

$$H_{G_n}(q,r) = \prod_{i=1}^{n} \frac{1}{1-q^{i-1}r}. \tag{6.2}$$

Proof. We prove this by induction on n. For $n = 1$, the proposition is clearly true. Assume it is also

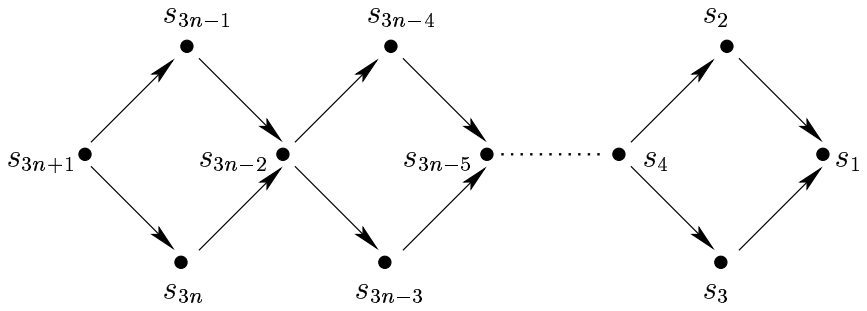


Figure 6.4: Plane partition diamonds; constraint graph G_n

true for all $n < k$. Consider the case $n = k$. We have from (6.1),

$$\begin{aligned} H_{G_k}(q,r) &= \frac{H_{G_{k-1}}(q,qr)}{1-r} \\ &= \frac{1}{1-r}\prod_{i=1}^{k-1}\frac{1}{1-q^{i-1}qr} \\ &= \frac{1}{1-r}\prod_{i=2}^{k}\frac{1}{1-q^{i-1}r} \\ &= \prod_{i=1}^{k}\frac{1}{1-q^{i-1}r} \end{aligned}$$

which proves the theorem. ■

Observe that replacing r by q in (6.2) gives us

$$H_{G_n}(q,q) = \prod_{i=1}^{n}\frac{1}{1-q^i}$$

which is the same as the single-variable generating function (2.4) that was intuitive in Section 2.3.

6.2 Plane partition diamonds

Plane partition diamonds (Figure 6.4) were introduced by Andrews, Paule and Riese in [9], wherein they obtain the generating function for the family using MacMahon's Partition Analysis and the $\underset{\geq}{\Omega}$ operator (see Section 3.1). The simplest case of the plane partition diamonds, G_1, was first considered and solved by MacMahon in [27]. An alternative proof using combinatorial techniques was presented by Corteel and Savage in [19].

We approach the plane partition diamonds problem using the constraint graph technique. We begin the decomposition of graph G_n of Figure 6.4 by applying the inclusion-exclusion operator on

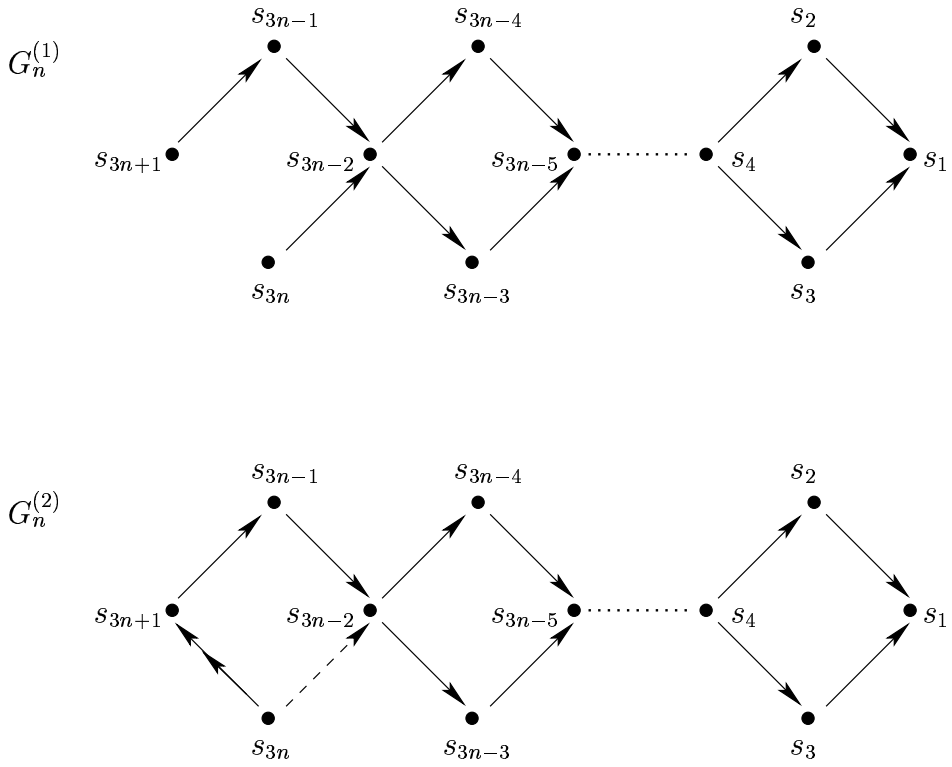


Figure 6.5: Constraint graphs $G_n^{(1)}$ and $G_n^{(2)}$ for plane partition diamonds

edge (s_{3n+1}, s_{3n}) of G_n, as shown in Figure 6.5. Observe in $G_n^{(2)}$ that edge (s_{3n}, s_{3n-2}) is redundant. The entire decomposition can be represented as

$$\begin{aligned}
G_n &= -((s_{3n+1}, s_{3n}), 0, G_n^{(1)}, G_n^{(2)}) \\
G_n^{(1)} &= I((s_{3n}, s_{3n-2}), 0, G_n^{(3)}) \\
G_n^{(3)} &= I((s_{3n+1}, s_{3n-1}), 0, G_n^{(4)}) \\
G_n^{(4)} &= I((s_{3n-1}, s_{3n-2}), 0, G_n^{(5)}) \\
G_n^{(5)} &= G_{n-1} \\
G_n^{(2)} &= R((s_{3n}, s_{3n-2}), 0, G_n^{(6)}) \\
G_n^{(6)} &= I((s_{3n}, s_{3n+1}), 1, G_n^{(7)}) \\
G_n^{(7)} &= I((s_{3n+1}, s_{3n-1}), 0, G_n^{(8)}) \\
G_n^{(8)} &= I((s_{3n-1}, s_{3n-2}), 0, G_n^{(9)}) \\
G_n^{(9)} &= G_{n-1}
\end{aligned}$$

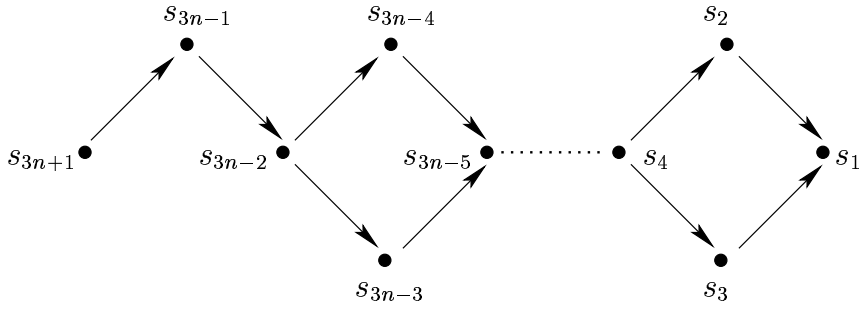


Figure 6.6: Constraint graph for $G_n^{(3)}$ and $G_n^{(7)}$ for plane partition diamonds

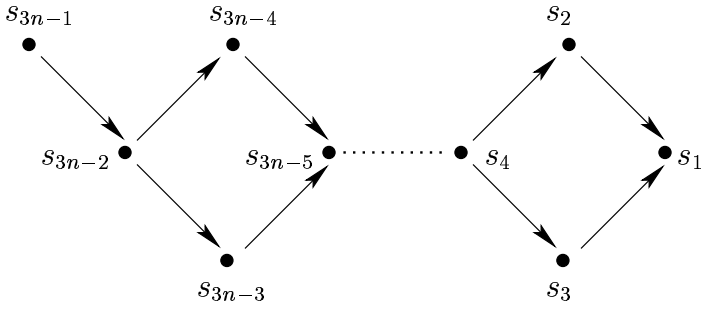


Figure 6.7: Constraint graph for $G_n^{(3)}$ and $G_n^{(7)}$ for plane partition diamonds

where $G_n^{(3)}$ and $G_n^{(7)}$ can be represented by Figure 6.6, $G_n^{(4)}$ and $G_n^{(8)}$ can be represented by Figure 6.7 and $G_n^{(5)}$ and $G_n^{(9)}$ can be represented by Figure 6.8. The construction tree corresponding to this decomposition is shown in Figure 6.9. Constraint graph G_0, the base case, is simply a single-vertex graph with vertex s_{3n+1}.

Assuming that the finite variable recurrence needs no more than one special variable, we invoke procedure Recurrence of the `GFPartitions` package as

```
GFPartitions[Recurrence]([
    ["-", "i", x[3*n], x[3*n-2], "i", x[3*n+1], x[3*n-1],
                "i",x[3*n-1], x[3*n-2], "%",
          "i*", x[3*n], x[3*n+1], "i", x[3*n+1], x[3*n-1],
                "i",x[3*n-1], x[3*n-2], "%"],
    n=0, [x[3*n+1]]],
    [s])
```

The output generated is

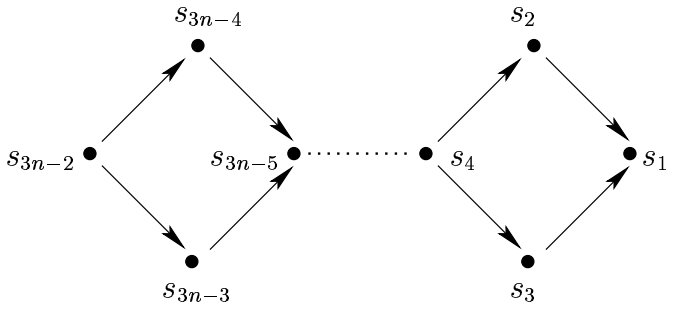


Figure 6.8: Constraint graph G_{n-1} for plane partition diamonds

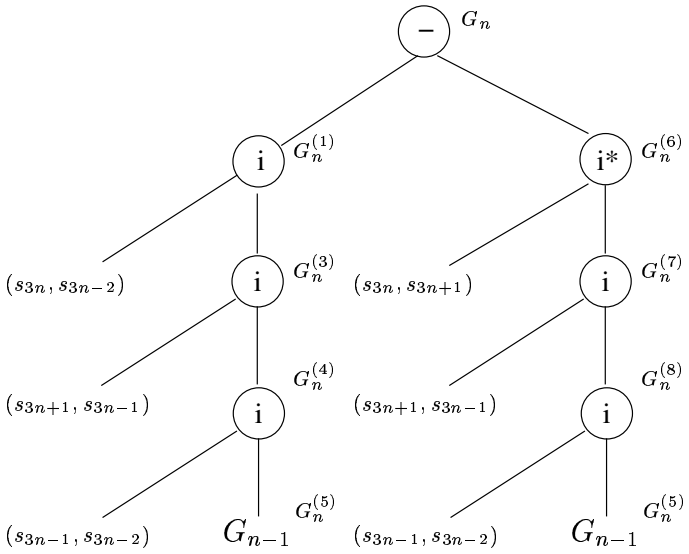


Figure 6.9: Construction tree for G_n for plane partition diamonds

---

```
========================  Multi-variable recurrence  ========================

Case "default"
```

$$\frac{R_{11}}{(1-x_{3\,n-1}x_{3\,n+1})\,(1-x_{3\,n+1})\,(1-x_{3\,n})} - \frac{R_{21}x_{3\,n}}{(1-x_{3\,n-1}x_{3\,n+1}x_{3\,n})\,(1-x_{3\,n+1}x_{3\,n})\,(1-x_{3\,n})}$$

```
            R[21] -- F(n-1)
                                x[3*n-2] <--- x[3*n-2]*x[3*n]*x[3*n-1]*x[3*n+1]
            R[11] -- F(n-1)
                                x[3*n-2] <--- x[3*n-2]*x[3*n]*x[3*n-1]*x[3*n+1]
Case n = 0
```

$$\frac{1}{(1-x_{3\,n+1})}$$

```
========================  Finite-variable recurrence  ========================
            x[3*n+1] --> s
```

$$H\,(n,q,s)$$

```
if n = 0 then
```

$$\frac{1}{(1-s)}$$

```
else
```

$$\frac{H\,(n-1,q,q^3s)}{(1-qs)\,(1-s)\,(1-q)} - \frac{H\,(n-1,q,q^3s)\,q}{(1-q^2s)\,(1-qs)\,(1-q)}$$

```
fi;
```

Thus, the multi-variable recurrence is obtained as

$$F_{G_n}(x_1,\ldots,x_{3n-1},x_{3n},x_{3n+1}) = \begin{cases} \frac{1}{(1-x_{3n+1})} & n=0 \\ \frac{F_{G_{n-1}}(x_1,\ldots,x_{3n-3},x_{3n-2}x_{3n-1}x_{3n}x_{3n+1})}{(1-x3n-1x3n+1)(1-x3n+1)(1-x3n)} - \frac{F_{G_{n-1}}(x_1,\ldots,x_{3n-3},x_{3n-2}x_{3n-1}x_{3n}x_{3n+1})}{(1-x3n-1x3n+1x3n)(1-x3n+1x_{3n})(1-x3n)} & otherwise \end{cases}$$

and the finite-variable recurrence is

$$H_{G_n}(q,s) = \begin{cases} \frac{1}{(1-s)} & n=0 \\ \frac{H_{G_{n-1}}(q,q^3s)(1+qs)}{(1-q^2s)(1-qs)(1-s)} & otherwise. \end{cases} \tag{6.3}$$

where $H_{G_n}(q,s) = F_{G_n}(q,q,\ldots,q,s)$.

This finite variable recurrence for plane partition diamonds was solved in [33]. We repeat the process here in order to reiterate the significance of the recurrence and the simplicity of the approach. Proving the following theorem will enable us to obtain the closed form generating function for the problem.

Theorem 8 *The solution to (6.3) is*

$$R_{G_n}(q,s) = \frac{1}{(1-s)} \prod_{i=1}^{n} \frac{(1+q^{3i-2}s)}{(1-q^{3i}s)(1-q^{3i-1}s)(1-q^{3i-2}s)}$$

Proof. We prove this by induction over n. For $n = 0$, $H_{G_n}(q,s) = \frac{1}{(1-s)} = R_{G_n}(q,s)$. Assume $H_{G_k}(q,s) = R_{G_k}(q,s)$ for every $k < n$. We now prove that $H_{G_k}(q,s) = R_{G_k}(q,s)$ for $k = n$.

From the recurrence 6.3, we have,

$$H_{G_n}(q,s) = \frac{H_{G_{n-1}}(q,q^3s)(1+qs)}{(1-q^2s)(1-qs)(1-s)}$$

Since we assume that $H_{G_k}(q,s) = R_{G_k}(q,s)$ for every $k < n$, it must be true for $k = n-1$. Hence,

$$\begin{aligned} H_{G_n}(q,s) &= \frac{(1+qs)}{(1-q^2s)(1-qs)(1-s)} \left[\frac{1}{(1-q^3s)} \prod_{i=1}^{n-1} \frac{(1+q^{3i-2}q^3s)}{(1-q^{3i}q^3s)(1-q^{3i-1}q^3s)(1-q^{3i-2}q^3s)} \right] \\ &= \frac{1}{(1-s)} \left[\prod_{i=1}^{n} \frac{(1+q^{3i-2}s)}{(1-q^{3i}s)(1-q^{3i-1}s)(1-q^{3i-2}s)} \right] \\ &= R_{G_n}(q,s). \end{aligned}$$

■

Now consider

$$\begin{aligned} R_{G_n}(q,q) &= \frac{1}{(1-q)} \prod_{i=1}^{n} \frac{(1+q^{3i-2}q)}{(1-q^{3i}q)(1-q^{3i-1}q)(1-q^{3i-2}q)} \\ &= \frac{1}{(1-q)} \frac{\prod_{i=1}^{n}(1+q^{3i-1})}{\prod_{i=1}^{n}(1-q^{3i+1})(1-q^{3i})(1-q^{3i-1})} \\ &= \frac{\prod_{i=1}^{n}(1+q^{3i-1})}{\prod_{i=1}^{3n+1}(1-q^i)}. \end{aligned}$$

This is the same as the generating function derived by Andrews, Paule and Riese in [9].

6.3 Up-down sequences

In this section, we consider the up-down sequences [14] of Figure 6.10 and use the Seven Rules and `GFPartitions` package to find recurrences for its generating function.

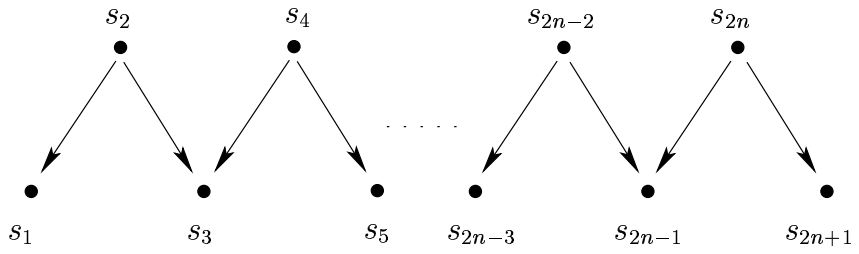


Figure 6.10: Up-down sequences; constraint graph G_n

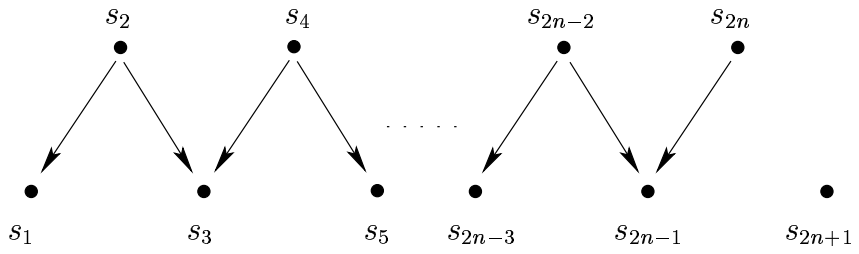


Figure 6.11: Constraint graph $G_n^{(1)}$ for up-down sequences

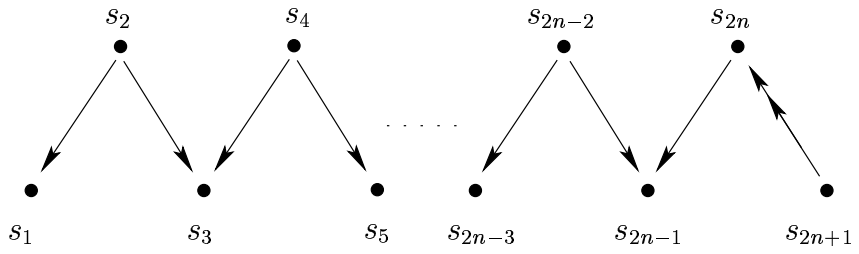


Figure 6.12: Constraint graph $G_n^{(2)}$ for up-down sequences

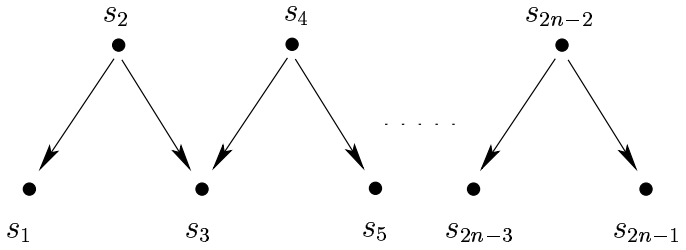


Figure 6.13: Constraint graph G_{n-1} for up-down sequences

Decomposition of up-down sequence G_n can proceed as follows. First, we handle the outgoing edge (s_{2n+1}, s_{2n}) by applying the inclusion-exclusion operation on G_n to obtain the graphs $G_n^{(1)}$ (Figure 6.11) and $G_n^{(2)}$ (Figure 6.12). Thereafter, simple applications of Rule 3 (single vertex graph), Rule 5 (incoming edge) and finally Rule 7 (graph instance) complete the decomposition process of G_n. Specifically, the decomposition is

$$\begin{aligned}
G_n &= -((s_{2n}, s_{2n+1}), 0, G_n^{(1)}, G_n^{(2)}) \\
G_n^{(1)} &= V((s_{2n+1}), G_n^{(3)}) \\
G_n^{(3)} &= I((s_{2n}, s_{2n-1}), 0, G_n^{(4)}) \\
G_n^{(4)} &= G_{n-1} \\
G_n^{(2)} &= I((s_{2n+1}, s_{2n}), 1, G_n^{(5)}) \\
G_n^{(5)} &= I((s_{2n}, s_{2n-1}), 0, G_n^{(6)}) \\
G_n^{(6)} &= G_{n-1}
\end{aligned}$$

where G_{n-1} is the graph instance of Figure 6.13. The base case G_0 is the graph with the single node s_{2n+1}.

