AutoGF: An Automated System to Calculate Coefficients of Generating Functions

by

Huantian Cao

(Under the direction of Robert W. Robinson)

Abstract

AutoGF, an automated system to calculate coefficients of generating functions, is developed in this thesis. The algorithms for power series manipulations such as addition, substraction, multiplication, division, powers, exponentiation, logorithm, reversion and composition are implemented in AutoGF. Maclaurin series provide the basis for calculating the coefficients of basic generating functions. User defined generating functions are also provided for. By writing a text file listing the algebraic steps defining a target generating function, the user of AutoGF can obtain the coefficients of very complicated generating functions.

Exponential generating functions are widely used to count labeled graphs and digraphs. By applying a Hadamard product to the coefficients of exponential generating functions, the exact numbers of labeled graphs or digraphs with different numbers of vertices can be obtained under various restrictions using AutoGF. This is demonstrated for connected graphs, blocks, connected eulerian graphs, acyclic digraphs, strong digraphs, and forests. The time performances of these graphical enumerations are investigated and their time complexities are estimated.

Generating functions can also be applied to count integer partitions. The numbers of all partitions and of partitions into distinct parts are calculated by using AutoGF. The time performances of these partition enumerations are investigated and their time complexities are estimated.

Index words: Generating Function, Exponential Generating Function, Power Series, Graph Enumeration, Integer Partitions, Python

AutoGF: An Automated System to Calculate Coefficients of Generating Functions

by

Huantian Cao

B.S., China Textile University, China, 1994

M.S., China Textile University, China, 1997

Ph.D., The University of Georgia, 2000

A Thesis Submitted to the Graduate Faculty

of The University of Georgia in Partial Fulfillment

of the

Requirements for the Degree

Master of Science

Athens, Georgia

2002

AutoGF: An Automated System to Calculate Coefficients of Generating Functions

by

Huantian Cao

Approved:

Major Professor: Robert W. Robinson

Committee: E. Rodney Canfield
David Gries

Electronic Version Approved:

Gordhan L. Patel
Dean of the Graduate School
The University of Georgia
May 2002

Acknowledgments

The author would like to express his deep appreciation to his major professor, Dr. Robert W. Robinson, for his remarkable encouragement, support, guidance, assistance, and kindness. Without Dr. Robinson's advice and effort, completion of this thesis would have been impossible.

The author is thankful to Dr. E. Rodney Canfield and Dr. David Gries for serving on his thesis committee. The author expresses his gratitude to Dr. Michael A. Covinton for providing LaTeX style sheet `uga.sty` to make it easier to type this thesis. The author appreciates the faculty, staff, and graduate students of the Department of Computer Science for their cooperation and help.

The author is thankful for the financial assistance provided by the Deaprtment of Computer Science and the Department of Textiles, Merchandising and Interiors of the University of Georgia.

Table of Contents

List of Tables

Chapter 1

Introduction

1.1 Generating Functions

A generating function is a power series whose coefficients are a sequence of numbers, $a_0, a_1, a_2, \cdots$ (Wilf 1990, p. 1). That is, $g(x)$ is a *generating function* for a_r if $g(x)$ has the polynomial expansion

$$g(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_r x^r + \cdots + a_n x^n.$$

If the function has an infinite number of terms, it is called a *power series* (Tucker 1995, p. 243). In the application of generating functions, it does not matter whether the power series converges, all power series discussed in this thesis are *formal power series* meaning that we are not concerned with questions of convergence (Brent and Kung 1978).

For a problem whose answer is a sequence of numbers, $a_0, a_1, a_2, \cdots$, an explicit formula such as

$$a_n = n^2 + 2n + 1,$$

or a recurrence relation such as

$$F_{n+1} = F_n + F_{n-1}, (n \geq 1; F_0 = 0, F_1 = 1)$$

may be available to calculate the terms. However, there are situations where a simple formula or a recurrence relation does not exist or is hard to obtain. In these situations, relations involving generating functions may be another useful way to deal with the sequence of numbers problem.

For example, we have the Fibonacci numbers $F_0, F_1, F_2, \cdots$, where

$$F_{n+1} = F_n + F_{n-1}, (n \geq 1; F_0 = 0, F_1 = 1).$$

The sequence begins with $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \cdots$. But in addition to the recurrence relation, we can also use the generating function

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

to represent the sequence of Fibonacci numbers. That is, the *nth* Fibonacci number, F_n, is the coefficient of x^n in the expansion of the function $x/(1-x-x^2)$ as a power series about $x = 0$.