Procedure Recurrence can now be invoked as

```
GFPartitions[Recurrence]([
    ["-", ".", x[2*n+1], "i", x[2*n], x[2*n-1], "%",
         "i*", x[2*n+1], x[2*n], "i", x[2*n], x[2*n-1], "%"],
    n=0,
    [x[2*n+1]]   ],
    [r])
```

and the output generated is

```
========================  Multi-variable recurrence  ========================

Case "default"
```

$$\frac{R_7}{(1-x_{2\,n})\,(1-x_{2\,n+1})} - \frac{R_{14}x_{2\,n+1}}{(1-x_{2\,n}x_{2\,n+1})\,(1-x_{2\,n+1})}$$

```
            R[14] -- F(n-1)

                                x[2*n-1] <--- x[2*n-1]*x[2*n]*x[2*n+1]

            R[7] -- F(n-1)

                                x[2*n-1] <--- x[2*n-1]*x[2*n]

Case n = 0
```

$$\frac{1}{(1-x_{2\,n+1})}$$

```
========================  Finite-variable recurrence  ========================

            x[2*n+1] --> r
```

$$H\,(n,q,r)$$

```
if n = 0 then
```

$$\frac{1}{(1-r)}$$

```
else
```

$$\frac{H\,\left(n-1,q,q^2\right)}{(1-q)\,(1-r)} - \frac{H\,\left(n-1,q,q^2r\right)r}{(1-qr)\,(1-r)}$$

```
fi;
```

6.4 2-rowed plane partitions

The 2-rowed plane partitions of Figure 6.14 were first considered by MacMahon in [28]. In [4], Andrews revisits 2-rowed plane partitions and obtains the generating function using MacMahon's Partition Analysis. Corteel, Lee and Savage approach the same problem in [18] using the Five Guidelines technique. We now demonstrate the ease with which the Seven Rules in conjunction with the `GFPartitions` package can handle this problem.

We described the entire process of obtaining a recurrence for 2-rowed plane partitions spread out across sections 4.3.3, 4.3.4, 4.4.2, 5.2 and 5.3 of this thesis. We collate them here for convenience.

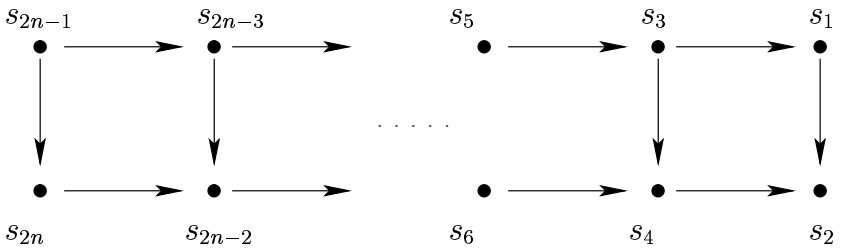


Figure 6.14: 2-rowed plane partitions; constraint graph G_n

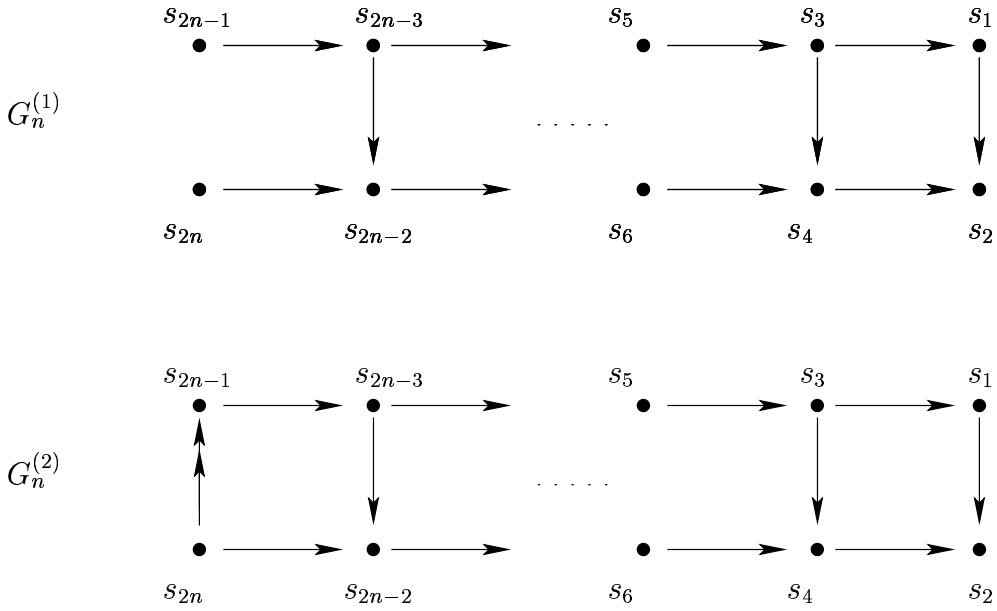


Figure 6.15: Graphs $G_n^{(1)}$ and $G_n^{(2)}$ for 2-rowed plane partitions

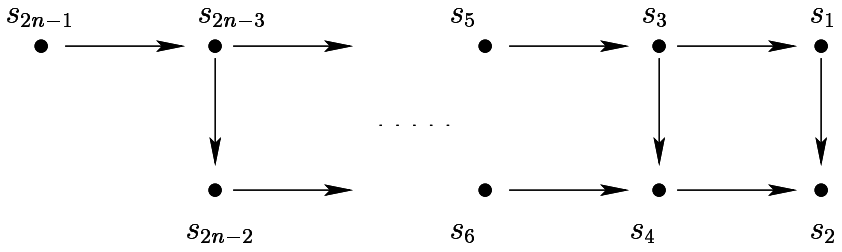


Figure 6.16: Graphs $G_n^{(3a)}$ and $G_n^{(3b)}$ for 2-rowed plane partitions

The base case for 2-rowed plane partitions is simply the edge (s_1, s_2) which can be decomposed easily by

$$\begin{aligned} G_1 &= I((s_{2n-1}, s_{2n}), 0, G_1^{(1)}) \\ G_1^{(1)} &= s_{2n}. \end{aligned}$$

The general case G_n can be decomposed by first applying the inclusion-exclusion operator on (s_{2n-1}, s_{2n}) to obtain $G_n^{(1)}$ and $G_n^{(2)}$ of Figure 6.15. Observe that (s_{2n}, s_{2n-2}) in $G_n^{(2)}$ is redundant. The rest is straightforward, and the entire decomposition of G_n into G_{n-1} can be represented as

$$\begin{aligned} G_n &= -((s_{2n-1}, s_{2n}), 0, G_n^{(1)}, G_n^{(2)}) \\ G_n^{(1)} &= I((s_{2n}, s_{2n-2}), 0, G_n^{(3a)}) \\ G_n^{(3a)} &= I((s_{2n-1}, s_{2n-3}), 0, G_n^{(5a)}) \\ G_n^{(5a)} &= G_{n-1} \\ G_n^{(2)} &= R((s_{2n}, s_{2n-2}), 0, G_n^{(4)}) \\ G_n^{(4)} &= I((s_{2n}, s_{2n-1}), 1, G_n^{(3b)}) \\ G_n^{(3b)} &= I((s_{2n-1}, s_{2n-3}), 0, G_n^{(5b)}) \\ G_n^{(5b)} &= G_{n-1}. \end{aligned} \tag{6.4}$$

where $G_n^{(3a)}$ and $G_n^{(3b)}$ are represented by Figure 6.16, $G_n^{(4)}$ by Figure 6.17, and $G_n^{(5a)}$ and $G_n^{(5b)}$ by Figure 6.18. The construction tree for the general case is shown in Figure 6.19.

We invoke procedure `Recurrence` as

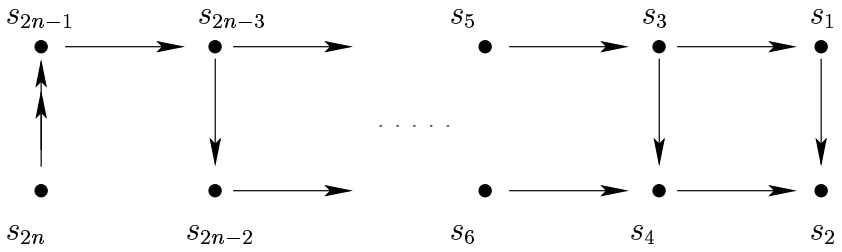


Figure 6.17: Graph $G_n^{(4)}$ for 2-rowed plane partitions

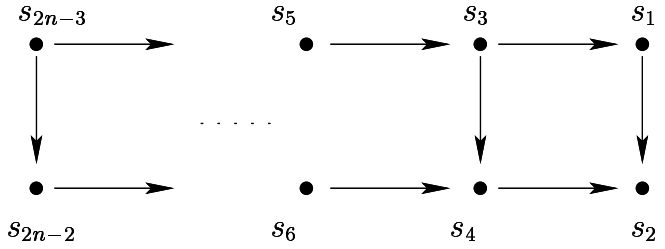


Figure 6.18: Graph G_{n-1} for 2-rowed plane partitions

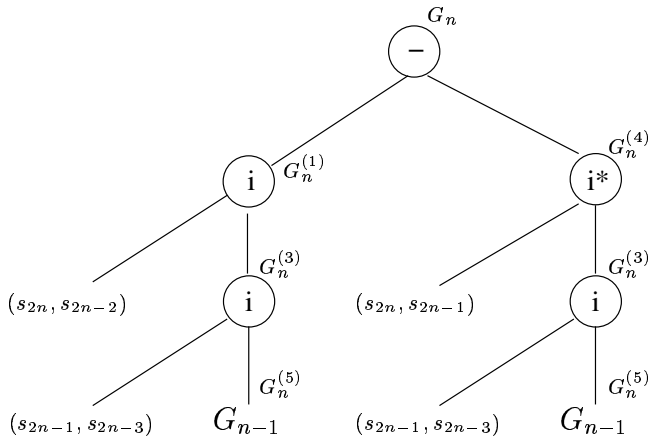


Figure 6.19: Construction tree for the decomposition of G_n for 2-rowed plane partitions

```
GFPartitions[Recurrence]([
    ["-", "i", x[2*n], x[2*n-2], "i", x[2*n-1], x[2*n-3], "%",
         "i*", x[2*n], x[2*n-1], "i", x[2*n-1], x[2*n-3], "%"],
    n=1,
    ["i", x[2*n-1], x[2*n], x[2*n]]   ],
    [s, t])
```

and the output generated is

```
========================  Multi-variable recurrence  ========================
```

```
Case "default"
```

$$\frac{R_8}{(1-x_{2\,n-1})\,(1-x_{2\,n})}-\frac{R_{15}x_{2\,n}}{(1-x_{2\,n-1}x_{2\,n})\,(1-x_{2\,n})}$$

```
    R[15] -- F(n-1)
                        x[2*n-3] <--- x[2*n-3]*x[2*n-1]*x[2*n]
    R[8] -- F(n-1)
                        x[2*n-2] <--- x[2*n-2]*x[2*n]
                        x[2*n-3] <--- x[2*n-3]*x[2*n-1]
Case n = 1
```

$$\frac{1}{(1-x_{2\,n-1}x_{2\,n})\,(1-x_{2\,n-1})}$$

```
========================  Finite-variable recurrence  ========================
    x[2*n-1] --> s
    x[2*n] --> t
```

$$H(n,q,s,t)$$

```
if n = 1 then
```

$$\frac{1}{(1-st)\,(1-s)}$$

```
else
```

$$\frac{H\,(n-1,q,qs,qt)}{(1-s)\,(1-t)}-\frac{H\,(n-1,q,qst,q)\,t}{(1-st)\,(1-t)}$$

```
fi;
```

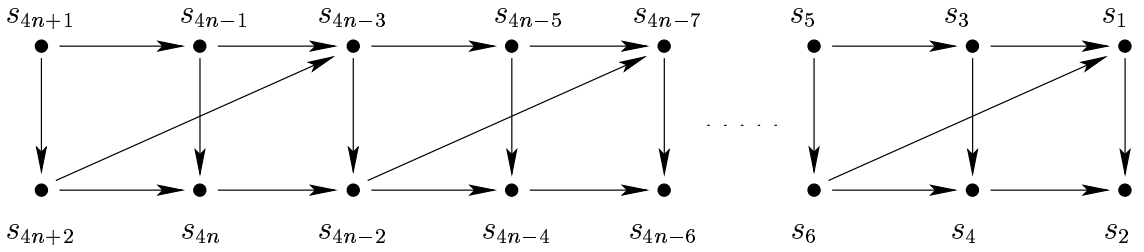


Figure 6.20: Constraint graph G_n for 2-rowed plane partitions with diagonals

This recurrence can be solved to obtain the same generating function obtained in [18].

6.5 2-rowed plane partitions with diagonals

We consider here a variation of 2-rowed plane partitions. In [1], Andrews, Paule and Riese introduce 2-rowed plane partitions with diagonals (Figure 6.20) and user MacMahon's Partition Analysis to obtain a closed form of the single-variable generating function. We demonstrate application of the Seven Rules and the `GFPartitions` package on the same problem.

Constraint graph G_n can be decomposed as follows.

$$
\begin{aligned}
G_n &= -((s_{4n+1}, s_{4n+2}), 0, G_n^{(1)}, G_n^{(2)}) \\
G_n^{(1)} &= I((s_{4n+1}, s_{4n-1}), 0, G_n^{(3)}) \\
G_n^{(3)} &= -((s_{4n+2}, s_{4n}), 0, G_n^{(4)}, G_n^{(5)}) \\
G_n^{(4)} &= I((s_{4n+2}, s_{4n-3}), 0, G_n^{(6)}) \\
G_n^{(6)} &= -((s_{4n-1}, s_{4n}), 0, G_n^{(7)}, G_n^{(8)}) \\
G_n^{(7)} &= I((s_{4n-1}, s_{4n-3}), 0, G_n^{(9)}) \\
G_n^{(9)} &= I((s_{4n}, s_{4n-2}), 0, G_n^{(10)}) \\
G_n^{(10)} &= G_{n-1} \\
G_n^{(8)} &= R((s_{4n+2}, s_{4n}), 0, G_n^{(11)}) \\
G_n^{(11)} &= I((s_{4n}, s_{4n-1}), 1, G_n^{(12)}) \\
G_n^{(12)} &= I((s_{4n-1}, s_{4n-3}), 0, G_n^{(13)}) \\
G_n^{(13)} &= G_{n-1}
\end{aligned}
$$

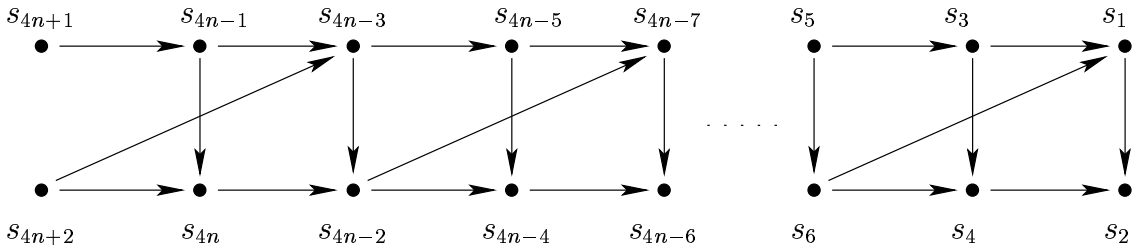


Figure 6.21: Constraint graph $G_n^{(1)}$ for 2-rowed plane partitions with diagonals

$$
\begin{aligned}
G_n^{(5)} &= R((s_{4n-1}, s_{4n-3}), 0, G_n^{(14)}) \\
G_n^{(14)} &= R((s_{4n}, s_{4n-2}), 0, G_n^{(15)}) \\
G_n^{(15)} &= I((s_{4n-1}, s_{4n}), 0, G_n^{(16)}) \\
G_n^{(16)} &= I((s_{4n}, s_{4n+2}), 1, G_n^{(17)}) \\
G_n^{(17)} &= I((s_{4n+2}, s_{4n-3}), 0, G_n^{(18)}) \\
G_n^{(18)} &= G_{n-1} \\
G_n^{(2)} &= I((s_{4n+2}, s_{4n+1}), 1, G_n^{(19)}) \\
G_n^{(19)} &= I((s_{4n+1}, s_{4n-1}), 0, G_n^{(20)}) \\
G_n^{(20)} &= -((s_{4n-1}, s_{4n}), 0, G_n^{(21)}, G_n^{(22)}) \\
G_n^{(21)} &= I((s_{4n-1}, s_{4n-3}), 0, G_n^{(23)}) \\
G_n^{(23)} &= I((s_{4n}, s_{4n-2}), 0, G_n^{(24)}) \\
G_n^{(24)} &= G_{n-1} \\
G_n^{(22)} &= R((s_{4n+2}, s_{4n}), 0, G_n^{(25)}) \\
G_n^{(25)} &= I((s_{4n}, s_{4n-1}), 1, G_n^{(26)}) \\
G_n^{(26)} &= I((s_{4n-1}, s_{4n-3}), 0, G_n^{(27)}) \\
G_n^{(27)} &= G_{n-1}
\end{aligned}
$$

where $G_n^{(1)}$ is represented by Figure 6.21, $G_n^{(2)}$ by Figure 6.22, $G_n^{(4)}$ by Figure 6.23, $G_n^{(5)}$ by Figure 6.24, $G_n^{(7)}$ and $G_n^{(21)}$ by Figure 6.25, $G_n^{(8)}$ and $G_n^{(22)}$ by Figure 6.26 and G_{n-1} by Figure 6.27. The construction tree for the general case is shown in Figure 6.28. The base case G_0 is decomposed simply as

$$
\begin{aligned}
G_0 &= I((s_{4n+1}, s_{4n+2}), 0, G_0^{(1)}) \\
G_0^{(1)} &= s_{4n+2}.
\end{aligned}
$$

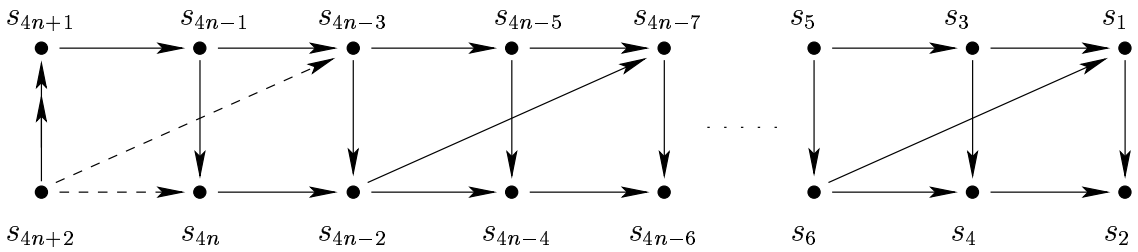


Figure 6.22: Constraint graph $G_n^{(2)}$ for 2-rowed plane partitions with diagonals

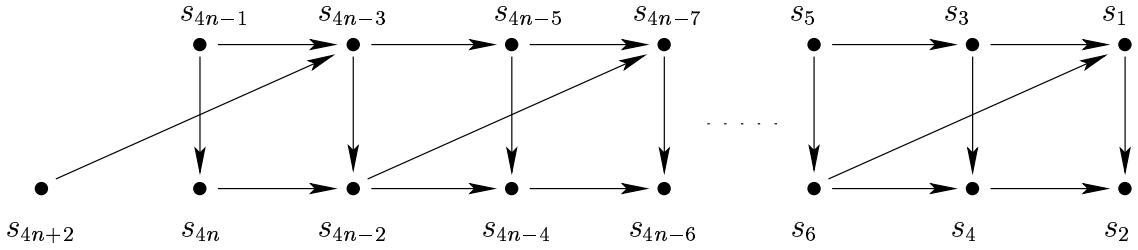


Figure 6.23: Constraint graph $G_n^{(4)}$ for 2-rowed plane partitions with diagonals

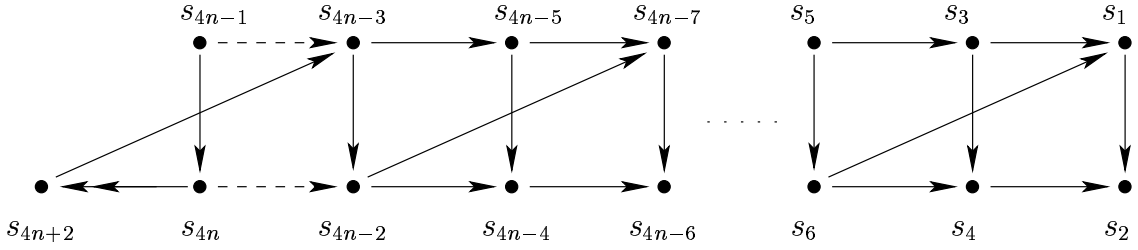


Figure 6.24: Constraint graph $G_n^{(5)}$ for 2-rowed plane partitions with diagonals

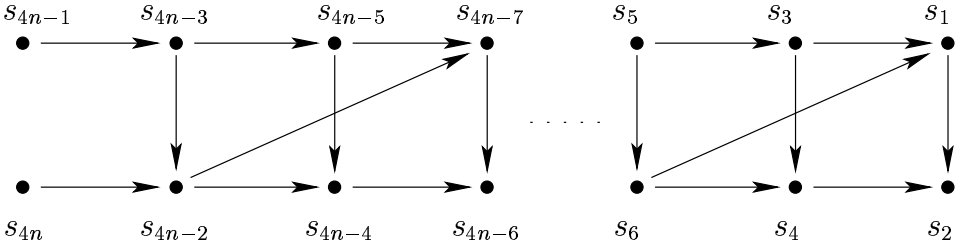


Figure 6.25: Constraint graphs $G_n^{(7)}$ and $G_n^{(21)}$ for 2-rowed plane partitions with diagonals

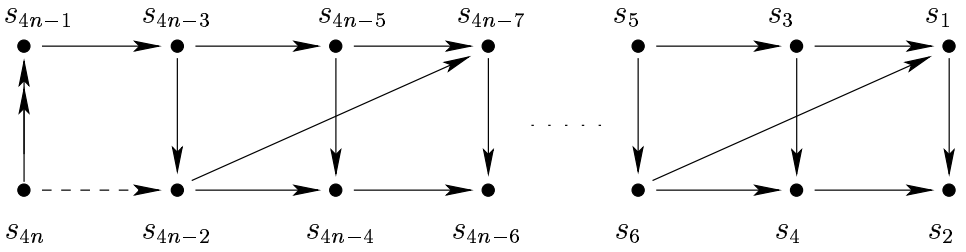


Figure 6.26: Constraint graphs $G_n^{(8)}$ and $G_n^{(22)}$ for 2-rowed plane partitions with diagonals

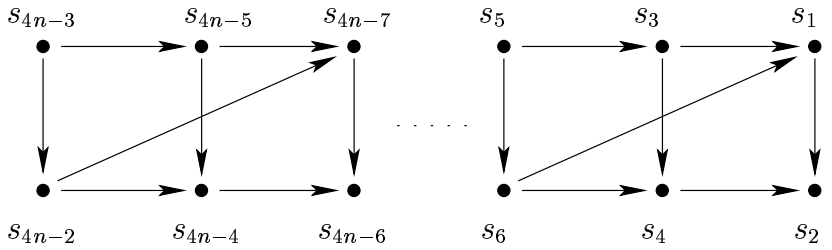


Figure 6.27: Constraint graph G_{n-1} for 2-rowed plane partitions with diagonals

We invoke procedure `Recurrence` as

```
GFPartitions[Recurrence]([
    ["-",
        "i", x[4*n+1], x[4*n-1],
        "-",
            x[4*n+2], x[4*n], "i", x[4*n+2], x[4*n-3], "-",
                "i", x[4*n-1], x[4*n-3], "i", x[4*n], x[4*n-2], "%",
                "i*", x[4*n], x[4*n-1], "i", x[4*n-1], x[4*n-3]), "%",
            "i", x[4*n-1], x[4*n]), "i*", x[4*n], x[4*n+2],
                "i", x[4*n+2], x[4*n-3], "%",
         "i*", x[4*n+2], x[4*n+1]), "i", x[4*n+1], x[4*n-1]), "-",
            "i", x[4*n-1], x[4*n-3]), "i", x[4*n], x[4*n-2]), "%",
            "i*", x[4*n], x[4*n-1]), "i", x[4*n-1], x[4*n-3]), "%"],
    n=0, ["i", x[4*n+1], x[4*n+2], x[4*n+2]]   ],
    [s, t])
```

and the output generated is

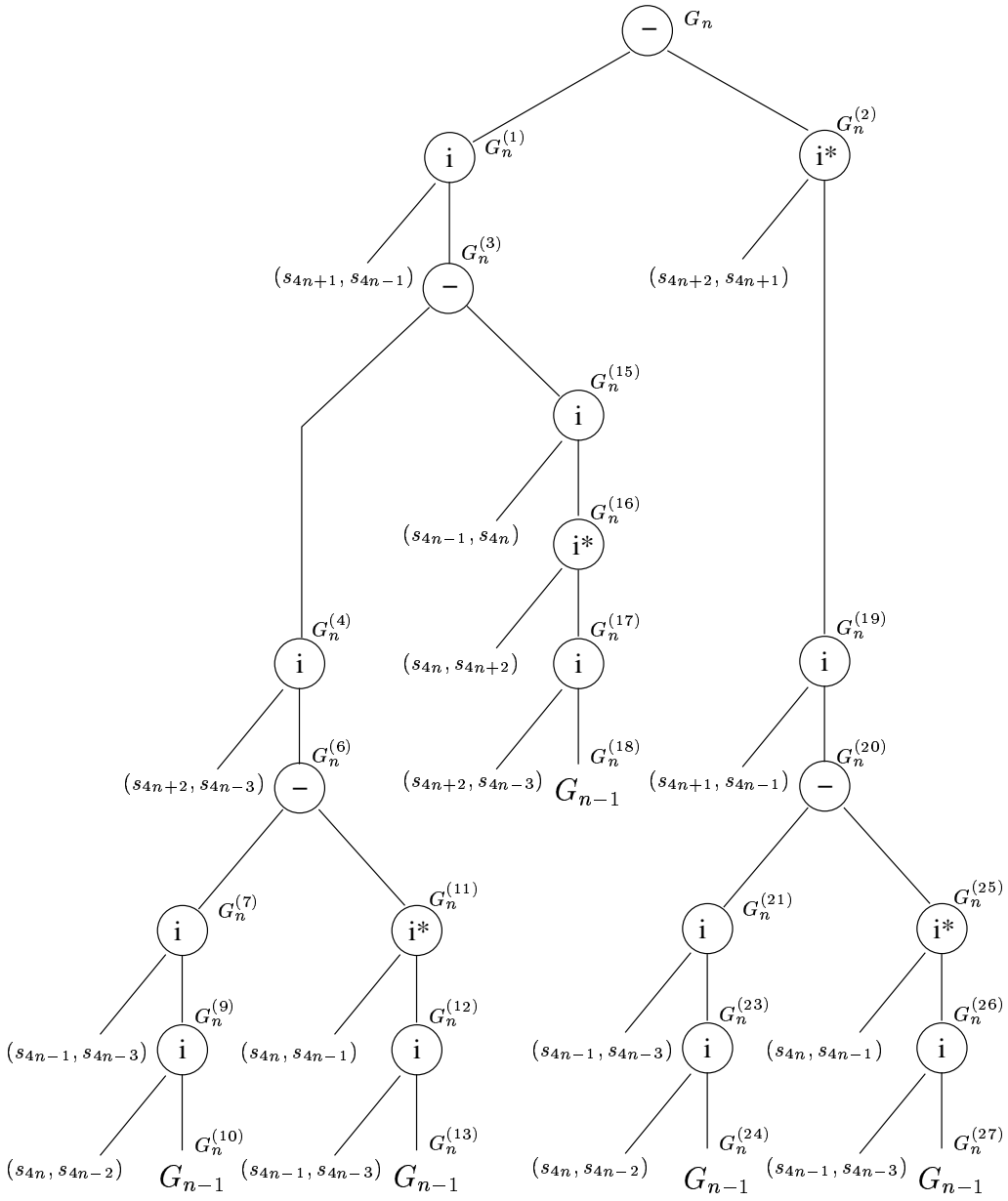


Figure 6.28: Construction tree for G_n of 2-rowed plane partitions with diagonals

---

```
========================  Multi-variable recurrence  ========================
```

```
Case "default"
```

$$\left(\frac{R_{16}}{(1-x_{4\,n})\,(1-x_{4\,n-1}x_{4\,n+1})}-\frac{x_{4\,n}R_{23}}{(1-x_{4\,n-1}x_{4\,n+1}x_{4\,n})\,(1-x_{4\,n})}\right)\frac{1}{(1-x_{4\,n+2})\,(1-x_{4\,n+1})}$$
$$-\frac{x_{4\,n-1}x_{4\,n+1}x_{4\,n}R_{33}}{(1-x_{4\,n+2}x_{4\,n-1}x_{4\,n+1}x_{4\,n})\,(1-x_{4\,n-1}x_{4\,n+1}x_{4\,n})\,(1-x_{4\,n-1}x_{4\,n+1})}\frac{1}{(1-x_{4\,n+1})}$$
$$-\frac{\left(\frac{R_{47}}{(1-x_{4\,n})(1-x_{4\,n-1}x_{4\,n+1}x_{4\,n+2})}-\frac{R_{54}x_{4\,n}}{(1-x_{4\,n+2}x_{4\,n-1}x_{4\,n+1}x_{4\,n})(1-x_{4\,n})}\right)x_{4\,n+2}}{(1-x_{4\,n+1}x_{4\,n+2})\,(1-x_{4\,n+2})}$$

```
        R[47] -- F(n-1)
                x[4*n-2] <--- x[4*n-2]*x[4*n]
                x[4*n-3] <--- x[4*n-3]*x[4*n+2]*x[4*n-1]*x[4*n+1]
        R[23] -- F(n-1)
                x[4*n-3] <--- x[4*n-3]*x[4*n+2]*x[4*n-1]*x[4*n+1]*x[4*n]
        R[16] -- F(n-1)
                x[4*n-2] <--- x[4*n-2]*x[4*n]
                x[4*n-3] <--- x[4*n-3]*x[4*n+2]*x[4*n-1]*x[4*n+1]
        R[33] -- F(n-1)
                x[4*n-3] <--- x[4*n-3]*x[4*n+2]*x[4*n-1]*x[4*n+1]*x[4*n]
        R[54] -- F(n-1)
                x[4*n-3] <--- x[4*n-3]*x[4*n+2]*x[4*n-1]*x[4*n+1]*x[4*n]
```

```
Case n = 0
```

$$\frac{1}{(1-x_{4\,n+1}x_{4\,n+2})\,(1-x_{4\,n+1})}$$

```
========================  Finite-variable recurrence  ========================
            x[4*n+1] --> s
            x[4*n+2] --> t
```

$$H(n,q,s,t)$$

```
if n = 0 then
```

$$\frac{1}{(1-ts)(1-s)}$$

```
else
```

$$\left(\frac{\left(\frac{H\left(n-1,q,q^2ts,q^2\right)}{(1-q)(1-qs)}-\frac{H\left(n-1,q,q^3ts,q\right)q}{(1-q^2s)(1-q)}\right)}{(1-t)}-\frac{H\left(n-1,q,q^3ts,q\right)q^2s}{(1-q^2ts)(1-q^2s)(1-qs)}\right)\frac{1}{(1-s)}$$

$$-\left(\frac{H\left(n-1,q,q^2ts,q^2\right)}{(1-q)(1-qst)}-\frac{H\left(n-1,q,q^3ts,q\right)q}{(1-q^2ts)(1-q)}\right)\frac{t}{(1-ts)(1-t)}$$

```
fi;
```

It can be shown that this finite-variable recurrence leads to the closed-form generating function found in [1].

6.6 Plane partition hexagonals

Plane partition hexagonals (Figure 6.29) were first considered by Andrews, Paule and Riese in [2] wherein they obtained the generating function for the family. Here, we use the techniques developed in this thesis and the `GFPartitions` package to find a recurrence for its generating function.

The general case G_n for plane partition hexagonals is shown in Figure 6.29 and can be decomposed

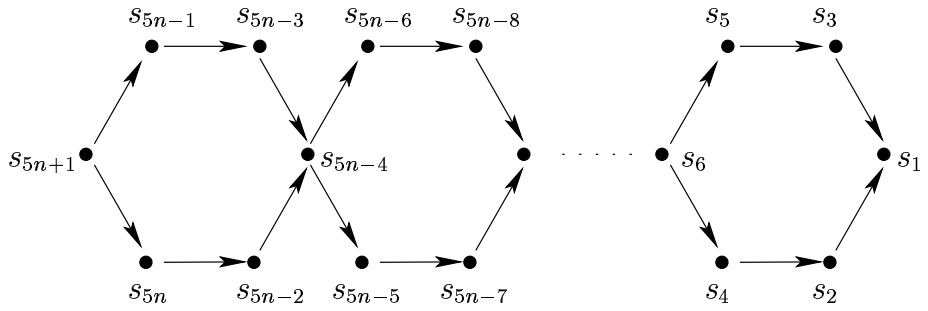


Figure 6.29: Plane partitions hexagonals; constraint graph G_n

as

$$
\begin{aligned}
G_n &= -((s_{5n+1}, s_{5n-1}), 0, G_n^{(1)}, G_n^{(2)}) \\
G_n^{(1)} &= I((s_{5n+1}, s_{5n}), 0, G_n^{(3)}) \\
G_n^{(3)} &= I((s_{5n}, s_{5n-2}), 0, G_n^{(4)}) \\
G_n^{(4)} &= I((s_{5n-2}, s_{5n-4}), 0, G_n^{(5)}) \\
G_n^{(5)} &= I((s_{5n-1}, s_{5n-3}), 0, G_n^{(6)}) \\
G_n^{(6)} &= I((s_{5n-3}, s_{5n-4}), 0, G_n^{(7)}) \\
G_n^{(7)} &= G_{n-1} \\
G_n^{(2)} &= -((s_{5n-1}, s_{5n-3}), 0, G_n^{(8)}, G_n^{(9)}) \\
G_n^{(8)} &= I((s_{5n-1}, s_{5n+1}), 1, G_n^{(10)}) \\
G_n^{(10)} &= I((s_{5n+1}, s_{5n}), 0, G_n^{(11)}) \\
G_n^{(11)} &= I((s_{5n}, s_{5n-2}), 0, G_n^{(12)}) \\
G_n^{(12)} &= I((s_{5n-2}, s_{5n-4}), 0, G_n^{(13)}) \\
G_n^{(13)} &= I((s_{5n-3}, s_{5n-4}), 0, G_n^{(14)}) \\
G_n^{(14)} &= G_{n-1} \\
G_n^{(9)} &= R((s_{5n-3}, s_{5n-4}), 0, G_n^{(15)}) \\
G_n^{(15)} &= I((s_{5n-3}, s_{5n-1}), 1, G_n^{(16)}) \\
G_n^{(16)} &= I((s_{5n-1}, s_{5n+1}), 1, G_n^{(17)}) \\
G_n^{(17)} &= I((s_{5n+1}, s_{5n}), 0, G_n^{(18)}) \\
G_n^{(18)} &= I((s_{5n}, s_{5n-2}), 0, G_n^{(19)}) \\
G_n^{(19)} &= I((s_{5n-2}, s_{5n-4}), 0, G_n^{(20)}) \\
G_n^{(20)} &= G_{n-1}
\end{aligned}
$$

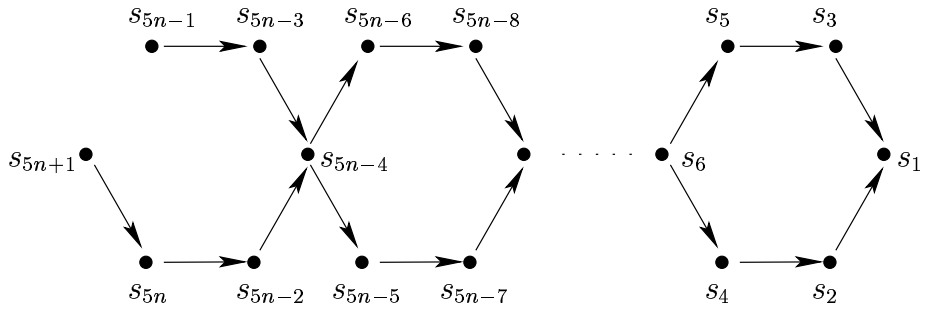


Figure 6.30: Constraint graph $G_n^{(1)}$ for plane partitions hexagonals

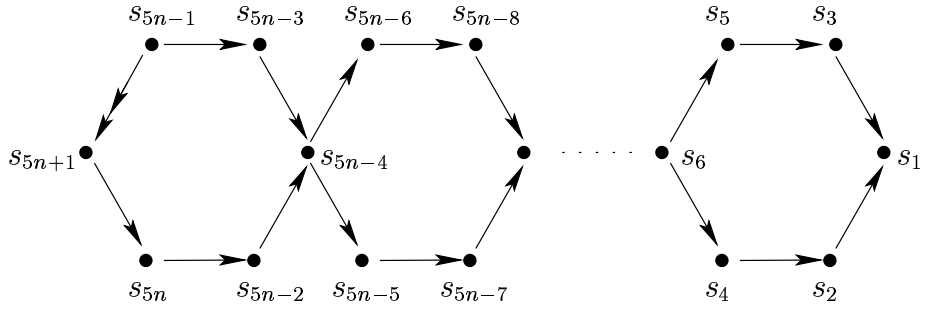


Figure 6.31: Constraint graph $G_n^{(2)}$ for plane partitions hexagonals

where $G_n^{(1)}$ is represented by Figure 6.30, $G_n^{(2)}$ by Figure 6.31, $G_n^{(8)}$ by Figure 6.32, $G_n^{(9)}$ by Figure 6.33, and G_{n-1} by Figure 6.34. We build the construction tree for the general case as in Figure 6.35. The base case G_0 is simply the vertex s_{5n+1}. It is decomposed as