The generating functions discussed above are called *ordinary generating functions*. They are used to model selection with repetition problems. There is another category of generating functions called *exponential generating functions*, which are usually used to model and solve problems involving arrangements and distributions of distinct objects. An *exponential generating function* $g(x)$ for a_r is a function with the power series expansion (Tucker 1995, p. 267):

$$g(x) = a_0 + a_1 x + a_2 \frac{x^2}{2!} + a_3 \frac{x^3}{3!} + \cdots + a_r \frac{x^r}{r!} + \cdots$$

Exponential generating functions are defined in the same way as ordinary generating functions except that each power x^r is divided by $r!$. For a given expanded exponential generating function, a_r can be obtained by the operation of the Hadamard product (@), which is defined as

$$< f_i > @ < g_i > = < f_i g_i >$$

Like exact formulas and recurrence relations, generating functions allow the user to do almost anything he wants to do with a sequence. In many cases, the generating functions are elegant, simple and easy to handle when exact formulas and recurrence relations are very complicated or very difficult to find.

In his book *Generatingfunctionology*, Herbert S. Wilf described things that generating functions can do (Wilf 1990, p. 2).

(a) *Find an exact formula for the sequence numbers.* Generating functions cannot guarantee to find the exact formula, but they provide a powerful way to find exact formulas in some cases.

(b) *Find a recurrence relation.* Usually generating functions are obtained from recurrence relations. However, from a generating function, the user may derive a new recurrence relation and gain new insight into the nature of the sequence.

(c) *Find the average and other statistical properties of the sequence.* Generating functions can provide quick ways to calculate probabilistic aspects of the problem represented by a sequence of numbers.

(d) *Find an asymptotic formula for the sequence.* For some cases, it is impossible to derive an the exact formula for a sequence, but an approximate formula may be obtained by using generating functions.

(e) *Prove unimodality, convexity, etc.* A sequence is called unimodal if it increases steadily at first and then decreases steadily. By analyzing a generating function, one may be able to analyze the rises and falls of the sequence of coefficients.

(f) *Prove identities.* The identities referred to here are those in which a certain formula is asserted to be equal to another formula for stated values of the free variables. To prove this, we can define a generating function whose coefficients are the sequence shown on the left side of the identity and a generating function whose coefficients are the sequence shown on the right side of the identity and show that they are the same function.

(g) *Other.* Generating functions may offer a way for the user who wants to know something else about the sequence. For example, if a generating function has a striking resemblance to some other known generating functions. This may lead the user to discover that the problem is closely related to another one.

1.2 Purpose of This Study

In the application of generating functions, usually a generating function is obtained as a result for a sequence of numbers. However, for a generating function, like

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

it is highly desired to know the sequence numbers it represents. That is, we want to know $a_0, a_1, a_2, \cdots, a_r, \cdots$, such that

$$\frac{x}{1 - x - x^2} = a_0 + a_1 x + a_2 x^2 + \cdots + a_r x^r + \cdots.$$

Because computer memory is finite, the user must specify a bound N so that coefficients are calculated and stored only for powers less than N.

The purpose of the study is to develop an automatic system, called `AutoGF`, to calculate coefficients of generating functions. The user of `AutoGF` must analyze the generating function and express it in terms of simple or standard generating functions and operations on them. `AutoGF` can then carry out the manipulation of generating functions and calculate the coefficients. For example, for the generating function

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

we can divide it into two simple generating functions,

$$f(x) = x,$$

whose coefficients are $0, 1, 0, 0, 0, \cdots$ and

$$g(x) = 1 - x - x^2,$$

whose coefficients are $1, -1, -1, 0, 0, \cdots$, and use the division operation

$$F(x) = \frac{f(x)}{g(x)}$$

to express $F(x)$.

Exponential generating functions are widely used in graph enumeration. In this thesis, `AutoGF` is used with exponential generating functions to obtain the numbers of connected labeled graphs, labeled blocks, connected labeled eulerian graphs, labeled acyclic digraphs, labeled strong digraphs, labeled trees, and labeled forests. `AutoGF` is also used with ordinary generating functions to count general integer partitions and partitions into distinct parts. The amounts of time used to obtain these numbers are measured and the time complexities are estimated. These experiments serve as test cases for `AutoGF` since all of these numbers had been calculated previously.