$$G_0 \quad = \quad s_{5n+1}.$$

We use the `GFPartitions` package by calling

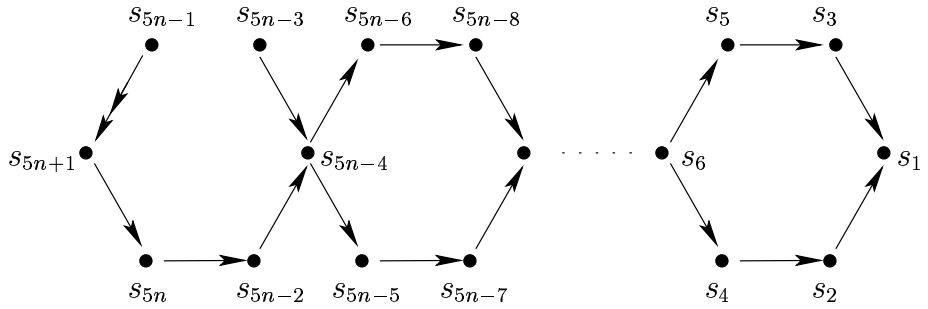


Figure 6.32: Constraint graph $G_n^{(8)}$ for plane partitions hexagonals

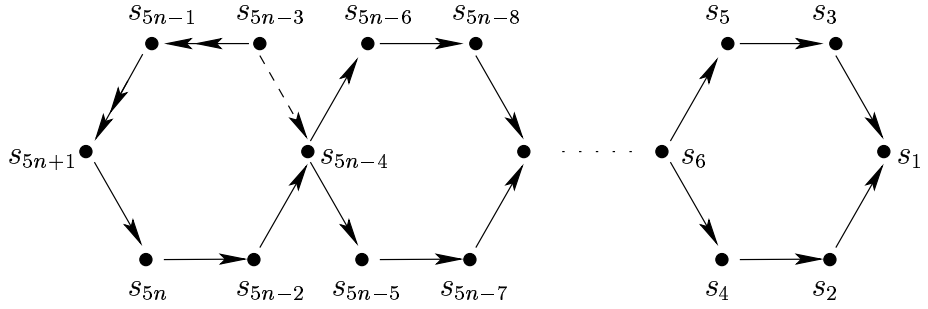


Figure 6.33: Constraint graph $G_n^{(9)}$ for plane partitions hexagonals

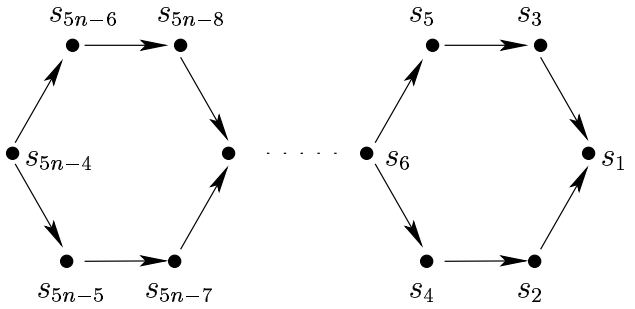


Figure 6.34: Constraint graph G_{n-1} for plane partitions hexagonals

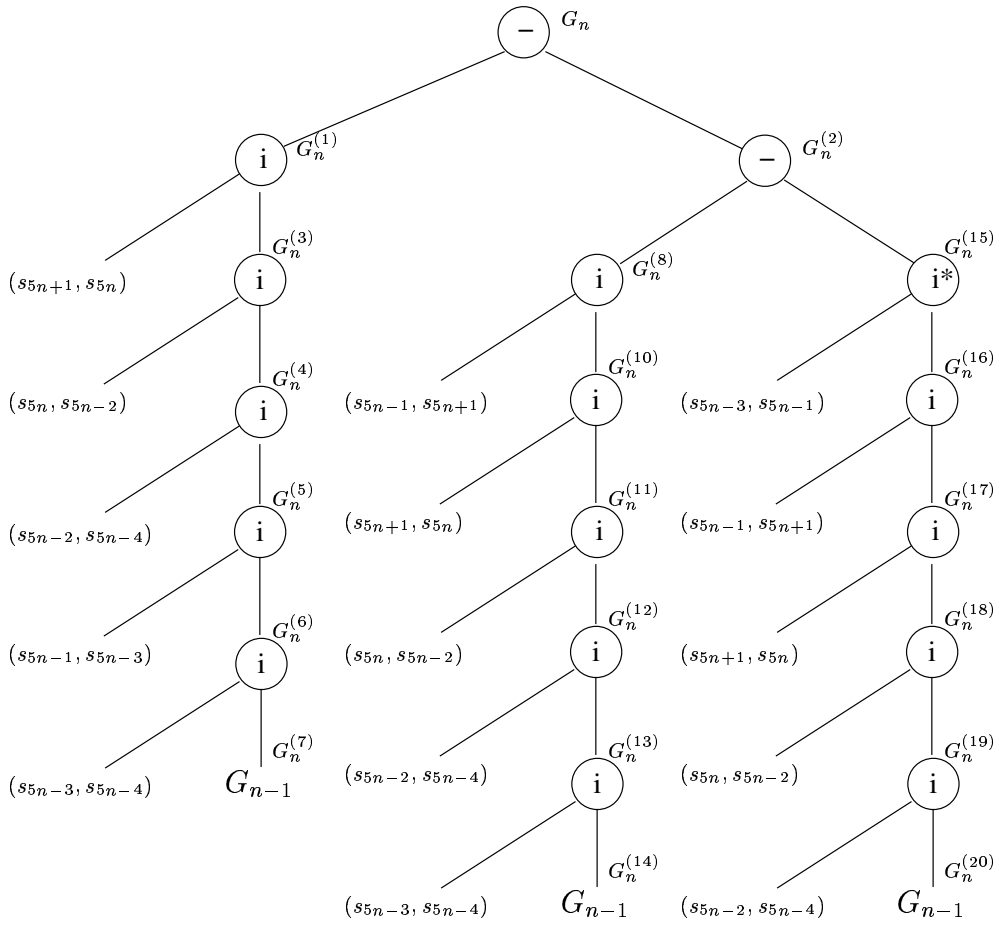


Figure 6.35: Construction tree for G_n of plane partitions hexagonals

```
GFPartitions[Recurrence]([
    ["-",
        "i", x[5*n+1], x[5*n], "i", x[5*n], x[5*n-2],
        "i", x[5*n-2], x[5*n-4], "i", x[5*n-1], x[5*n-3],
            "i", x[5*n-3], x[5*n-4], "%", "-",
                "i*", x[5*n-1], x[5*n+1], "i", x[5*n+1], x[5*n],
                    "i", x[5*n],x[5*n-2], "i", x[5*n-2], x[5*n-4],
                    "i", x[5*n-3], x[5*n-4], "%",
                "i*", x[5*n-3], x[5*n-1], "i*", x[5*n-1],x[5*n+1],
                    "i", x[5*n+1], x[5*n], "i", x[5*n], x[5*n-2],
                    "i", x[5*n-2], x[5*n-4], "%"],
    n=0, [x[5*n+1]]],
    [s])
```

and the output generated is

```
========================  Multi-variable recurrence  ========================
```

```
Case "default"
```

$$\frac{R_{17}}{(1-x_{5\,n-3}x_{5\,n-1})\,(1-x_{5\,n-1})\,(1-x_{5\,n-2}x_{5\,n}x_{5\,n+1})\,(1-x_{5\,n}x_{5\,n+1})\,(1-x_{5\,n+1})}$$

$$-\frac{R_{34}x_{5\,n-1}}{(1-x_{5\,n-3})\,(1-x_{5\,n-2}x_{5\,n}x_{5\,n+1}x_{5\,n-1})\,(1-x_{5\,n}x_{5\,n+1}x_{5\,n-1})\,(1-x_{5\,n+1}x_{5\,n-1})\,(1-x_{5\,n-1})}$$

$$+\frac{(1-x_{5\,n-2}x_{5\,n}x_{5\,n+1}x_{5\,n-3}x_{5\,n-1})^{-1}\,R_{50}x_{5\,n-3}{}^{2}x_{5\,n-1}}{(1-x_{5\,n}x_{5\,n+1}x_{5\,n-3}x_{5\,n-1})\,(1-x_{5\,n+1}x_{5\,n-3}x_{5\,n-1})\,(1-x_{5\,n-3}x_{5\,n-1})\,(1-x_{5\,n-3})}$$

```
    R[50] -- F(n-1)
            x[5*n-4] <--- x[5*n-4]*x[5*n-2]*x[5*n]*x[5*n+1]*x[5*n-3]*x[5*n-1]
    R[34] -- F(n-1)
            x[5*n-4] <--- x[5*n-4]*x[5*n-2]*x[5*n]*x[5*n+1]*x[5*n-3]*x[5*n-1]
    R[17] -- F(n-1)
            x[5*n-4] <--- x[5*n-4]*x[5*n-2]*x[5*n]*x[5*n+1]*x[5*n-3]*x[5*n-1]
Case n = 0
```

$$\frac{1}{(1-x_{5\,n+1})}$$

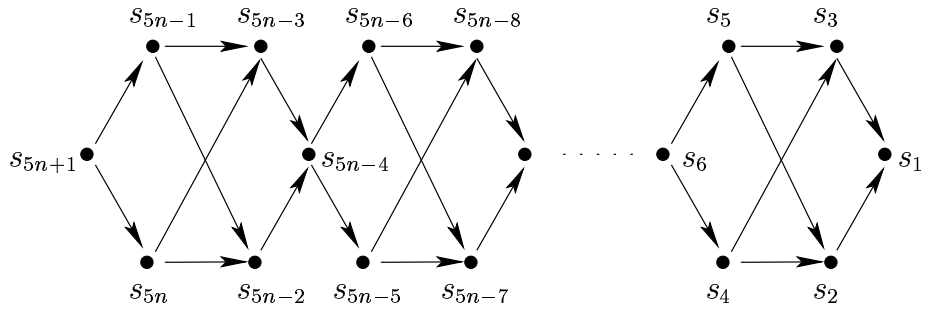


Figure 6.36: Plane partitions hexagonals with diagonals; constraint graph G_n

```
========================  Finite-variable recurrence  ======================
            x[5*n] --> s
```

$$H(n,q,s)$$

```
if n = 0 then
```

$$\frac{1}{(1-s)}$$

```
else
```

$$\frac{H\left(n-1,q,q^5s\right)}{(1-q^2)\,(1-q)\,(1-q^2s)\,(1-qs)\,(1-s)}-\frac{H\left(n-1,q,q^5s\right)q}{(1-q)^2\,(1-q^3s)\,(1-q^2s)\,(1-qs)}$$
$$+\frac{H\left(n-1,q,q^5s\right)q^3}{(1-q^4s)\,(1-q^3s)\,(1-q^2s)\,(1-q^2)\,(1-q)}$$

```
fi;
```

The above recurrence can be shown to correspond to the generating function obtained in [2].

6.7 Plane partition hexagonals with diagonals

Andrews, Paule and Riese observed in [2] that when diagonals were added to plane partition hexagonals, they factored better. We find a recurrence for these partitions using the Seven Rules and the `GFPartitions` package.

The base case G_0 is simply the vertex s_{5n+1}. The general case G_n for plane partition hexagonals

with diagonals is shown in Figure 6.36 and can be decomposed as

$$
\begin{aligned}
G_n &= -((s_{5n+1}, s_{5n}), 0, G_n^{(1)}, G_n^{(2)}) \\
G_n^{(1)} &= I((s_{5n+1}, s_{5n-1}), 0, G_n^{(3)}) \\
G_n^{(3)} &= -((s_{5n-1}, s_{5n-2}), 0, G_n^{(4)}, G_n^{(5)}) \\
G_n^{(4)} &= I((s_{5n-1}, s_{5n-3}), 0, G_n^{(6)}) \\
G_n^{(6)} &= -((s_{5n}, s_{5n-3}), 0, G_n^{(7)}, G_n^{(8)}) \\
G_n^{(7)} &= I((s_{5n-3}, s_{5n-4}), 0, G_n^{(9)}) \\
G_n^{(9)} &= I((s_{5n}, s_{5n-2}), 0, G_n^{(10)}) \\
G_n^{(10)} &= I((s_{5n-2}, s_{5n-4}), 0, G_n^{(11)}) \\
G_n^{(11)} &= G_{n-1} \\
G_n^{(8)} &= R((s_{5n-3}, s_{5n-4}), 0, G_n^{(12)}) \\
G_n^{(12)} &= I((s_{5n-3}, s_{5n}), 1, G_n^{(13)}) \\
G_n^{(13)} &= I((s_{5n}, s_{5n-2}), 0, G_n^{(14)}) \\
G_n^{(14)} &= I((s_{5n-2}, s_{5n-4}), 0, G_n^{(15)}) \\
G_n^{(15)} &= G_{n-1} \\
G_n^{(5)} &= R((s_{5n}, s_{5n-3}), 0, G_n^{(16)}) \\
G_n^{(16)} &= R((s_{5n-2}, s_{5n-4}), 0, G_n^{(17)}) \\
G_n^{(17)} &= I((s_{5n}, s_{5n-2}), 0, G_n^{(18)}) \\
G_n^{(18)} &= I((s_{5n-2}, s_{5n+1}), 1, G_n^{(19)}) \\
G_n^{(19)} &= I((s_{5n-1}, s_{5n-3}), 0, G_n^{(20)}) \\
G_n^{(20)} &= I((s_{5n-3}, s_{5n-4}), 0, G_n^{(21)}) \\
G_n^{(21)} &= G_{n-1}
\end{aligned}
$$

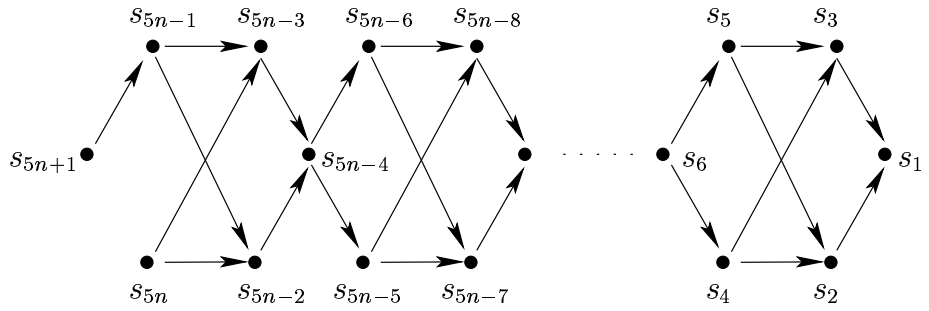


Figure 6.37: Constraint graph $G_n^{(1)}$ for plane partitions hexagonals with diagonals

$$
\begin{aligned}
G_n^{(2)} &= R((s_{5n}, s_{5n-3}), 0, G_n^{(22)}) \\
G_n^{(22)} &= R((s_{5n}, s_{5n-2}), 0, G_n^{(23)}) \\
G_n^{(23)} &= I((s_{5n}, s_{5n+1}), 1, G_n^{(24)}) \\
G_n^{(24)} &= I((s_{5n+1}, s_{5n-1}), 0, G_n^{(25)}) \\
G_n^{(25)} &= -((s_{5n-1}, s_{5n-3}), 0, G_n^{(26)}, G_n^{(27)}) \\
G_n^{(26)} &= I((s_{5n-1}, s_{5n-2}), 0, G_n^{(28)}) \\
G_n^{(28)} &= I((s_{5n-2}, s_{5n-4}), 0, G_n^{(29)}) \\
G_n^{(29)} &= I((s_{5n-3}, s_{5n-4}), 0, G_n^{(30)}) \\
G_n^{(30)} &= G_{n-1} \\
G_n^{(27)} &= R((s_{5n-3}, s_{5n-4}), 0, G_n^{(31)}) \\
G_n^{(31)} &= I((s_{5n-3}, s_{5n-1}), 1, G_n^{(32)}) \\
G_n^{(32)} &= I((s_{5n-1}, s_{5n-2}), 0, G_n^{(33)}) \\
G_n^{(33)} &= I((s_{5n-2}, s_{5n-4}), 0, G_n^{(34)}) \\
G_n^{(34)} &= G_{n-1}
\end{aligned}
$$

where $G_n^{(1)}$ is represented by Figure 6.37, $G_n^{(2)}$ by Figure 6.38, $G_n^{(4)}$ by Figure 6.39, $G_n^{(5)}$ by Figure 6.40, $G_n^{(7)}$ by Figure 6.41, $G_n^{(8)}$ by Figure 6.42, $G_n^{(26)}$ by Figure 6.43, $G_n^{(27)}$ by Figure 6.44 and G_{n-1} by Figure 6.45. We build the construction tree for the general case as in Figure 6.46. The base case G_0 is decomposed simply as

$$G_0 = s_{5n+1}.$$

We use the `GFPartitions` package by calling

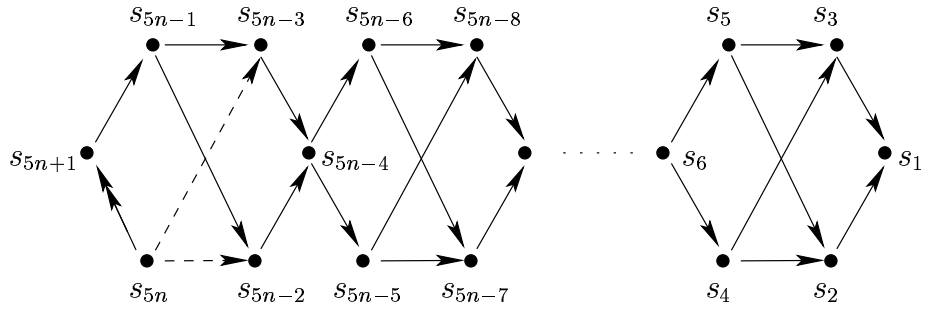


Figure 6.38: Constraint graph $G_n^{(2)}$ for plane partitions hexagonals with diagonals

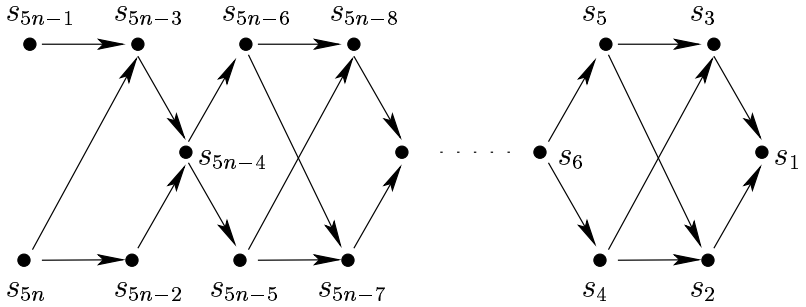


Figure 6.39: Constraint graph $G_n^{(4)}$ for plane partitions hexagonals with diagonals

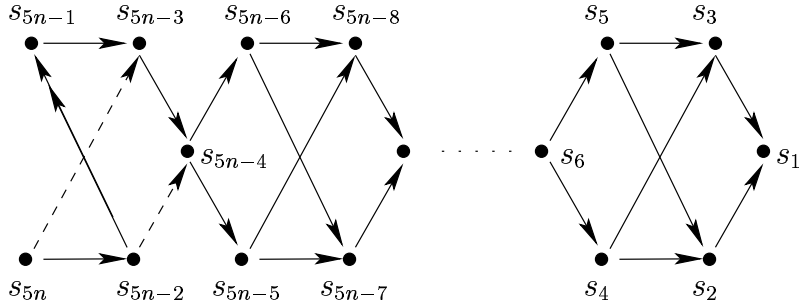


Figure 6.40: Constraint graph $G_n^{(5)}$ for plane partitions hexagonals with diagonals

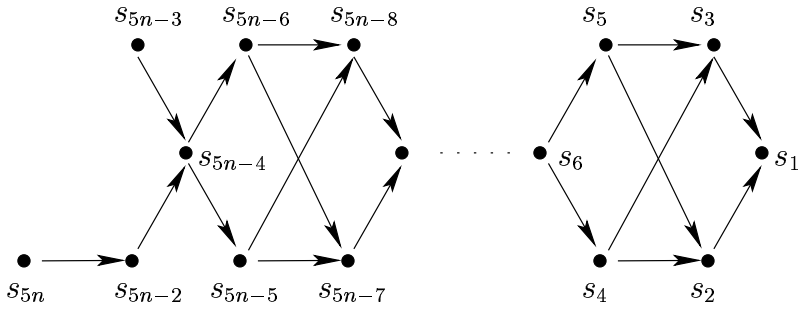


Figure 6.41: Constraint graph $G_n^{(7)}$ for plane partitions hexagonals with diagonals

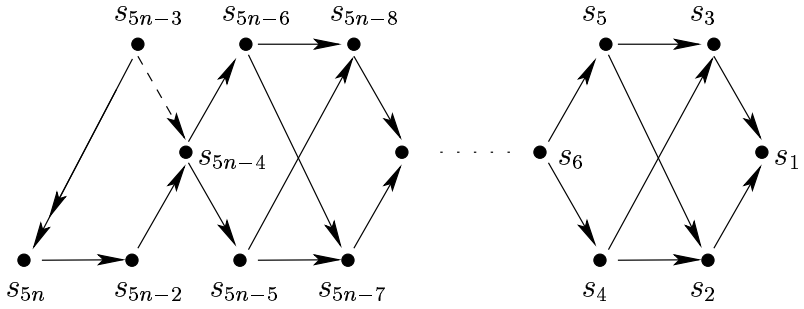


Figure 6.42: Constraint graph $G_n^{(8)}$ for plane partitions hexagonals with diagonals

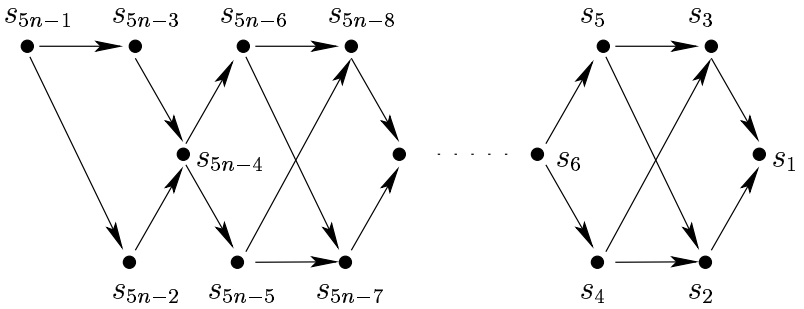


Figure 6.43: Constraint graph $G_n^{(26)}$ for plane partitions hexagonals with diagonals

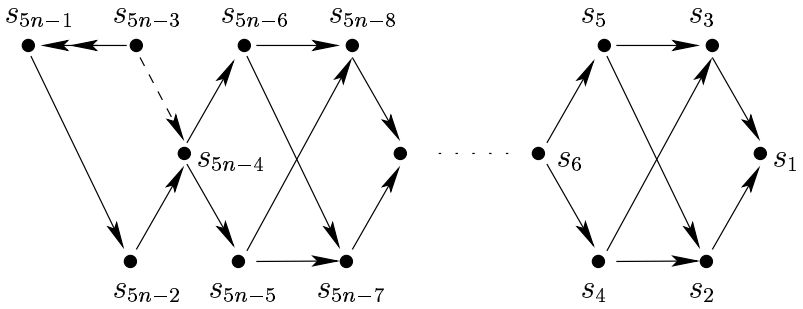


Figure 6.44: Constraint graph $G_n^{(27)}$ for plane partitions hexagonals with diagonals