Chapter 2

Algorithms For Manipulating Generating Functions

2.1 Rational Arithmetic

To calculate quantities defined by generating functions, it is desirable to calculate the coefficients exactly rather than as floating point values. To accomplish this, coefficients are represented as rational numbers with no preset limits on the size of the numerator or denominator. Rational numbers can be represented as pairs of integers (u/u'), where $u' \neq 0$. The operations on two rational numbers (u/u') (for $u' \neq 0$) and (v/v') (for $v' \neq 0$) are as follows:

$$\frac{u}{u'} \pm \frac{v}{v'} = \frac{uv' \pm u'v}{u'v'},$$
$$\frac{u}{u'} \times \frac{v}{v'} = \frac{uv}{u'v'},$$
$$\frac{u}{u'} \div \frac{v}{v'} = \frac{uv'}{u'v} (v' \neq 0).$$

If the numerator a and denominator a' of the result are not relative prime, it is usually best for efficiency to compute the greatest common divisor $d = gcd(a, a')$ and reduce the fraction to w/w' where $w = a/d$ and $w' = a'/d$. The *modern Euclidean algorithm* (Knuth 1998, Algorithm A, p. 337), described below, is used to determine greatest common divisor in this thesis.

Given nonnegative integers u and v, this algorithm finds their greatest common divisor. The greatest common divisor of arbitrary integers u and v may be obtained by applying the algorithm to $|u|$ and $|v|$ because

$$gcd(u, v) = gcd(v, u)$$

and

$$gcd(u, v) = gcd(-u, v)$$

1. [$v = 0$?] If $v = 0$, the algorithm terminates with u as the answer.

2. [Take u mod v.] Set $r \leftarrow u \bmod v$, $u \leftarrow v$, $v \leftarrow r$, and return to step 1. Application of step 2 decreases the value of v but leaves $gcd(u, v)$ unchanged.

2.2 Expansions of Basic Functions

For efficiency, it is helpful for a generatingfunctionologist to have available a reference list of known power series and other series that occur frequently in applications. In our system for automatically calculating coefficients of complicated generating functions, these basic power series serve as bricks for the building since many complicated generating functions are obtained from the manipulations of these basic power series.

The Maclaurin series generated by a function f, which is the Taylor series for f at $x = 0$, can often be used to expand functions and produce useful power series (Finney et al. 2000, p. 671). The Maclaurin series generated by the function f is

$$\sum_{k=0}^{\infty} \frac{f^{(k)}(0)}{k!}x^k = f(0) + f'(0)x + \frac{f''(0)}{2!}x^2 + \cdots + \frac{f^{(n)}(0)}{n!}x^n + \cdots.$$

For example, we can use a Maclaurin series to expand e^x as follows. Since

$$f(x) = e^x, f'(x) = e^x, \cdots, f^{(n)}(x) = e^x, \cdots,$$

we have

$$f(0) = e^0 = 1, f'(0) = e^0 = 1, \cdots, f^{(n)}(0) = e^0 = 1, \cdots.$$

The Maclaurin series generated by $f(x) = e^x$ is

$$f(0) + f'(0)x + \frac{f''(0)}{2!}x^2 + \cdots + \frac{f^{(n)}(0)}{n!}x^n + \cdots = 1 + x + \frac{1}{2!}x^2 + \cdots + \frac{1}{n!}x^n + \cdots.$$

That is,

$$e^x = 1 + x + \frac{1}{2!}x^2 + \cdots + \frac{1}{n!}x^n + \cdots.$$

Here is a list of basic power series used in this study.

$$(1+x)^n = 1 + nx + \frac{n(n-1)}{2}x^2 + \cdots + x^n = \sum_{k=0}^{n} \binom{n}{k} x^k$$

$$e^x = 1 + x + \frac{1}{2!}x^2 + \cdots = \sum_{k=0}^{\infty} \frac{1}{k!}x^k$$

$$\ln(1+x) = x - \frac{1}{2}x^2 + \frac{1}{3}x^3 + \cdots = \sum_{k=0}^{\infty} \frac{(-1)^{k+1}}{k}x^k$$

$$\ln \frac{1}{1-x} = x + \frac{1}{2}x^2 + \frac{1}{3}x^3 + \cdots = \sum_{k=0}^{\infty} \frac{1}{k}x^k$$

$$\frac{1}{1-x} = 1 + x + x^2 + \cdots = \sum_{k=0}^{\infty} x^k$$

$$\frac{1}{1+x} = 1 - x + x^2 - \cdots = \sum_{k=0}^{\infty} (-1)^k x^k$$

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} + \cdots = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{(2k+1)!}$$