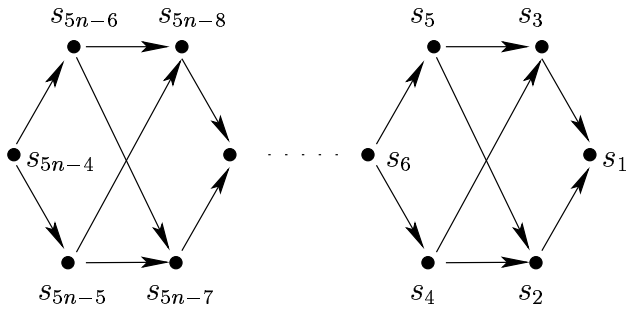


Figure 6.45: Constraint graph G_{n-1} for plane partitions hexagonals with diagonals

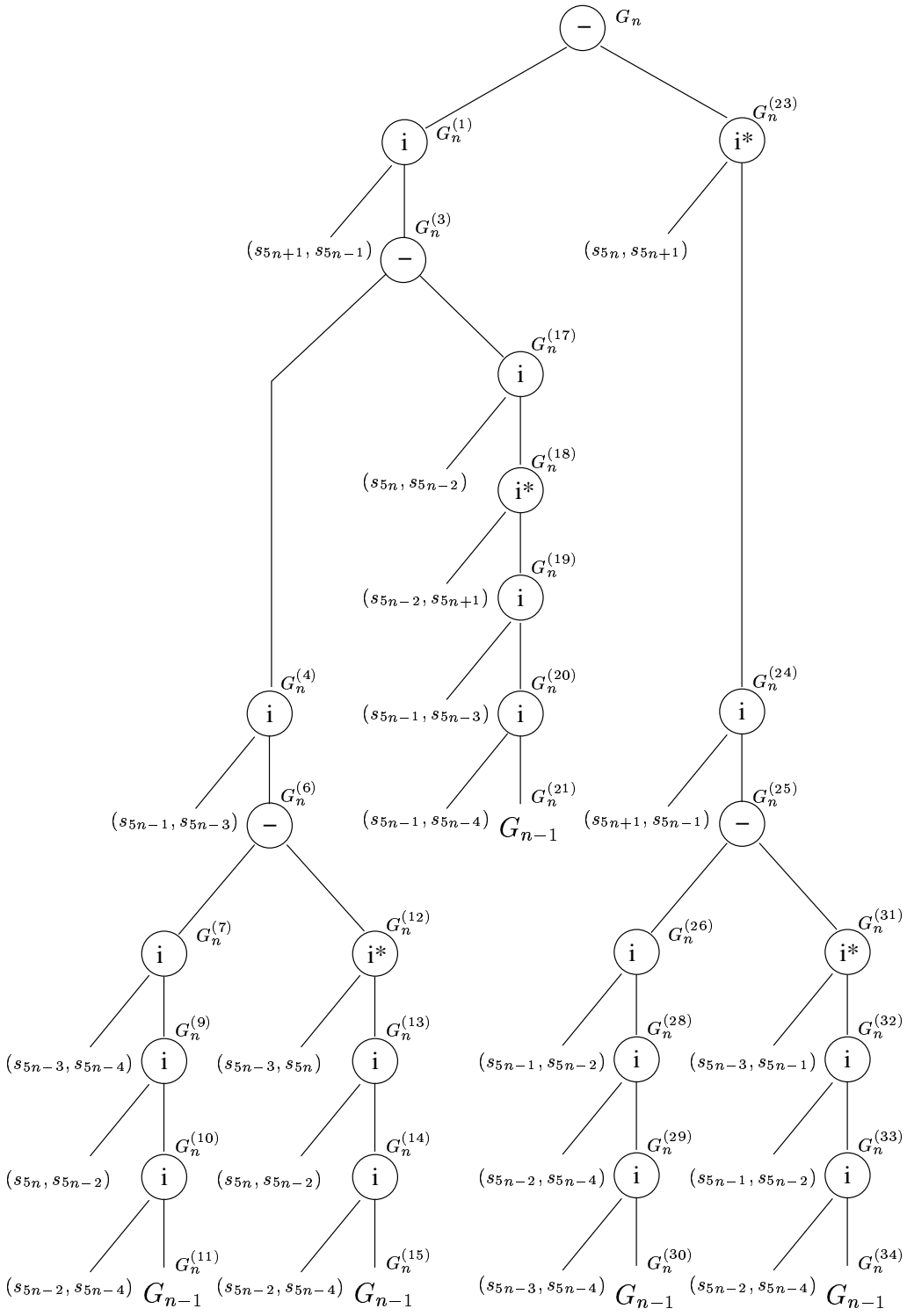


Figure 6.46: Construction tree for plane partitions hexagonals with diagonals

```
GFPartitions[Recurrence]([
    ["-",
        "i", x[5*n+1], x[5*n-1], "-",
            "i", x[5*n-1], x[5*n-3], "-",
                "i", x[5*n-3], x[5*n-4], "i", x[5*n],x[5*n-2],
                    "i",x[5*n-2], x[5*n-4], "%",
                "i*", x[5*n-3], x[5*n], "i", x[5*n], x[5*n-2],
                    "i", x[5*n-2], x[5*n-4], "%",
            "i", x[5*n], x[5*n-2], "i*", x[5*n-2], x[5*n-1],
                "i", x[5*n-1], x[5*n-3], "i", x[5*n-3], x[5*n-4], "%",
        "i*", x[5*n], x[5*n+1], "i", x[5*n+1], x[5*n-1], "-",
            "i", x[5*n-1], x[5*n-2], "i", x[5*n-2], x[5*n-4],
                "i",x[5*n-3], x[5*n-4], "%",
            "i*", x[5*n-3], x[5*n-1], "i", x[5*n-1], x[5*n-2],
        "i", x[5*n-2], x[5*n-4], "%"],
    n=0, [x[5*n+1]]],
    [s])
```

and the output generated is

```
========================  Multi-variable recurrence  ========================
```

```
Case "default"
```

$$\frac{R_{19}}{(1-x_{5\,n+1})\,(1-x_{5\,n-2}x_{5\,n})\,(1-x_{5\,n})\,(1-x_{5\,n-3}x_{5\,n-1}x_{5\,n+1})\,(1-x_{5\,n-1}x_{5\,n+1})}$$

$$-\frac{(1-x_{5\,n+1})^{-1}\,(1-x_{5\,n-1}x_{5\,n+1})^{-1}\,R_{29}x_{5\,n-3}x_{5\,n-1}x_{5\,n+1}}{(1-x_{5\,n-2}x_{5\,n}x_{5\,n-3}x_{5\,n-1}x_{5\,n+1})\,(1-x_{5\,n}x_{5\,n-3}x_{5\,n-1}x_{5\,n+1})\,(1-x_{5\,n-3}x_{5\,n-1}x_{5\,n+1})}$$

$$-\frac{(1-x_{5\,n+1})^{-1}\,R_{42}x_{5\,n-2}x_{5\,n}}{(1-x_{5\,n-2}x_{5\,n}x_{5\,n-3}x_{5\,n-1}x_{5\,n+1})\,(1-x_{5\,n-1}x_{5\,n+1}x_{5\,n-2}x_{5\,n})\,(1-x_{5\,n-2}x_{5\,n})\,(1-x_{5\,n})}$$

$$-\frac{R_{59}x_{5\,n}}{(1-x_{5\,n+1}x_{5\,n})\,(1-x_{5\,n})\,(1-x_{5\,n-3})\,(1-x_{5\,n-1}x_{5\,n+1}x_{5\,n-2}x_{5\,n})\,(1-x_{5\,n-1}x_{5\,n+1}x_{5\,n})}$$

$$-\frac{R_{69}x_{5\,n}x_{5\,n-3}\,(1-x_{5\,n+1}x_{5\,n})^{-1}}{(1-x_{5\,n})\,(1-x_{5\,n-2}x_{5\,n}x_{5\,n-3}x_{5\,n-1}x_{5\,n+1})\,(1-x_{5\,n}x_{5\,n-3}x_{5\,n-1}x_{5\,n+1})\,(1-x_{5\,n-3})}$$

```
        R[42] -- F(n-1)
                x[5*n-4] <--- x[5*n-4]*x[5*n-3]*x[5*n-1]*x[5*n+1]*x[5*n-2]*x[5*n]
        R[59] -- F(n-1)
                x[5*n-4] <--- x[5*n-4]*x[5*n-3]*x[5*n-1]*x[5*n+1]*x[5*n-2]*x[5*n]
        R[29] -- F(n-1)
                x[5*n-4] <--- x[5*n-4]*x[5*n-3]*x[5*n-1]*x[5*n+1]*x[5*n-2]*x[5*n]
        R[19] -- F(n-1)
                x[5*n-4] <--- x[5*n-4]*x[5*n-3]*x[5*n-1]*x[5*n+1]*x[5*n-2]*x[5*n]
        R[69] -- F(n-1)
                x[5*n-4] <--- x[5*n-4]*x[5*n-3]*x[5*n-1]*x[5*n+1]*x[5*n-2]*x[5*n]
Case n = 0
```

$$\frac{1}{(1-x_{5\,n+1})}$$

```
======================== Finite-variable recurrence ========================
            x[5*n+1] --> s
```

$$H(n,q,s)$$

```
if n = 0 then
```

$$\frac{1}{(1-s)}$$

```
else
```

$$\frac{\frac{H\left(n-1,q,q^5 s\right)}{(1-q^2)(1-q)(1-q^2 s)}-\frac{H\left(n-1,q,q^5 s\right)q^2 s}{(1-q^4 s)(1-q^3 s)(1-q^2 s)}}{(1-sq)\,(1-s)}-\frac{\frac{H\left(n-1,q,q^5 s\right)q^2}{(1-q^4 s)(1-q^3 s)(1-q^2)(1-q)}}{(1-s)}$$

$$-\frac{\frac{H\left(n-1,q,q^5 s\right)q}{(1-q)(1-q^3 s)(1-q^2 s)}-\frac{H\left(n-1,q,q^5 s\right)q^2}{(1-q^4 s)(1-q^3 s)(1-q)}}{(1-sq)\,(1-q)}$$

```
fi;
```

The above recurrence can be shown to correspond to the generating function obtained in [2].

6.8 Gordon sequences

In [22], Gordon provides a generalization of the Rogers-Ramanujan identities. Figure 6.47 corresponds to a family of sequences enumerated in this theorem, such that every alternate integer must differ by at least 2.

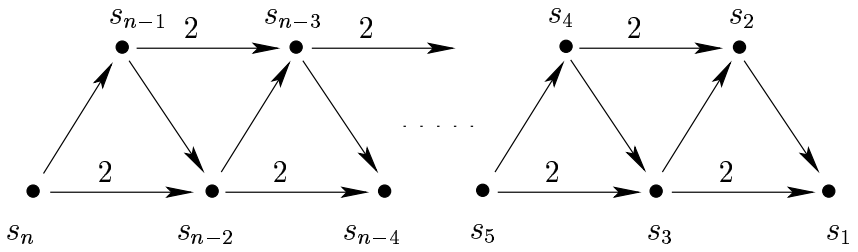


Figure 6.47: Gordon sequences; constraint graph G_n

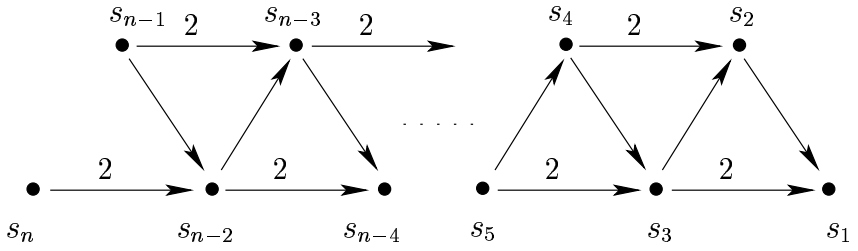


Figure 6.48: Constraint graph $G_n^{(1)}$ for Gordon sequences

The general case G_n for Gordon sequences can be decomposed as

$$\begin{aligned}
G_n &= -((s_n, s_{n-1}), 0, G_n^{(1)}, G_n^{(2)}) \\
G_n^{(1)} &= I((s_n, s_{n-2}), 2, G_n^{(3)}) \\
G_n^{(3)} &= G_{n-1} \\
G_n^{(2)} &= R((s_{n-1}, s_{n-2}), 0, G_n^{(4)}) \\
G_n^{(4)} &= R((s_{n-1}, s_{n-3}), 2, G_n^{(5)}) \\
G_n^{(5)} &= I((s_{n-1}, s_n), 1, G_n^{(6)}) \\
G_n^{(6)} &= I((s_n, s_{n-2}), 2, G_n^{(7)}) \\
G_n^{(7)} &= G_{n-2}
\end{aligned}$$

where $G_n^{(1)}$ is represented by Figure 6.48, $G_n^{(2)}$ by Figure 6.49, G_{n-1} by Figure 6.50 and G_{n-2} by Figure 6.51. We build the construction tree for the general case as in Figure 6.52.

We note that we need two base cases G_1 and G_2 (Figure 6.53) because the general case is defined upon both G_{n-1} and G_{n-2}. Graphs G_1 and G_2 can be decomposed as follows.

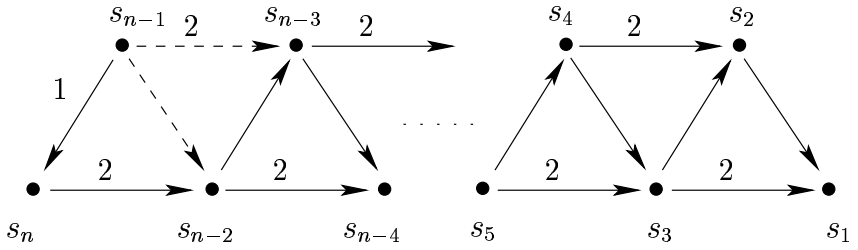


Figure 6.49: Constraint graph $G_n^{(2)}$ for Gordon sequences

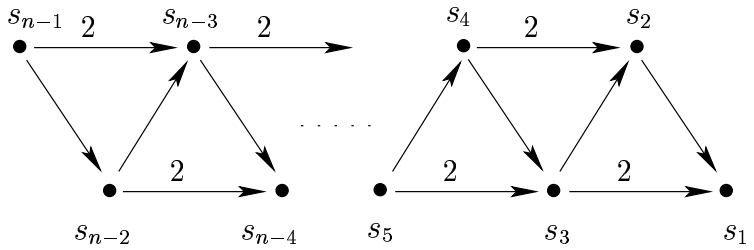


Figure 6.50: Constraint graph G_{n-1} for Gordon sequences

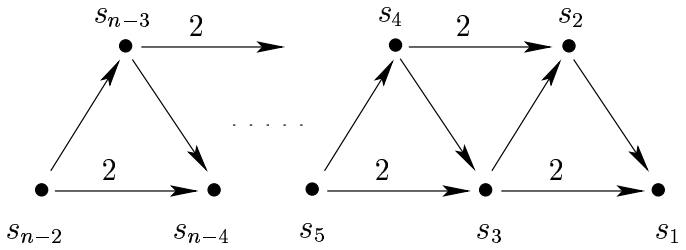


Figure 6.51: Constraint graph G_{n-2} for Gordon sequences

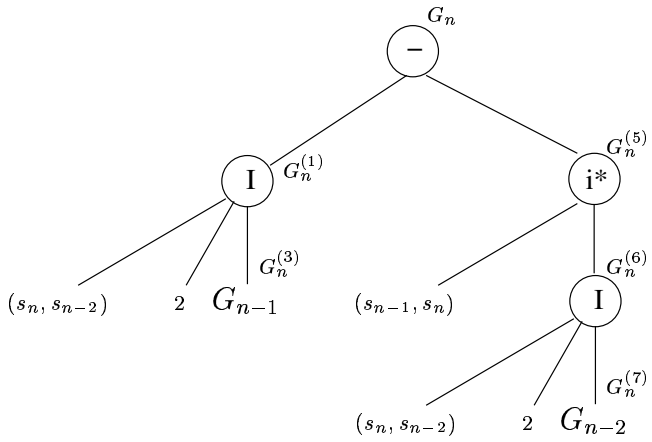


Figure 6.52: Construction tree for Gordon sequences

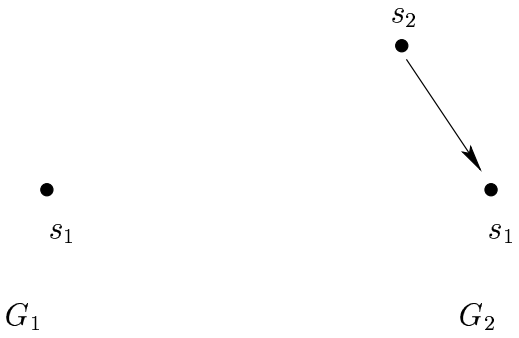


Figure 6.53: Constraint graphs G_1 and G_2 for Gordon sequences

$$\begin{aligned} G_1 &= s_n \\ G_2 &= I((s_n, s_{n-1}), 0, G_2^{(1)}) \\ G_2^{(1)} &= s_{n-1} \end{aligned}$$

Procedure `Recurrence` is invoked as

```
GFPartitions[Recurrence]([
    ["-",
        "I", x[n], x[n-2], 2, "%",
        "i*", x[n-1], x[n], "I", x[n], x[n-2], 2, "%%", 2],
    n=1, [x[n]],
    n=2, ["i", x[n], x[n-1], x[n-1]]],
    [s,t])
```

and the output generated is

```
========================  Multi-variable recurrence  ========================
```

```
Case "default"
```

$$\frac{R_6 {x_n}^2}{1 - x_n} - \frac{R_{14} {x_n}^2 {x_{n-1}}^3}{(1 - x_n x_{n-1})(1 - x_{n-1})}$$

```
        R[6] -- F(n-1)
                 x[n-2] <--- x[n-2]*x[n]
        R[14] -- F(n-2)
                 x[n-2] <--- x[n-2]*x[n]*x[n-1]
Case n = 1
```

$$\frac{1}{(1 - x_n)}$$

```
Case n = 2
```

$$\frac{1}{(1 - x_n x_{n-1})(1 - x_n)}$$

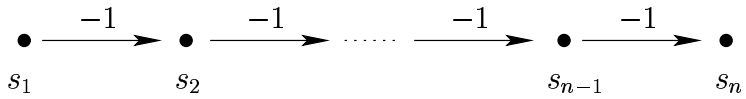


Figure 6.54: 1-compositions; constraint graph G_n

```
======================== Finite-variable recurrence ========================
            x[n-1] --> s
            x[n] --> t
```

$$H(n,q,s,t)$$

```
if n = 1 then
```

$$\frac{1}{(1-t)}$$

```
elif n = 2 then
```

$$\frac{1}{(1-ts)\,(1-t)}$$

```
else
```

$$\frac{H\,(n-1,q,qt,s)\,t^2}{1-t} - \frac{H\,(n-2,q,q,qts)\,t^2 s^3}{(1-ts)\,(1-s)}$$

```
fi;
```

6.9 1-compositions

We apply the Seven Rules and the `GFPartitions` package to solve 1-compositions, defined by G_n of Figure 6.54. These were studied and solved in [26].

The base case G_1 is simply the vertex s_1. The general case G_n for 1-compositions can be decomposed as

$$\begin{aligned}
G_n &= -((s_{n-1}, s_n), -1, G_n^{(1)}, G_n^{(2)}) \\
G_n^{(1)} &= V(s_n, G_n^{(3)}) \\
G_n^{(3)} &= G_{n-1} \\
G_n^{(2)} &= I((s_n, s_{n-1}), 2, G_n^{(4)}) \\
G_n^{(4)} &= G_{n-1}
\end{aligned}$$

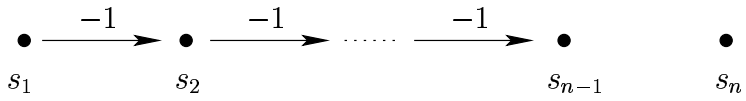


Figure 6.55: Constraint graph $G_n^{(1)}$ for 1-compositions

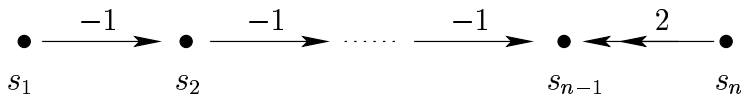


Figure 6.56: Constraint graph $G_n^{(2)}$ for 1-compositions

where $G_n^{(1)}$ is represented by Figure 6.55, $G_n^{(2)}$ by Figure 6.56, and G_{n-1} by Figure 6.57. The general case has the construction tree of Figure 6.58. The base case G_1 is decomposed simply as

$$G_1 \quad = \quad s_n.$$

The `Recurrence` procedure in the `GFPartitions` package is called as

```
GFPartitions[Recurrence]([
    ["-",
        ".", x[n], "%",
        "I", x[n], x[n-1], 2, "%"],
    n=1, [x[n]]],
    [s])
```

and the output generated is

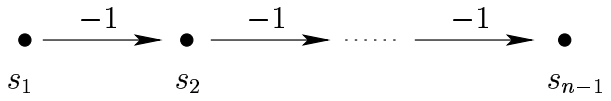


Figure 6.57: Constraint graph G_{n-1} for 1-compositions

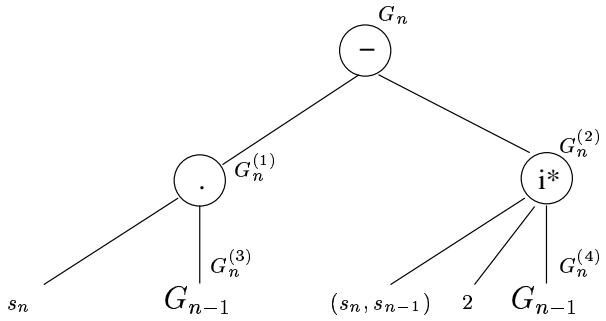


Figure 6.58: Construction tree for 1-compositions

```
========================  Multi-variable recurrence  ========================
```

```
Case "default"
```

$$\frac{R_4}{1-x_n} - \frac{R_9 {x_n}^2}{1-x_n}$$

```
        R[4] -- F(n-1)
        R[9] -- F(n-1)
                 x[n-1] <--- x[n-1]*x[n]
Case n = 1
```

$$\frac{1}{(1-x_n)}$$

```
========================  Finite-variable recurrence  ========================
            x[n] --> s
```

$$H(n,q,s)$$

```
if n = 1 then
```

$$\frac{1}{(1-s)}$$

```
else
```

$$\frac{H\left(n-1,q,q\right)}{1-s} - \frac{H\left(n-1,q,qs\right)s^2}{1-s}$$

```
fi;
```

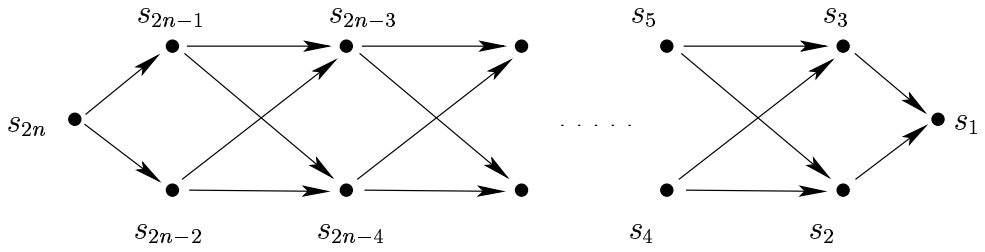


Figure 6.59: 2-rowed plane partitions with double diagonals; constraint graph G_n

6.10 2-rowed plane partitions with double diagonals

Consider the sequence of constraint graphs represented by Figure 6.59, first investigated by Corteel and whose generating function she conjectured [17] as

$$F_G(n,q) = \frac{(-q^2;q^2)_{n-1}}{(q;q)_{2n}} \tag{6.5}$$

We refer to these as 2-rowed plane partitions with double diagonals since they resemble the 2-rowed plane partitions with diagonals of Figure 6.20 except that there are here twice as many diagonals. We offer a proof for (6.5) after obtaining a recurrence for it.

The general case G_n for 2-rowed plane partitions with double diagonals can be decomposed as

$$\begin{aligned}
G_n &= -((s_{2n}, s_{2n-1}), 0, G_n^{(1)}, G_n^{(2)}) \\
G_n^{(1)} &= I((s_{2n}, s_{2n-2}), 0, G_n^{(3)}) \\
G_n^{(3)} &= -((s_{2n-1}, s_{2n-3}), 0, G_n^{(4)}, G_n^{(5)}) \\
G_n^{(4)} &= I((s_{2n-1}, s_{2n-4}), 0, G_n^{(6)}) \\
G_n^{(6)} &= G_{n-1} \\
G_n^{(5)} &= R((s_{2n-3}, s_{2n-5}), 0, G_n^{(7)}) \\
G_n^{(7)} &= R((s_{2n-3}, s_{2n-6}), 0, G_n^{(8)}) \\
G_n^{(8)} &= R((s_{2n-2}, s_{2n-4}), 0, G_n^{(9)}) \\
G_n^{(9)} &= I((s_{2n-2}, s_{2n-3}), 0, G_n^{(10)}) \\
G_n^{(10)} &= I((s_{2n-3}, s_{2n-1}), 1, G_n^{(11)}) \\
G_n^{(11)} &= I((s_{2n-1}, s_{2n-4}), 0, G_n^{(12)}) \\
G_n^{(12)} &= G_{n-2}
\end{aligned}$$

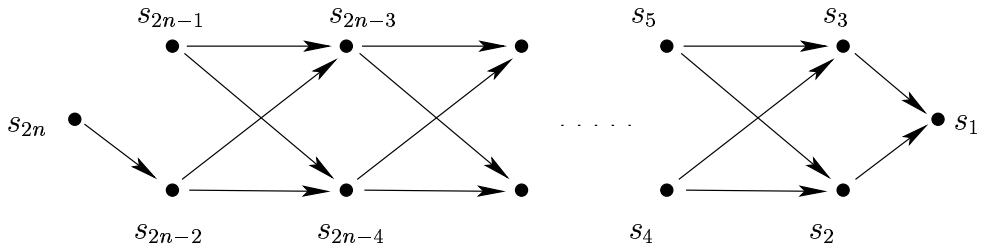


Figure 6.60: Constraint graph $G_n^{(1)}$ for 2-rowed plane partitions with double diagonals

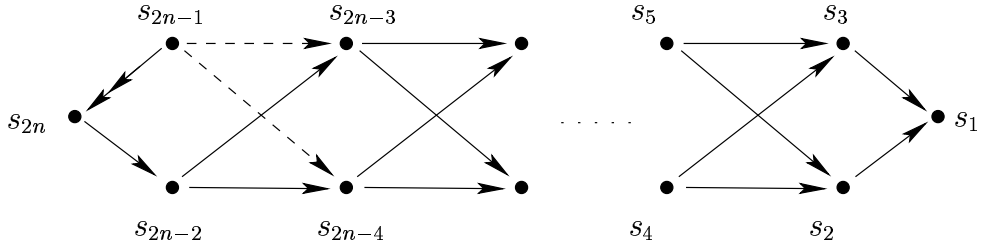


Figure 6.61: Constraint graph $G_n^{(2)}$ for 2-rowed plane partitions with double diagonals

$$
\begin{aligned}
G_n^{(2)} &= R((s_{2n-1}, s_{2n-3}), 0, G_n^{(13)}) \\
G_n^{(13)} &= R((s_{2n-1}, s_{2n-4}), 0, G_n^{(14)}) \\
G_n^{(14)} &= I((s_{2n-1}, s_{2n}), 1, G_n^{(15)}) \\
G_n^{(15)} &= I((s_{2n}, s_{2n-2}), 0, G_n^{(16)}) \\
G_n^{(16)} &= G_{n-1}
\end{aligned}
$$

where $G_n^{(1)}$ is represented by Figure 6.60, $G_n^{(2)}$ by Figure 6.61, $G_n^{(4)}$ by Figure 6.62, $G_n^{(5)}$ by Figure 6.63, G_{n-1} by Figure 6.64. and G_{n-2} by Figure 6.65. The construction tree for the general case is shown in Figure 6.66.

There are two base cases for 2-rowed plane partitions with double diagonals because G_n is defined over both G_{n-1} and G_{n-2}. Base cases G_1 and G_2 are shown in Figure 6.67 and can be decomposed

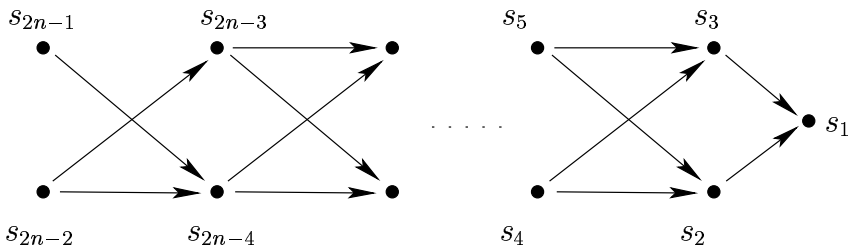


Figure 6.62: Constraint graph $G_n^{(4)}$ for 2-rowed plane partitions with double diagonals

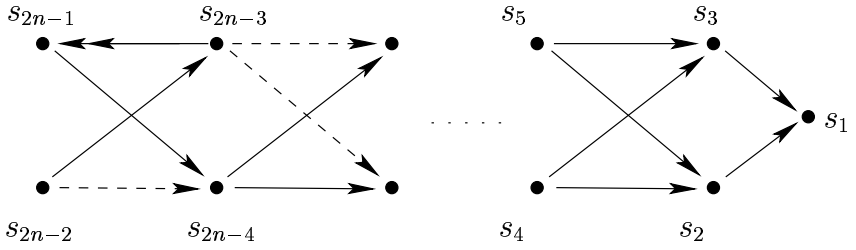


Figure 6.63: Constraint graph $G_n^{(5)}$ for 2-rowed plane partitions with double diagonals

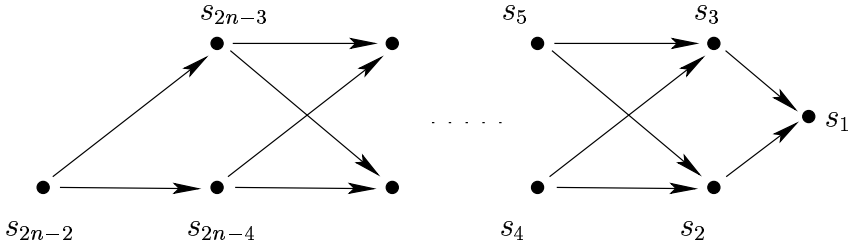


Figure 6.64: Constraint graph G_{n-1} for 2-rowed plane partitions with double diagonals

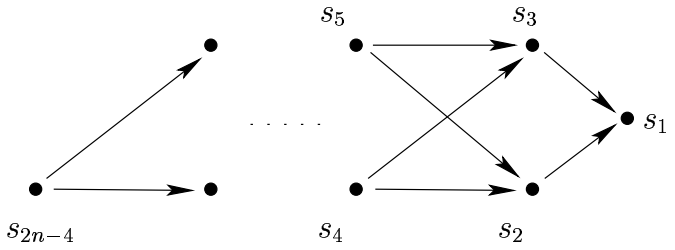


Figure 6.65: Constraint graph G_{n-2} for 2-rowed plane partitions with double diagonals

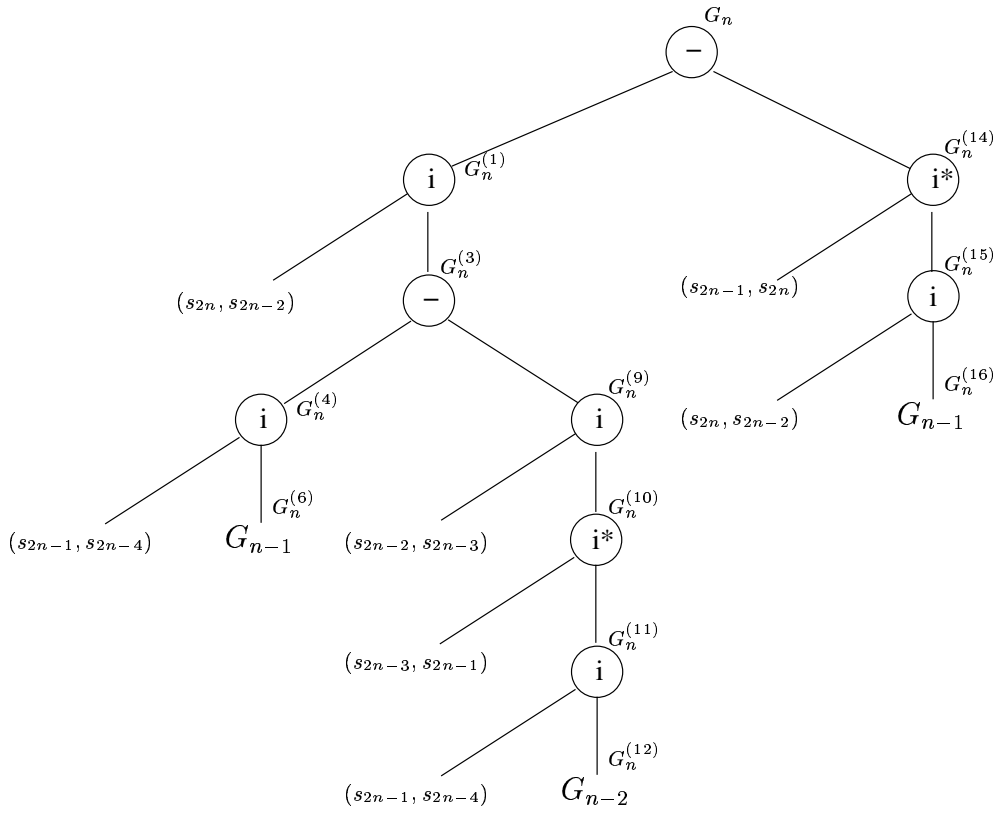


Figure 6.66: Construction tree for the decomposition of G_n for 2-rowed plane partitions with double diagonals

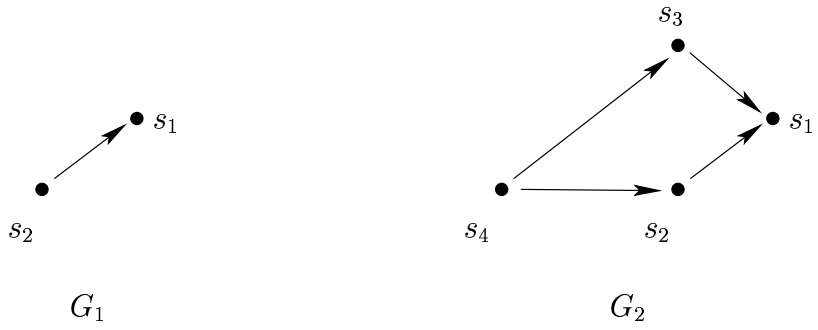


Figure 6.67: Constraint graphs G_1 and G_2 for 2-rowed plane partitions with double diagonals

as follows.

$$\begin{aligned}
G_1 &= I((s_{2n}, s_{2n-1}), 0, G_1^{(1)}) \\
G_1^{(1)} &= s_{2n-1} \\
G_2 &= -((s_{2n}, s_{2n-1}), 0, G_2^{(1)}, G_2^{(2)}) \\
G_2^{(1)} &= I((s_{2n}, s_{2n-2}), 0, G_2^{(3)}) \\
G_2^{(3)} &= I((s_{2n-2}, s_{2n-3}), 0, G_2^{(4)}) \\
G_2^{(4)} &= I((s_{2n-1}, s_{2n-3}), 0, G_2^{(5)}) \\
G_2^{(5)} &= s_{2n-3} \\
G_2^{(2)} &= R((s_{2n-1}, s_{2n-3}), 0, G_n^{(6)}) \\
G_2^{(6)} &= I((s_{2n-1}, s_{2n}), 1, G_2^{(7)}) \\
G_2^{(7)} &= I((s_{2n}, s_{2n-2}), 0, G_2^{(8)}) \\
G_2^{(8)} &= I((s_{2n-2}, s_{2n-3}), 0, G_2^{(9)}) \\
G_2^{(9)} &= s_{2n-3}.
\end{aligned}$$

We invoke procedure `Recurrence` as

```
GFPartitions[Recurrence]([
    ["-",
        "i", x[2*n], x[2*n-2], "-",
            "i", x[2*n-1], x[2*n-4], "%", "i", x[2*n-2], x[2*n-3],
            "i*", x[2*n-3], x[2*n-1], "i", x[2*n-1], x[2*n-4], "%%", 2,
        "i*", x[2*n-1], x[2*n], "i", x[2*n], x[2*n-2], "%"],
    n=1,
    ["i", x[2*n], x[2*n-1], x[2*n-1]],
    n=2,
    ["-",
        "i", x[2*n], x[2*n-2], "i", x[2*n-2], x[2*n-3],
        "i", x[2*n-1], x[2*n-3], x[2*n-3],
        "i*", x[2*n-1], x[2*n], "i", x[2*n], x[2*n-2],
        "i", x[2*n-2], x[2*n-3], x[2*n-3]]],
    [r,t])
```

and the output generated is

```
========================  Multi-variable recurrence  ========================
```

Case "default"

$$\frac{\frac{R_9}{1-x_{2\,n-1}} - \frac{R_{19}\,x_{2\,n-3}\,x_{2\,n-2}\,x_{2\,n}}{(1-x_{2\,n-1}\,x_{2\,n-3}\,x_{2\,n-2}\,x_{2\,n})\,(1-x_{2\,n-3}\,x_{2\,n-2}\,x_{2\,n})\,(1-x_{2\,n-2}\,x_{2\,n})}}{1-x_{2\,n}} - \frac{R_{27}\,x_{2\,n-1}}{(1-x_{2\,n}\,x_{2\,n-1})\,(1-x_{2\,n-1})}$$

```
        R[19] -- F(n-2)
                 x[2*n-4] <--- x[2*n-4]*x[2*n-1]*x[2*n-3]*x[2*n-2]*x[2*n]
        R[27] -- F(n-1)
                 x[2*n-2] <--- x[2*n-2]*x[2*n]*x[2*n-1]
        R[9] -- F(n-1)
                 x[2*n-2] <--- x[2*n-2]*x[2*n]
                 x[2*n-4] <--- x[2*n-4]*x[2*n-1]
```

Case n = 1

$$\frac{1}{(1-x_{2\,n}\,x_{2\,n-1})\,(1-x_{2\,n})}$$

Case n = 2

$$\frac{1}{(1-x_{2\,n-1}\,x_{2\,n-3}\,x_{2\,n-2}\,x_{2\,n})\,(1-x_{2\,n-1})\,(1-x_{2\,n-2}\,x_{2\,n})\;(1-x_{2\,n})} - \frac{x_{2\,n-1}}{(1-x_{2\,n-1}\,x_{2\,n-3}\,x_{2\,n-2}\,x_{2\,n})\,(1-x_{2\,n-2}\,x_{2\,n}\,x_{2\,n-1})\;(1-x_{2\,n}\,x_{2\,n-1})\,(1-x_{2\,n-1})}$$

```
========================  Finite-variable recurrence  ========================
            x[2*n-2] --> r
            x[2*n] --> t
```

$$H(n,q,r,t)$$

```
if n = 1 then
```

$$\frac{1}{(1-tq)\,(1-t)}$$

```
elif n = 2 then
```

$$\frac{1}{(1-q^2rt)\,(1-q)\,(1-rt)\,(1-t)} - \frac{q}{(1-q^2rt)\,(1-qrt)\,(1-tq)\,(1-q)}$$

```
else
```

$$\frac{\frac{H\left(n-1,q,q^2,rt\right)}{1-q} - \frac{H\left(n-2,q,q,q^3rt\right)qrt}{(1-q^2rt)(1-qrt)(1-rt)}}{(1-t)} - \frac{H\,(n-1,q,q,qrt)\,q}{(1-tq)\,(1-q)}$$

```
fi;
```

Theorem 9 *The solution to the recurrence for 2-rowed plane partitions with double diagonals is*

$$H'(n,q,r,t) = \frac{(1-qrt^2)\prod_{i=1}^{n-2}\frac{(1+q^{2i}rt)}{(1-q^{2i+1}rt)(1-q^{2i+2}rt)}}{(1-rt)(1-qrt)(1-t)(1-qt)(1-q^2rt)}$$

for $n \geq 2$ *,and*

$$H'(n,q,r,t) = \frac{1}{(1-t)(1-qt)}$$

for $n = 1$.