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \cdots = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k}}{(2k)!}$$

$$\arctan x = x - \frac{x^3}{3} + \frac{x^5}{5} - \cdots = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{2k+1}$$

$$\sinh x = \frac{e^x - e^{-x}}{2} = x + \frac{x^3}{3} + \frac{x^5}{5} + \cdots = \sum_{k=0}^{\infty} \frac{x^{2k+1}}{(2k+1)!}$$

$$\cosh x = \frac{e^x + e^{-x}}{2} = 1 + \frac{x^2}{2} + \frac{x^4}{4} + \cdots = \sum_{k=0}^{\infty} \frac{x^{2k}}{(2k)!}$$

2.3 Manipulation of Generating Functions

For two generating functions

$$U(x) = U_0 + U_1 x + U_2 x^2 + \cdots$$

and

$$V(x) = V_0 + V_1 x + V_2 x^2 + \cdots,$$

their sum, difference, product, quotient, and composition can be used to express new generating functions. For one generating function, its powers, exponential, logarithm and reversion can also be used to form new generating functions. Because only a finite number of terms can be represented and stored in a computer, we work only with the first N coefficients of any generating function, where N is a user-specified bound. In the following sections, we use $[x^n]W(x)$ to denote the nth coefficient of generating function $W(x)$.

2.3.1 Sum and Difference of Generating Functions

For the sum and difference of two generating functions, $W(x) = U(x) \pm V(x)$, the coefficients of $W(x)$ can be computed termwize (Knuth 1998, p. 525):

$$[x^n](U(x) \pm V(x)) = U_n \pm V_n.$$

2.3.2 Product of Generating Functions

The convolution rule is used to calculate the coefficients of $W(x) = U(x)V(x)$ (Knuth 1998, p. 525):

$$W_n = [x^n]W(x) = \sum_{k=0}^{n} U_k V_{n-k} = U_0 V_n + U_1 V_{n-1} + \cdots + U_n V_0.$$

The termwize or *Hadamard* product is essential to easily create exponential generating functions and obtain exact numbers from them. We use @ to represent Hadamard product: $W(x) = U(x)@V(x)$, where

$$W_n = [x^n]W(x) = U_n V_n.$$

2.3.3 Quotient of Generating Functions

The rule to obtain the quotient $W(x) = U(x)/V(x)$, when $V_0 \neq 0$, is to interchange U and W in the previous formula for multiplication (Knuth 1998, p. 525):

$$W_n = \frac{U_n - \sum_{k=0}^{n-1} W_k V_{n-k}}{V_0} = \frac{U_n - W_0 V_n - W_1 V_{n-1} - \cdots - W_{n-1} V_1}{V_0}.$$

In some cases, the quotient of generating function $W(x) = U(x)/V(x)$ can also be calculated when $V_0 = 0$. Let

$$U(x) = U_k x^k + U_{k+1} x^{k+1} + \cdots$$

and

$$V(x) = V_m x^m + V_{m+1} x^{m+1} + \cdots$$

where U_k is the first non-zero coefficient of $U(x)$ and V_m is the first non-zero coefficient of $V(x)$. Coefficients of generating function $W(x) = U(x)/V(x)$ can be calculated when $k \geq m$ since

$$\frac{U(x)}{V(x)} = \frac{U(x)/x^m}{V(x)/x^m}$$

and the first m zero coefficients of denominator and numerator can be removed.

2.3.4 Powers of Generating Functions

For $W(x) = V(x)^\alpha$, where α is an arbitrary power, and V_m is the first nonzero coefficient of $V(x)$, we have

$$V(x) = V_m x^m (1 + (V_{m+1}/V_m)x + (V_{m+1}/V_m)x^2 + \cdots)$$

and

$$V(x)^\alpha = V_m^\alpha x^{\alpha m} (1 + (V_{m+1}/V_m)x + (V_{m+1}/V_m)x^2 + \cdots)^\alpha.$$

This will be a power series if and only if αm is a nonnegative integer. The operation of computing $W(x) = V(x)^\alpha$, where α is an arbitrary power, can thus be reduced to

the case that $V_0 = 1$ and

$$W(x) = (1 + V_1 x + V_2 x^2 + V_3 x^3 + \cdots)^\alpha$$

Clearly $W_0 = 1^\alpha = 1$.

In 1748, Leonhard Euler published a simple and efficient way to compute powers of power series (Knuth 1998, p. 526). If $W(x) = V(x)^\alpha$, then by differentiation, we get

$$W_1 + 2W_2 x + 3W_3 x^2 + \cdots = W'(x) = \alpha V(x)^{\alpha-1} V'(x)$$

and therefore

$$W'(x)V(x) = \alpha W(x)V'(x).$$

Equating the coefficients of x^{n-1}, we find

$$\sum_{k=0}^{n} kW_k V_{n-k} = \alpha \sum_{k=0}^{n} (n-k)W_k V_{n-k}.$$

This gives us a useful computational rule valid for all $n \geq 1$ and thus a simple online algorithm to successively compute $W_1, W_2, \cdots$:

$$\begin{aligned} W_n &= \sum_{k=1}^{n} ((\frac{\alpha+1}{n})k - 1)V_k W_{n-k} \\ &= \frac{(\alpha+1-n)V_1 W_{n-1} + \cdots + (k\alpha - (n-k))V_k W_{n-k} + \cdots + n\alpha V_n W_0}{n}. \end{aligned}$$

For us, α must be rational. In case α is not an integer, $V(x)^\alpha$ is not unique so $W(x)$ represents just one of the possible values for $V(x)^\alpha$. In most applications, the desired value for $V(x)^\alpha$ will be $W(x)$ or $-W(x)$.

Online algorithms are used in this power operation and the operations discussed below. Here *online* algorithm means that the calculation of the *nth* coefficient is based on the $n-1$ coefficients obtained previously instead of from user input numbers. Online algorithms save space (and hence, to some extent, time) compared to offline algorithms.

2.3.5 Exponential of Generating Functions

The calculation of coefficients of the exponential of a generating function, $W(x) = e^{U(x)}$, where $U_0 = 0$, is similar to the calculation of the powers of generating functions. Differentiation of $W(x)$ gives rise to a useful recurrence, as follows:

$$W(0) = e^{U(0)} = e^0 = 1;$$

$$W'(x) = e^{U(x)}U'(x) = W(x)U'(x).$$

Equating the coefficients of x^{n-1}, we have for all $n \geq 1$

$$nW_n = \sum_{k=1}^{n} W_{n-k}kU_k.$$

This gives us a useful computational rule valid for all $n \geq 1$ and thus a simple online algorithm to successively compute $W_1, W_2, \cdots$:

$$W_n = \frac{1}{n}\sum_{k=1}^{n} W_{n-k}kU_k.$$

2.3.6 Logarithm of Generating Functions

The reversion of the exponential of generating functions is the logarithm of generating functions. That is, if

$$W(x) = e^{U(x)}$$

then

$$U(x) = \ln(W(x)).$$

Here $W_0 = 1$ and $U_0 = 0$. From the exponential relation we have

$$W_n = \frac{1}{n}\sum_{k=1}^{n} W_{n-k}kU_k,$$

hence

$$W_n = \frac{1}{n}\sum_{k=1}^{n-1} W_{n-k}kU_k + \frac{1}{n}nU_n,$$

and so (for $n \geq 1$)

$$U_n = W_n - \frac{1}{n}\sum_{k=1}^{n-1} kU_k W_{n-k}.$$

The initial values for this formula are $W_0 = 1$ and $U_0 = 0$. We can also use this formula to calculate the coefficients of $U(x) = \ln(1 + V(x))$ with $V_0 = 0$ and $U_0 = 0$.

2.3.7 Differentiation and Anti-differentiation of Generating Functions

For a given generating function $U(x)$ where

$$U(x) = U_0 + U_1 x + U_2 x^2 + \cdots,$$

the generating function $U'(x)$, which is the derivative of $U(x)$, is

$$U'(x) = U_1 + 2U_2 x + 3U_3 x^2 + \cdots.$$

If we know the coefficients of $U(x)$, denoted by U_i, the coefficients of the derivative of $U(x)$ can be calculated as

$$DIFF(U_i)_{(i\geq 0)} = (iU_i)_{(i\geq 1)}.$$

If we know the coefficients of the derivative $U'(x)$, denoted by u_i, the coefficients of $U(x)$ can be calculated as

$$ANTIDIFF(u_i) = (0, u_0, \frac{u_1}{2}, \frac{u_2}{3}, \cdots, \frac{u_i}{i+1}).$$

2.3.8 Reversion of Generating Functions

The problem of reversion of series is to solve the equation

$$x = t + V_2 t^2 + V_3 t^3 + V_4 t^4 + \cdots$$

for t, obtaining the coefficients of the power series

$$t = x + W_2 x^2 + W_3 x^3 + W_4 x^4 + \cdots,$$

where $x(t(x)) = x$ and $t(x(t)) = t$.