Proof. We prove by induction on n. For $n = 1$ and $n = 2$ the proof is direct. Assume $H'(n-1,q,r,t) = H(n-1,q,r,t)$ for some $n > 2$. We prove that $H'(n,q,r,t) = H(n,q,r,t)$. From the recurrence, we know that for $n > 2$,

$$H(n,q,r,t) = \frac{H(n-1,q,q^2,rt)}{(1-t)(1-q)} - qrt\frac{H(n-2,q,q,q^3rt)}{(1-rt)(1-t)(1-qrt)(1-q^2rt)} - q\frac{H(n-1,q,q,qrt)}{(1-qt)(1-q)}.$$

Since $H(n-1,q,r,qt) = H'(n-1,q,r,t)$,

$$H(n,q,r,t) \quad = \quad \frac{H'(n-1,q,q^2,rt)}{(1-t)(1-q)} - qrt\frac{H'(n-2,q,q,q^3rt)}{(1-rt)(1-t)(1-qrt)(1-q^2rt)} - q\frac{H'(n-1,q,q,qrt)}{(1-qt)(1-q)}$$

$$
\begin{aligned}
&= \frac{(1-q^3r^2t^2)\prod_{i=1}^{n-3}\frac{(1+q^{2i-1}q^3rt)}{(1-q^{2i}q^3rt)(1-q^{2i+1}q^3rt)}}{(1-t)(1-q)(1-q^2rt)(1-q^3rt)(1-rt)(1-qrt)(1-q^4rt)}\\
&\quad -\frac{qrt(1-q^8r^2t^2)\prod_{i=1}^{n-4}\frac{(1-q^{2i-1}q^5rt)}{(1-q^{2i}q^5rt)(1-q^{2i+1}q^5rt)}}{(1-q^2rt)(1-qrt)(1-rt)(1-t)(1-q^4rt)(1-q^5rt)(1-q^3rt)(1-q^4rt)(1-q^6rt)}\\
&\quad -\frac{q(1-q^4r^2t^2)\prod_{i=1}^{n-3}\frac{(1-q^{2i-1}q^3rt)}{(1-q^{2i}q^3rt)(1-q^{2i+1}q^3rt)}}{(1-qt)(1-q)(1-q^2rt)(1-q^3rt)(1-qrt)(1-q^2rt)(1-q^4rt)}\\
&= \frac{(1-q^3r^2t^2)\prod_{i=2}^{n-2}\frac{(1+q^{2i-1}qrt)}{(1-q^{2i}qrt)(1-q^{2i+1}qrt)}}{(1-t)(1-q)(1-rt)(1-qrt)(1-q^2rt)(1-q^3rt)(1-q^4rt)}\\
&\quad -\frac{qrt(1-q^8r^2t^2)\prod_{i=3}^{n-2}\frac{(1+q^{2i-1}qrt)}{(1-q^{2i}qrt)(1-q^{2i+1}qrt)}}{(1-t)(1-rt)(1-qrt)(1-q^2rt)(1-q^3rt)(1-q^4rt)^2(1-q^5rt)(1-q^6rt)}\\
&\quad -\frac{q(1-q^4r^2t^2)\prod_{i=2}^{n-2}\frac{(1+q^{2i-1}qrt)}{(1-q^{2i}qrt)(1-q^{2i+1}qrt)}}{(1-q)(1-qt)(1-qrt)(1-q^2rt)^2(1-q^3rt)(1-q^4rt)}\\
&= \frac{\prod_{i=3}^{n-2}\frac{(1+q^{2i-1}qrt)}{(1-q^{2i}qrt)(1-q^{2i+1}qrt)}}{(1-q^2rt)(1-q^3rt)(1-q^4rt)}\left[\frac{(1-q^3r^2t^2)(1+q^4rt)}{(1-q^5rt)(1-q^6rt)(1-t)(1-q)(1-rt)(1-qrt)}\right.\\
&\quad -\frac{qrt(1-q^8r^2t^2)}{(1-t)(1-rt)(1-qrt)(1-q^4rt)(1-q^5rt)(1-q^6rt)}\\
&\quad \left.-\frac{q(1-q^4r^2t^2)(1+q^4rt)}{(1-q^5rt)(1-q^6rt)(1-q)(1-qt)(1-qrt)(1-q^2rt)}\right]\\
&= \frac{\prod_{i=2}^{n-2}\frac{(1+q^{2i-1}qrt)}{(1-q^{2i}qrt)(1-q^{2i+1}qrt)}}{(1-q^2rt)(1-q^3rt)(1-q^4rt)}\left[\frac{(1-q^3r^2t^2)}{(1-t)(1-q)(1-rt)(1-qrt)}\right.\\
&\quad \left.-\frac{qrt(1-q^4rt)}{(1-t)(1-rt)(1-qrt)(1-q^4rt)}-\frac{q(1-q^4r^2t^2)}{(1-q)(1-qt)(1-qrt)(1-q^2rt)}\right]\\
&= \frac{\prod_{i=2}^{n-2}\frac{(1+q^{2i-1}qrt)}{(1-q^{2i}qrt)(1-q^{2i+1}qrt)}}{(1-q^2rt)(1-q^3rt)(1-q^4rt)}\\
&\quad \left[\frac{(1-q^3r^2t^2)(1-qt)-qrt(1-qt)(1-q)-q(1+q^2rt)(1-t)(1-rt)}{(1-t)(1-qt)(1-qrt)(1-rt)(1-q)}\right]\\
&= \frac{\prod_{i=2}^{n-2}\frac{(1+q^{2i-1}qrt)}{(1-q^{2i}qrt)(1-q^{2i+1}qrt)}}{(1-q^2rt)(1-q^3rt)(1-q^4rt)(1-t)(1-qt)(1-qrt)(1-rt)}\\
&\quad \left[\frac{(1+q^2rt)((q^2t^2r-qt-qrt+1)-(q+qrt^2-qrt-qt))}{(1-q)}\right]\\
&= \frac{(1-qrt^2)\prod_{i=1}^{n-2}\frac{(1+q^{2i-1}qrt)}{(1-q^{2i}qrt)(1-q^{2i+1}qrt)}}{(1-rt)(1-qrt)(1-t)(1-qt)(1-q^2rt)}\\
&= H'(n,q,r,t).
\end{aligned}
$$

■

Note that replacing r and t by q in $H'(n,q,r,t)$ gives

$$\begin{aligned} H'(n,q,q,q) &= \frac{(1-q^4)\prod_{i=1}^{n-2}\frac{(1+q^{2i+2})}{(1-q^{2i+3})(1-q^{2i+4})}}{(1-q^2)(1-q^3)(1-q)(1-q^2)(1-q^4)} \\ &= \frac{\prod_{i=0}^{n-2}\frac{(1+q^{2i+2})}{(1-q^{2i+3})(1-q^{2i+4})}}{(1-q)(1-q^2)} \\ &= \frac{\prod_{i=1}^{n-1}\frac{(1+q^{2i})}{(1-q^{2i+1})(1-q^{2i+2})}}{(1-q)(1-q^2)} \\ &= \frac{(-q^2;q^2)_{n-1}}{(q;q)_{2n}}, \end{aligned}$$

thus proving Corteel's conjecture [17].

6.11 Left-shifted 2-rowed plane partitions

We now consider the left-shifted 2-rowed plane partitions of Figure 6.68.

The general case G_n for left-shifted 2-rowed plane partitions can be decomposed as

$$\begin{aligned} G_n &= -((s_{2n}, s_{2n-1}), 0, G_n^{(1)}, G_n^{(2)}) \\ G_n^{(1)} &= I((s_{2n}, s_{2n-2}), 0, G_n^{(3)}) \\ G_n^{(3)} &= I((s_{2n-1}, s_{2n-4}), 0, G_n^{(4)}) \\ G_n^{(4)} &= G_{n-1} \\ G_n^{(2)} &= R((s_{2n-1}, s_{2n-4}), 0, G_n^{(5)}) \\ G_n^{(5)} &= I((s_{2n-1}, s_{2n}), 1, G_n^{(6)}) \\ G_n^{(6)} &= I((s_{2n}, s_{2n-2}), 0, G_n^{(7)}) \\ G_n^{(7)} &= G_{n-1} \end{aligned}$$

where $G_n^{(1)}$ is represented by Figure 6.69, $G_n^{(2)}$ by Figure 6.70, and G_{n-1} by Figure 6.71. The construction tree for the general case is shown in Figure 6.72.

There are two base cases for left-shifted 2-rowed plane partitions (Figure 6.73). In addition to G_1 we need the decomposition for G_2 since the labelling of the vertices does not allow G_n to be applied for $n = 2$. Graph G_2 can be decomposed as

$$\begin{aligned} G_2 &= -((s_{2n}, s_{2n-1}), 0, G_2^{(1)}, G_2^{(2)}) \\ G_2^{(1)} &= I((s_{2n}, s_{2n-2}), 0, G_2^{(3)}) \\ G_2^{(3)} &= I((s_{2n-1}, s_{2n-3}), 0, G_2^{(4)}) \\ G_2^{(4)} &= G_{n-1} \end{aligned}$$

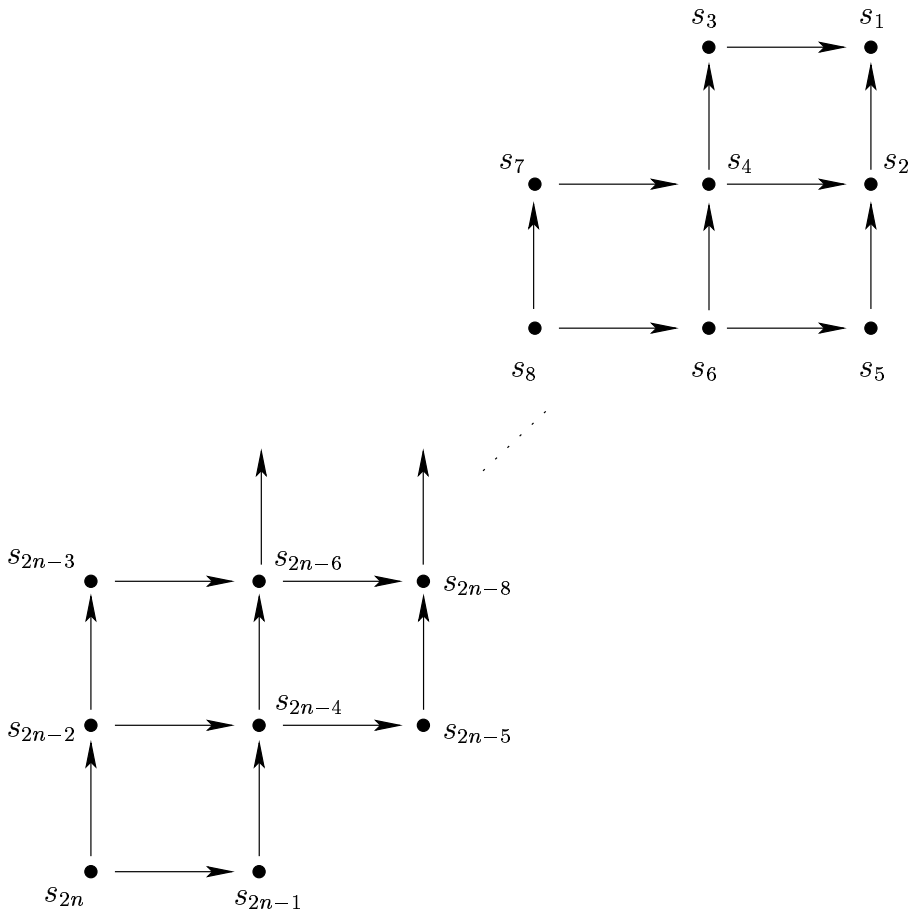


Figure 6.68: Left-shifted 2-rowed plane partitions; constraint graph G_n

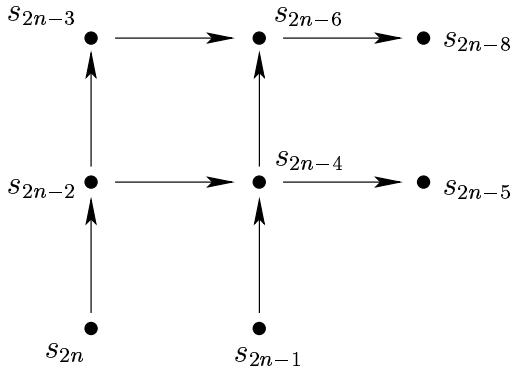


Figure 6.69: Constraint graph $G_n^{(1)}$ for left-shifted 2-rowed plane partitions

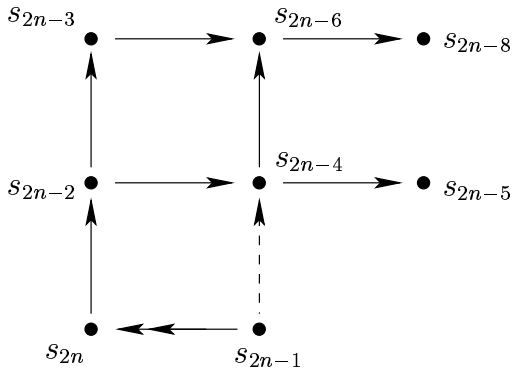


Figure 6.70: Constraint graph $G_n^{(2)}$ for left-shifted 2-rowed plane partitions

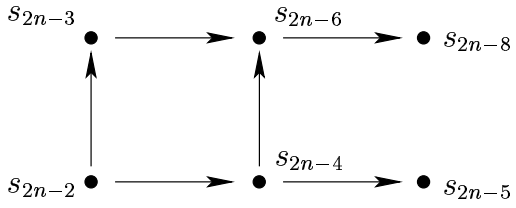


Figure 6.71: Constraint graph G_{n-1} for left-shifted 2-rowed plane partitions

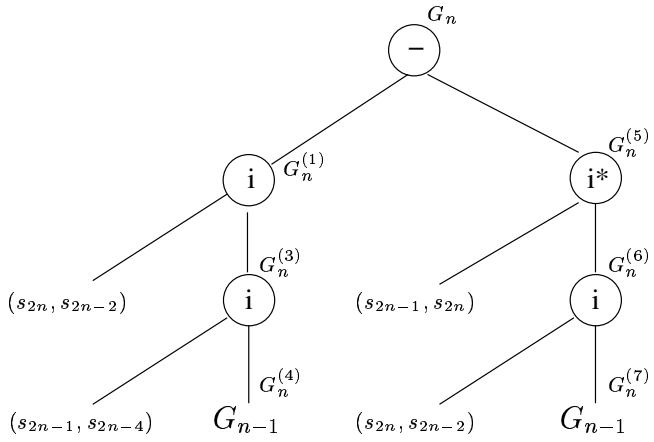


Figure 6.72: Construction tree for G_n of left-shifted 2-rowed plane partitions

Figure 6.73: Constraint graphs G_1 and G_2 for left-shifted 2-rowed plane partitions

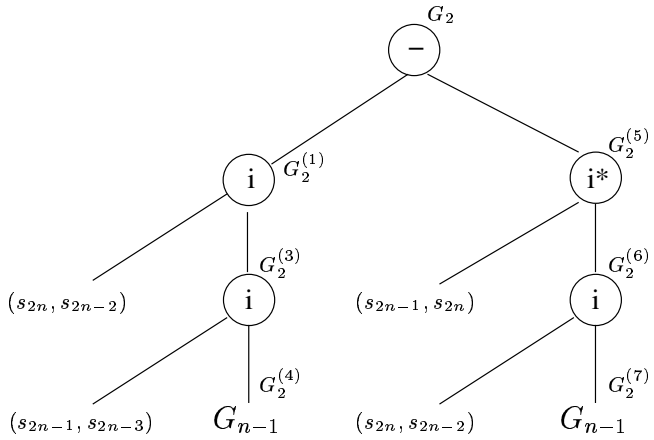


Figure 6.74: Construction tree for G_2 of left-shifted 2-rowed plane partitions

$$
\begin{aligned}
G_2^{(2)} &= R((s_{2n-1}, s_{2n-4}), 0, G_2^{(5)}) \\
G_2^{(5)} &= I((s_{2n-1}, s_{2n}), 1, G_2^{(6)}) \\
G_2^{(6)} &= I((s_{2n}, s_{2n-2}), 0, G_2^{(7)}) \\
G_2^{(7)} &= G_{n-1}
\end{aligned}
$$

and the corresponding construction tree is shown in Figure 6.74. Base case G_1 is simply the edge (s_{2n}, s_{2n-1}).

We invoke procedure `Recurrence` as

```
GFPartitions[Recurrence]([
    ["-",
        "i", x[2*n], x[2*n-2],
        "i", x[2*n-1], x[2*n-4], "%",
        "i*", x[2*n-1],x[2*n],
        "i", x[2*n], x[2*n-2], "%"],

    n=1,
    ["i", x[2*n], x[2*n-1], x[2*n-1]],
    n=2,
    ["-",
        "i", x[2*n], x[2*n-2],
        "i", x[2*n-1], x[2*n-3], "%",
        "i*", x[2*n-1],x[2*n],
        "i", x[2*n], x[2*n-2], "%"]],
    [s,t])
```

and the output generated is

```
========================  Multi-variable recurrence  ========================
```

```
Case "default"
```

$$\frac{R_8}{(1-x_{2\,n-1})\,(1-x_{2\,n})} - \frac{R_{15}x_{2\,n-1}}{(1-x_{2\,n}x_{2\,n-1})\,(1-x_{2\,n-1})}$$

```
        R[15] -- F(n-1)
                 x[2*n-2] <--- x[2*n-2]*x[2*n]*x[2*n-1]
        R[8] -- F(n-1)
                 x[2*n-2] <--- x[2*n-2]*x[2*n]
                 x[2*n-4] <--- x[2*n-4]*x[2*n-1]
Case n = 1
```

$$\frac{1}{(1-x_{2\,n}x_{2\,n-1})\,(1-x_{2\,n})}$$

```
Case n = 2
```

$$\frac{R_8}{(1-x_{2\,n-1})\,(1-x_{2\,n})} - \frac{R_{15}x_{2\,n-1}}{(1-x_{2\,n}x_{2\,n-1})\,(1-x_{2\,n-1})}$$

```
        R[15] -- F(n-1)
                 x[2*n-2] <--- x[2*n-2]*x[2*n]*x[2*n-1]
        R[8] -- F(n-1)
                 x[2*n-3] <--- x[2*n-3]*x[2*n-1]
                 x[2*n-2] <--- x[2*n-2]*x[2*n]
======================== Finite-variable recurrence ========================
            x[2*n-2] --> r
            x[2*n-1] --> s
            x[2*n] --> t
```

$$H(n,q,r,s,t)$$

```
if n = 1 then
```

$$\frac{1}{(1-ts)\,(1-t)}$$

```
elif n = 2 then
```

$$\frac{H\,(n-1,q,q,qs,rt)}{(1-s)\,(1-t)} - \frac{H\,(n-1,q,q,q,rts)\,s}{(1-ts)\,(1-s)}$$

```
else
```

$$\frac{H\,(n-1,q,qs,q,rt)}{(1-s)\,(1-t)} - \frac{H\,(n-1,q,q,q,rts)\,s}{(1-ts)\,(1-s)}$$

```
fi;
```

Chapter 7

Conclusion

In this thesis, we have formulated the constraint graph decomposition technique and the Seven Rules as a new set of tools for building generating functions for sequences of integers defined by directed graphs. We further automated the construction of generating functions and their recurrences in the `GFPartitions` package. Thus equipped, we tackled several well-known problems in a much simpler and more powerful way than done earlier. In this chapter, we summarize our contributions and consider future directions for research.

7.1 Summary of contributions

Our contributions in this thesis include the following.

- *Seven Rules*: We proposed the Seven Rules of decomposition for the construction of generating functions and recurrences for constraint graphs.
- *Sufficiency proof*: We proved the sufficiency of the technique for the decomposition of any constraint graph with edge weights 0 or 1.
- *Special variable rules*: We proposed a technique for the formulation of finite-variable generating function recurrences from their corresponding multi-variable ones.
- *Automation tool `GFPartitions`*: We designed and created a Maple package for the automation of the construction of generating functions and recurrences for a family of constraint graphs given the construction tree of its decomposition.
- *Generating function recurrences*: We illustrated the elegance and ease of procuring generating function recurrences for several well-known problems,
  - 2-rowed plane partitions (with diagonals and without),
  - plane partition hexagonals (with diagonals and without),
  - Gordon sequences,

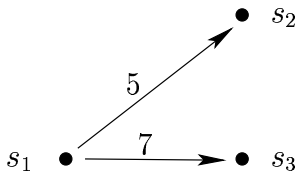


Figure 7.1: A constraint graph which the Seven Rules cannot handle

- 1-compositions,
- up-down sequences and
- plane partition diamonds,

as well as some new ones,

- 2-rowed plane partitions with adjacent diagonals and
- left-shifted 2-rowed plane partitions.

We thus find that restricting the system of constraints addressed allows us to build a set of tools and techniques that are simple to use, yet powerful enough to handle several interesting problems. The intuitiveness of the approach and the automation of its tedious aspects elevates its appeal even further. While not guaranteed to produce recurrences for all problems in the considered domain, the technique proves significant if the recurrences are indeed obtained. At the very least, the tools and techniques we developed accelerate research by enabling speedier investigations of interesting problems in the field of enumerating partitions and compositions.