In 1768, Lagrange published an inversion formula that leads to a classical method to obtain the reversion of power series:

$$W_n = \frac{1}{n}[t^{n-1}](1 + V_2 t + V_3 t^2 + \cdots)^{-n}.$$

Therefore, if we successively compute the negative powers $(1+V_2t+V_3t^2+\cdots)^{-n}$ for $n = 1, 2, 3, \cdots$, an online reversion algorithm can be obtained. The following is the Lagrangian power series reversion algorithm implemented in `AutoGF` (Knuth 1998, Algorithm L, p. 527).

1. [Initialize.] Set $n \leftarrow 1$, $U_0 \leftarrow 1$. (The relation

$$(1 + V_2 t + V_3 t^2 + \cdots)^{-n} = U_0 + U_1 t + \cdots + U_{n-1} t^{n-1} + O(t^n)$$

will be maintained throughout this algorithm.)

2. [Input V_n.] Increase n by 1. If $n > N$, the algorithm terminates; otherwise input the next coefficient, V_n.

3. [Divide.] Set $U_k \leftarrow U_k - U_{k-1}V_2 - \cdots - U_1V_k - U_0V_{k+1}$, for $k = 1, 2, \cdots, n-2$ (in this order); then set

$$U_{n-1} \leftarrow -2U_{n-2}V_2 - 3U_{n-3}V_3 - \cdots - (n-1)U_1V_{n-1} - nU_0V_n.$$

(We have thereby divided $U(x)$ by $V(x)/x$.)

4. [Output W_n.] Output U_{n-1}/n (which is W_n) and return to step 2.

2.3.9 Composition of Generating Functions

The problem of composition of power series is to compute the first N coefficients of $U(V(x))$ for

$$U(x) = U_0 + U_1 x + U_2 x^2 + U_3 x^3 + \cdots$$

and

$$V(x) = V_1 x + V_2 x^2 + V_3 x^3 + \cdots.$$

The following algorithm provides a way to obtain the first N coefficients of composition of generating functions $U(V(x))$ (Knuth 1998, answer to exercise 11, p. 720).

1. [Initialize.] Set $W_0 \leftarrow U_0$. For $1 \leq k \leq N$, set $T_k \leftarrow V_k$ and $W_k \leftarrow 0$.

2. [Iteratively Compose.] For $n = 1, 2, \cdots, N$, do the following: Set $W_j \leftarrow W_j + U_n T_j$ for $n \leq j \leq N$; and then set $T_j \leftarrow T_{j-1}V_1 + \cdots + T_n V_{j-n}$ for $j = N, N-1, \cdots, n+1$.

The sequence $W_0, W_1, \cdots, W_{N-1}$ are the first N coefficients of composition of generating functions $U(V(x))$.

This composition algorithm, as well as the Lagrangian power series reversion discussed in previous section, are classical algorithms and require $O(n^3)$ time complexity. There are other reversion and composition algorithms which are faster asymptotically (Brent and Kung 1978). The Brent and Kung algorithms require $O((nlogn)^{2/3})$ time complexity. Because the Lagrangian reversion algorithm and the composition algorithm described previously are easy to implement, we use these two algorithms in our automated system.

Chapter 3

The Implementation of AutoGF

3.1 Python

In this thesis, AutoGF, an automated system to calculate coefficients of generating functions, is implemented in Python. The version we use is Python 1.5.2. Python is an open source language developed by Guido van Rossum and its first public release was in 1991. The version 1.5.2 was released in April, 1999 and the latest official version at the time of writing is Python 2.1.

3.1.1 What is Python?

Python is an interpreted high-level programming language. It is purely object-oriented, and provides a powerful server-side scripting language for the Web. Like all scripting languages, Python code resembles pseudo-code and does not provide a rich syntax. Thus Python code is readable among multiprogrammer development teams (Lessa 2001).

Python has been in the market for more than 10 years and source code and binaries for its interpreter and all standard libraries are freely available. Consequently, Python is a very stable language. Python provides automatic memory management, which collects unreachable Python objects and frees the user from the responsibility for garbage collection. Python also has exception handling, which helps the user catch errors without having to add a lot of error checking statements to the code. The design of Python ensures that Python programs never crash and always return

a *traceback* message. As an interpreted language, Python is a little bit slower than compiled languages (Lessa 2001).

3.1.2 Why Python?

The coefficients of a generating function can easily be very large integers that exceed the built-in largest integer defined in C++ or Java. In both C++ and Java, at most 64 bits can be used to store an integer. The largest commonly supported integer type in C++ is *unsigned long long*, which can be as large as $2^{64}-1$, and the largest integer type in Java is *long*, which can be as large as $2^{63}-1$. If the coefficients we calculate are greater than the largest number defined, an overflow error will occur. However, it is easy for a generating function to have coefficients greater than the largest integer defined in C++ or Java. For example, The $30th$ coefficient of the generating function

$$f(x) = e^x = \sum_{k\geq 0} \frac{1}{k!} x^k$$

is $\frac{1}{29!}$. The denominator is 29!, which is greater than the largest integer commonly supported in either C++ or Java. Java provides a class called *BigInteger* in package *java.math* to deal with arbitrary precision mathematical calculations that cannot be represented with Java's primitive data types. However, using class objects to represent arbitrary precision numbers can make the program slow.

In Python, there are two types of integers, *i.e.*, IntType and LongType. The range for IntType is -2^{31} to $2^{31}-1$, but the LongType in Python can represent arbitrary precision integers, which are whole numbers of unlimited range (only limited by available memory). Therefore, by using Python, the only thing we need to do is to define an integer as LongType to obtain arbitrary precision.

3.2 The Architecture of AutoGF

AutoGF consists of one class, Ration, and the four modules support.py, basicGF.py, funcOper.py, and AutoGF.py. We will discuss these in the subsections below.

3.2.1 Class Ration

Class Ration, which is defined in file Ration.py, provides the data structure for the rational representation of a number and the arithmetic operations. In generating function applications, it is desirable to compute the rational coefficients exactly instead of with floating point arithmetic. Therefore, rational representation and operations are necessary in this system.

In class Ration, some special methods in Python are implemented so that the Ration objects can work with Python's built-in operators. These special methods include:

initilizing objects ($\_\_init\_\_(self, numerator, denominator)$),

adding objects ($\_\_add\_\_(self, other)$),

subtracting objects ($\_\_sub\_\_(self, other)$),

multiplying objects ($\_\_mul\_\_(self, other)$),

dividing objects ($\_\_div\_\_(self, other)$),

negating objects ($\_\_neg\_\_(self)$), and

creating string representation of an object ($\_\_repr\_\_(self)$).

For example, if we have two ration objects a and b, method $\_\_add\_\_(a, b)$ is called when there is a statement a+b in the program and method $\_\_repr\_\_(a)$ is called when there is a statement print a in the program.

3.2.2 SUPPORT.PY

Module `support.py` provides all the supporting functions used in this system. These supporting functions include the computation of $n!$ ($factorial(n)$), $\binom{a}{b}$ ($choose(a,b)$), a^b ($power(a,b)$), the determination of whether a string is an integer number ($isdata(str)$), and the computation of the greatest common divisor of two integers ($gcd(a,b)$).

3.2.3 BASICGF.PY

Module `basicGF.py` provides functions to obtain the first N coefficients for basic generating functions listed in section 2.2. The result of the first N coefficients is represented as a list of rational numbers.

3.2.4 FUNCOPER.PY

Module `funcOper.py` provides functions to obtain the coefficients of a generating function obtained from operations on one or two generating functions. The operations are discussed in section 2.3. For two generating functions $U(x)$ and $V(x)$, the operations include sum ($U(x)+V(x)$), difference ($U(x)-V(x)$), series product ($U(x)V(x)$), Hadamard product $U(x)@V(x)$, quotient ($U(x)/V(x)$), power ($U(x)^\alpha$), exponential ($e^{U(x)}$), logarithm ($\ln(1+U(x))$), derivative, anti-derivative, reversion, and composition ($U(V(x))$. There are pre-conditions for some of these operations such as that the constant coefficient of $f(x)$ in the exponential operation $e^{f(x)}$ must be zero ($f_0 = 0$). An error message will be printed if a pre-condition of the operation is violated.

3.2.5 AutoGF.py

The user of this automated generating function system should write a text file related to his problem and pass the text file to the automated system. The next section will discuss how to write the text file. Module `AutoGF.py` provides the *main* function, which tokenizes and analyzes the text file provided by the user and prints out the result, which is the first N coefficients of the generating function.

3.3 How to Use The System

The main function is in module `AutoGF.py`. Suppose the text file the user provided is named *test* and the user wants the results to be written in the file named *result*. The command

python AutoGF.py test result

can be used to obtain the coefficients.