7.2 Open problems and future directions

While this thesis does make an appreciable contribution to the area of combinatorics under consideration, several problems remain open and unsolved. We present here a list of some interesting questions and possible directions for future research.

- How would we need to upgrade the Seven Rules to guarantee a decomposition for constraint graphs where edge weights are not restricted to 0 and 1? A simple example of a constraint graph which the Seven Rules cannot handle (but the Five Guidelines easily can) is shown in Figure 7.1.
- The Seven Rules, although sufficient for obtaining generating functions for constraint graphs with edge weights 0 and 1, does not guarantee *recurrences* for them. A simple example of a constraint graph sequence for which we found no recursive decomposition (using the Seven Rules) is shown in Figure 7.2. Note that the inclusion of a new rule that mirrored the fourth of the Five Guidelines by permitting the introduction of new

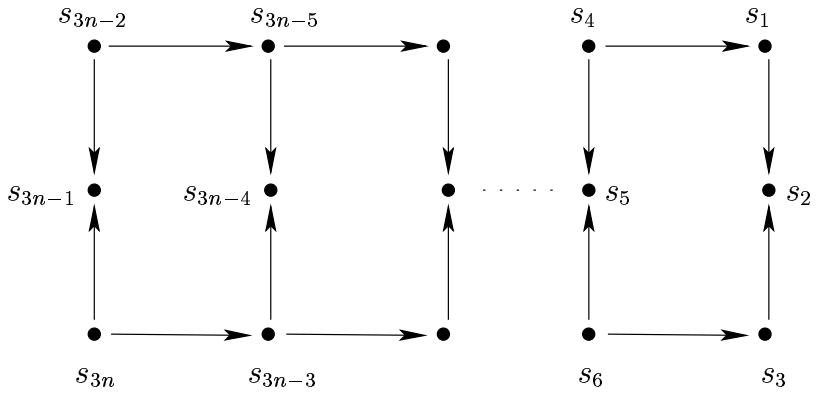


Figure 7.2: A constraint graph with no recursive decomposition (using the Seven Rules)

edges into the graph would solve this issue. What other rules may we additionally need to incorporate in order to guarantee recurrences, if that is at all possible?

- How would our techniques and rules need to be modified if some vertices of the constraint graph represent integer constants instead of variables?
- Is it possible to extend the constraint graph technique to handle non-integer edge weights?
- Some famous problems, such as the family of k-rowed plane partitions [27], are defined over more than one variable (n and k in the case of k-rowed plane partitions). This prevents us from obtaining finite-variable recurrences because a non-finite selection of special variables becomes necessary. Can we extend our technique to obtain recurrences for such 2-dimensional families?
- While the `GFPartitions` package successfully produces multi-variable and finite-variable recurrences given a decomposition as input, can we develop an efficient algorithm to obtain an optimal decomposition for a given sequence of constraint graphs?
- How do we automate the Five Guidelines technique?
- How do we automate the solution of recurrences, where possible?

We thus note that there remains much scope for further investigation in future research.

Bibliography

[1] G. E. Andrews, Peter Paule, and Axel Riese. MacMahon's partition analysis. X. Plane partitions with diagonals. *South East Asian J. Math. Math. Sci.*, 3(1):3–14, 2004.

[2] G. E. Andrews, Peter Paule, and Axel Riese. MacMahon's partition analysis. XI. hexagonal plane partitions. 2004.

[3] George E. Andrews. MacMahon's partition analysis. I. The lecture hall partition theorem. In *Mathematical essays in honor of Gian-Carlo Rota (Cambridge, MA, 1996)*, volume 161 of *Progr. Math.*, pages 1–22. Birkhäuser Boston, Boston, MA, 1998.

[4] George E. Andrews. MacMahon's partition analysis. II. Fundamental theorems. *Ann. Comb.*, 4(3-4):327–338, 2000. Conference on Combinatorics and Physics (Los Alamos, NM, 1998).

[5] George E. Andrews, Peter Paule, and Axel Riese. MacMahon's partition analysis. IX. k-gon partitions. *Bull. Austral. Math. Soc.*, 64(2):321–329, 2001.

[6] George E. Andrews, Peter Paule, and Axel Riese. MacMahon's partition analysis: the Omega package. *European J. Combin.*, 22(7):887–904, 2001.

[7] George E. Andrews, Peter Paule, and Axel Riese. MacMahon's partition analysis. VI. A new reduction algorithm. *Ann. Comb.*, 5(3-4):251–270, 2001. Dedicated to the memory of Gian-Carlo Rota (Tianjin, 1999).

[8] George E. Andrews, Peter Paule, and Axel Riese. MacMahon's partition analysis. VII. Constrained compositions. In *q-series with applications to combinatorics, number theory, and physics (Urbana, IL, 2000)*, volume 291 of *Contemp. Math.*, pages 11–27. Amer. Math. Soc., Providence, RI, 2001.

[9] George E. Andrews, Peter Paule, and Axel Riese. MacMahon's partition analysis. VIII. Plane partition diamonds. *Adv. in Appl. Math.*, 27(2-3):231–242, 2001. Special issue in honor of Dominique Foata's 65th birthday (Philadelphia, PA, 2000).

[10] George E. Andrews, Peter Paule, Axel Riese, and Volker Strehl. MacMahon's partition analysis. V. Bijections, recursions, and magic squares. In *Algebraic combinatorics and applications (Gößweinstein, 1999)*, pages 1–39. Springer, Berlin, 2001.

[11] Mireille Bousquet-Mélou and Kimmo Eriksson. Lecture hall partitions. *Ramanujan J.*, 1(1):101–111, 1997.

[12] Mireille Bousquet-Mélou and Kimmo Eriksson. Lecture hall partitions. II. *Ramanujan J.*, 1(2):165–185, 1997.

[13] Mireille Bousquet-Mélou and Kimmo Eriksson. A refinement of the lecture hall theorem. *J. Combin. Theory Ser. A*, 86(1):63–84, 1999.

[14] L. Carlitz. Enumeration of up-down sequences. *Discrete Math.*, 4:273–286, 1973.

[15] A. Cayley. On a problem in the partition of numbers. *Philosophical Mag.*, 13:245–248, 1857.

[16] Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2:137–167, 1959.

[17] Sylvie Corteel. Conjecture for 2-rowed plane partitions with adjacent diagonals. *Private communication.*

[18] Sylvie Corteel, Sunyoung Lee, and Carla D. Savage. Five guidelines for partition analysis with applications to lecture hall-type theorems. 2005.

[19] Sylvie Corteel and Carla D. Savage. Plane partition diamonds and generalizations. *Integers*, 3:A9, 8 pp. (electronic), 2003.

[20] Sylvie Corteel and Carla D. Savage. Partitions and compositions defined by inequalities. *Ramanujan J.*, 8(3):357–381, 2004.

[21] Sylvie Corteel, Carla D. Savage, and Herbert S. Wilf. A note on parititons and compositions defined by inequalities. *Submitted to the electronic journal INTEGERS*, 2005.

[22] Basil Gordon. A combinatorial generalization of the Rogers-Ramanujan identities. *Amer. J. Math.*, 83:393–399, 1961.

[23] Dick Grune and Ceriel J. H. Jacobs. *Parsing techniques: a practical guide.* Ellis Horwood, Upper Saddle River, NJ, USA, 1990.

[24] D. R. Hickerson. A partition identity of the Euler type. *Amer. Math. Monthly*, 81:627–629, 1974.

[25] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley Publishing Co., Reading, Mass., 1979. Addison-Wesley Series in Computer Science.

[26] Arnold Knopfmacher and Helmut Prodinger. On Carlitz compositions. *European J. Combin.*, 19(5):579–589, 1998.

[27] Percy A. MacMahon. *Combinatory analysis.* Two volumes (bound as one). Chelsea Publishing Co., New York, 1960.

[28] Percy Alexander MacMahon. *Collected papers. Vol. I.* MIT Press, Cambridge, Mass., 1978. Combinatorics, Mathematicians of Our Time, Edited and with a preface by George E. Andrews, With an introduction by Gian-Carlo Rota.

[29] Russell Merris. *Combinatorics.* Wiley-Interscience Series in Discrete Mathematics and Optimization. Wiley-Interscience [John Wiley & Sons], New York, second edition, 2003.

[30] Ivan Niven. Formal power series. *J. Combin. Inform. System Sci.*, 76(8):871–889, 1969.

[31] Fred S. Roberts. *Applied combinatorics, Second Edition.* Prentice Hall Inc., Englewood Cliffs, NJ, 2005.

[32] Jose Plínio O. Santos. On a new combinatorial interpretation for a theorem of Euler. *Adv. Stud. Contemp. Math. (Pusan)*, 3(2):31–38, 2001.

[33] Carla D. Savage. Partitions and compositions defined by digraphs. *In preparation.*

[34] James A. Sellers. Extending a recent result of Santos on partitions into odd parts. *Integers*, 3:A4, 5 pp. (electronic), 2003.

[35] James A. Sellers. Corrigendum to: "Extending a recent result of Santos on partitions into odd parts" [Integers **3** (2003), A4, 5 pp. (electronic); mr 1985666]. *Integers*, 4:A8, 1 pp. (electronic), 2004.

[36] Kenichi Taniguchi and Tadao Kasami. Reduction of context-free grammars. *Information and Control*, 17:92–108, 1970.

[37] Stephen H. Unger. A global parser for context-free phrase structure grammars. *Commun. ACM*, 11(4):240–247, 1968.

[38] Douglas B West. *Introduction to Graph Theory.* Prentice Hall, 2nd edition, 2001.

[39] Herbert S. Wilf. *generatingfunctionology.* Academic Press Inc., Boston, MA, second edition, 1994.

Appendix A

Program listing for `GFPartitions` (Maple 9.0)

```
GFPartitions := module() export Recurrence; local B, G, ListWrap;

Recurrence := proc(treeList::list, specialVariables::list)
    local f, i, fR, fTable, fRTable, condTable, vRtable, Hf, t, nT, nTi,
        CBaseProc, substTable, subst::table, sub, subList, max, min, vMin,
        allVars::table, SV::table, sVar, varIndList, varInd, indList, ind,
        svIndList, sVarList, svOp, paramList, pList, pHolder, sNewVar,
        svRewind, valueList, subI;

    vRtable := table();
    allVars := table();

    nT:=nops(treeList);

    # default case
    condTable[1] := "default";
    substTable[1] := table();
    (fTable[1], i, fRTable[1]) :=
                G(treeList[1], 1, vRtable, substTable[1], allVars);
    nTi := 1;

    # parse all other cases
    for t from 2 to nT by 2 do
        nTi := nTi + 1;
```

```
        condTable[nTi] := treeList[t];
        substTable[nTi] := table();
        (fTable[nTi], i, fRTable[nTi]) :=
                G(treeList[t+1], 1, vRtable, substTable[nTi], allVars);
    od;

    nT := nops([indices(condTable)]);

    # -------- Create the multi-variable g.f. B
    # define C
    CBaseProc := proc(t::integer, vtable::table, n_::integer)
        if eval(condTable[t], n=n_) then
            return fTable[t](vtable, n_);
        elif t<nT then
            return CBaseProc(t+1, vtable, n_);
        else
            return fTable[1](vtable, n_);
        fi;
    end proc;

    # define B
    B := proc(vtable::table, n::integer)
            return CBaseProc(2, vtable, n);
    end proc;

    # Finding index of max element
    varIndList := [indices(allVars)];
    max := op(varIndList[1][1]);
    for varInd in varIndList do
        if op(varInd[1])-max>0 then max:=op(varInd[1]); fi;
    od;

    printf ("\n============================");
    printf ("  Multi-variable recurrence  ");
    printf ("============================\n\n");
```

```
for t from 1 to nT do
    printf("Case %a", condTable[t]);
    print(fRTable[t]);
    subList := [indices(substTable[t])];
        for subI in subList do                # for each place holder
            vRtable := substTable[t][subI[1]];
            indList := [indices(vRtable)];
            printf("\t    %a -- F(n-%a)\n", subI[1], vRtable["levelJump"]);
            for ind in indList do        # for each substitution
                if ind[1]<>"newVars"
                    and ind[1]<>"levelJump"
                    and ind[1]<>vRtable[ind[1]]
                    then
                        sVar := eval(ind[1], n=n+vRtable["levelJump"]);
                        if max-op(sVar)>=0 then
                            printf("\t\t\t     %a <--- ", ind[1]);
                            printf("%a\n", vRtable[ind[1]]);
                        fi;
                fi;
            od;
        od;
od;

printf ("\n==========================");
printf ("  Finite-variable recurrence  ");
printf ("==========================\n\n");

# ---------- Obtain and display finite-variable recurrence
# ---- Step 1

# Finding Special Variables - Round 1
SV := table();
for t from 1 to nT do                  # for each case
        subList := [entries(substTable[t])];
        for sub in subList do                # for each place holder
```

```
            vRtable := sub[1];
                indList := [indices(vRtable)];
            for ind in indList do        # for each substitution
                if ind[1]<>"newVars"
                    and ind[1]<>"levelJump"
                    and ind[1]<>vRtable[ind[1]]
                    then
                        sVar := eval(ind[1], n=n+vRtable["levelJump"]);
                        if max-op(sVar)>=0 then
                        SV[sVar] := sVar;
                        fi;
                fi;
                od;
        od;
od;

# Finding Special Variables - Round 2
svIndList := [indices(SV)];
min := op(svIndList[1][1]);
vMin := svIndList[1][1];
for sVar in svIndList do
    svOp := op(sVar[1]);
    if min-svOp>0 then min:=svOp; vMin:= sVar[1]; fi;
    for i from 1 while (max-subs(n=n+i,svOp) >= 0) do
        sNewVar := subs(n=n+i, sVar[1]);
        if not(assigned(SV[sNewVar])) then SV[sNewVar] := sNewVar; fi;
    od;
od;

# ---- Step 2

# get an ordered list of the special variables
sVarList := [];
for i from 0 to (max-min) do
    sVar := subs(op(vMin)=min+i, vMin);
```

```
        if assigned(SV[sVar]) then
            sVarList := [op(sVarList), sVar];
        fi;
    od;

    paramList := [n, q, op(sVarList)];

    # "Expand" the placeholders
    unassign(H);
    for t from 1 to nT do
        pList := [indices(substTable[t])];
        for pHolder in pList do
            vRtable := substTable[t][pHolder[1]];
            valueList := [];
            for sVar in sVarList do
                svRewind := eval(sVar, n=n-vRtable["levelJump"]);
                if not(assigned(vRtable[svRewind])) then
                    vRtable[svRewind]:=svRewind;
                        allVars[svRewind]:=svRewind;
                fi;
                valueList:=[op(valueList), vRtable[svRewind]];
            od;
            Hf := apply(H, n-vRtable["levelJump"], q, op(valueList));
            fRTable[t] := subs(pHolder[1]=Hf, fRTable[t]);
        od;
    od;

    # ---- Step 3

    # replace non-SVs with q
    varIndList := [indices(allVars)];
    for varInd in varIndList do
        if not(assigned(SV[varInd[1]])) then
            for t from 1 to nT do
                fRTable[t] := subs(varInd[1]=q, fRTable[t]);
```

```
        od;
    fi;
od;

if nops(specialVariables)<nops(sVarList) then
        printf("Insufficient special variables. Needed %d", nops(sVarList));
        printf(" but received %d.\n", nops(specialVariables));
        print(apply(H, n, q, op(sVarList)));
        for t from 2 to nT do
            printf("if %a then\n", condTable[t]);
            print(fRTable[t]);
            printf("el");
        od;
        printf("se ");
            print(fRTable[1]);
        printf("fi;");
    return B;
elif nops(specialVariables)>nops(sVarList) then
    printf("Extra special variables discarded. Needed ");
    printf("%d but received %d.\n", nops(sVarList), nops(specialVariables));
fi;

for i from 1 to nops(sVarList) do
        printf("            %a --> %a\n",sVarList[i],specialVariables[i]);
od;

# ---- Step 4, 5
# unapplying and applying user's variables, and displaying
print(apply(H, n, q, op(1..nops(sVarList),specialVariables)));
for t from 2 to nT do
    printf("if %a then\n", condTable[t]);
    print(apply(unapply(fRTable[t], paramList), n, q, op(specialVariables)));
    printf("el");
od;
printf("se ");
```

```
    print(apply(unapply(fRTable[1], paramList), n, q, op(specialVariables)));
    printf("fi;");

    return ListWrap(B);
end proc;

G := proc(tree::list, j::integer, vRtable::table, substs::table, allVars::table)
    local t, a, b, f, f2, fSub, m, i, ir, fR, fR1, fR2, ftype, cVRtable, lJump;
  i := j;
  t := tree[i];
  i := i+1;
  if t="i" or t="i*" or t="I" then
        # add edge a->b
        (a,b) := (tree[i],tree[i+1]);

        m := 0;
        if t="i*" then
            m := 1;
        elif t="I" then
            m := tree[i+2];
            i := i+1;
        fi;

        if not(assigned(vRtable[a])) then vRtable[a]:=a; fi;
        if not(assigned(vRtable[b])) then vRtable[b]:=b; fi;
        allVars[a] := a; # keeping track of all variables ever referenced
        allVars[b] := b;

        vRtable[b] := vRtable[b] * vRtable[a];
        (fSub, i, fR) := G(tree, i+2, vRtable, substs, allVars);
        vRtable[b] := vRtable[b] / vRtable[a];

        fR := fR * vRtable[a]^m / (1-vRtable[a]);

        f := proc(vtable::table, n_::integer)
```

```
            local a1, b1, fS;
            a1 := eval(a, n=n_);
            b1 := eval(b, n=n_);
            if not(assigned(vtable[a1])) then vtable[a1]:=a1; fi;
            if not(assigned(vtable[b1])) then vtable[b1]:=b1; fi;
            vtable[b1] := vtable[b1] * vtable[a1];
            fS := fSub(vtable, n_) * vtable[a1]^m / (1-vtable[a1]);
            vtable[b1] := vtable[b1] / vtable[a1];
            return fS;
            end proc;
    elif t="-" then
        (fSub, i, fR1) := G(tree, i, vRtable, substs, allVars); # read graph 1
        (f, i, fR2) := G(tree, i, vRtable, substs, allVars); # subtract graph 2
        f := fSub - f;
        fR := fR1 - fR2;
    elif t="+" then
        (fSub, i, fR1) := G(tree, i, vRtable, substs, allVars); # read graph 1
        (f, i, fR2) := G(tree, i, vRtable, substs, allVars); # add graph 2
        f := fSub + f;
        fR := fR1 + fR2;
    elif t="." then
        a := tree[i];
        (fSub, i, fR) := G(tree, i+1, vRtable, substs, allVars);
        f := proc(vtable::table, n_::integer)
            local a1, fS;
            a1 := eval(a, n=n_);
                if not(assigned(vtable[a1])) then vtable[a1]:=a1; fi;
            fS := fSub(vtable, n_) / (1-vtable[a1]);
            return fS;
           end proc;
        if not(assigned(vRtable[a])) then vRtable[a]:=a; fi;
        allVars[a] := a;
        fR := fR / (1-vRtable[a]);
    elif t="%" or t="%%" then
        if t="%" then
```

```
            # G_n-1
            lJump := 1;
        elif t="%%" then
            # G_n-k
            lJump := tree[i];
            i := i + 1;
        fi;
        f := (vtable::table, n::integer) -> B(vtable, n-lJump);
        ftype := F[n-lJump];
        fR := R[j];
        cVRtable := copy(vRtable);
        cVRtable["levelJump"]:=lJump;
        substs[R[j]] := cVRtable;
    else # t is a single node
        f := proc(vtable::table, n_::integer)
            local a1, fS;
            a1 := eval(t, n=n_);
                #print("sn",a1);
                if not(assigned(vtable[a1])) then vtable[a1]:=a1; fi;
            fS := 1 / (1-vtable[a1]);
            return fS;
           end proc;
        if not(assigned(vRtable[t])) then vRtable[t]:=t; fi;
        allVars[t] := t;
        fR := 1 / (1-vRtable[t]);
  end if;
  return (f, i, fR);
end proc;

ListWrap := proc(B2::procedure)
    local GFInstantiator;
    GFInstantiator := proc(n::integer)
        local vtable, vList, b3, var, itable, max,i;

        # obtain generating equation
```

```
        vtable := table();
        b3 := simplify(B2(vtable, n));

        # find and sort parameter variables
        vList := [indices(vtable)];
        max := -1;
        for var in vList do
                itable[op(var[1])] := var[1];
                if (op(var[1])>max) then max := op(var[1]); fi;
        od;
        vList := [];
        for i from 0 to max  do
            if (assigned(itable[i])) then
                vList := [op(vList),itable[i]];
            fi;
        od;
        unapply(b3, vList);
    end proc;

    return GFInstantiator;
end proc;

end module;
```