3.3.1 How to Compose the Input File

The following is a sample text file that is used to calculate the first 20 coefficients of the generating function F(x) defined by

$$F(x) = e^{A(x)}$$

$$A(x) = U(x) + V(x)$$

$$U(x) = \sum_{n \geq 0} (2n-1)!!x^n$$

$$V(x) = -1$$

```
(1)  20
(2)  U(x) in def V(x) = -1
(3)  def U(x)
(4)  r=1L
```

```
(5)  for i in range(n):
(6)    result.append(Ration(r))
(7)    r=r*(2*i+1)
(8)  endef
(9)  begin
(10) A(x) = U(x) + V(x)
(11) F(x) = e ^ A(x)
(12) F(x)
```

Here, the line numbers are printed for reference purposes and would not be in the actual input file. The first line of the text file is the number of coefficients the user needs. All the basic generating functions should be declared in the second line of the text file. Because the program uses spaces to tokenize, all the spaces in the second line are necessary. There are two basic generating functions in this example, $U(x)$ and $V(x)$. $U(x)$ is a user defined generating function and $V(x)$ is a basic generating function specified in the declaration. Lines (3) through (8) contain the definition provided by the user for $U(x)$. The format for user defined functions is discussed in the next section. If the user wants to define more than one function, the definitions are listed one by one before the *begin* statement. The *begin* statement in line (9) marks the beginning of the generating function operations, which are in lines (10) and (11). The program uses spaces to tokenize, so the spaces in lines (10) and (11) are necessary. The last line of the file notifies the program which function's coefficients should be written to the output file. There is a dictionary in `AutoGF` program, which associates the names of the generating functions (such as $U(x)$, $V(x)$, $A(x)$, and $F(x)$), with the lists of rational numbers that are the first N coefficients of the functions.

3.3.2 How to Define a User Function

The user defined function should begin with *def* followed by the function name and end with *endef*, as in lines (3) and (8) of the example. The statements between

def and *endef* are passed as a string to the *exec* statement (Beazley 2001) in the program. Therefore, Python syntax is used in the function definition, so the user must know a little bit of Python. The number of the coefficients the user needs are represented in variable n, and the coefficient list to be output must be specified by variable *result*, as in lines (5) and (6) of the example.

Chapter 4

Applications of AutoGF to Labeled Graph Enumeration

Exponential generating functions have been widely applied to labeled graph enumeration, that is, to counting labeled graphs and digraphs. A labeled graph or digraph is one with a linear ordering on the vertices. Often this is indicated by naming the vertices $1, 2, \cdots, n$. The exponential generating function for the sequence $(a_0, a_1, a_2, \cdots)$ is

$$g(x) = a_0 + a_1 x + a_2 \frac{x^2}{2!} + a_3 \frac{x^3}{3!} + \cdots + a_r \frac{x^r}{r!} + \cdots$$

Therefore, the Hadamard product $< i! > @g(x)$ can be used to calculate a_i once $g(x)$ is determined. In this chapter, we will discuss the application of `AutoGF` to counting several types of labeled graphs: connected graphs, blocks, connected eulerian graphs, acyclic digraphs, strong digraphs, trees and forests.

4.1 Connected Labeled Graphs

Definition Let G be a graph and let $v_0, v_1, v_2, \cdots, v_n$ be a sequence of vertices of G such that v_i is adjacent to v_{i+1} for $i = 0$ to $n-1$. Such a sequence together with the n edges is called a *walk of length n*. If the edges $\{v_i, v_{i+1}\}$ for $i = 0$ to n are distinct, the walk is called a *trail*. If all the vertices and edges are distinct, it is called a *path of length n*. A *connected graph* is a graph in which any two vertices are joined by a path (Harary and Palmer 1973, p. 6).

The exponential generating function for the number of labeled graphs is:

$$A(x) = \sum_{n \geq 0} \frac{2^{\binom{n}{2}} x^n}{n!}$$

This is because each of the $\binom{n}{2}$ unordered pairs of distinct vertices presents a binary choice, to be included as an edge or not, in building an arbitrary graph on n vertices. The following theorem (Harary and Palmer 1973, p. 8) describes the relationship between labeled graphs and labeled connected graphs.

Theorem 1 *The exponential generating functions $A(x)$ and $C(x)$ for labeled graphs and labeled connected graphs satisfy the following relation:*

$$A(x) = e^{C(x)}.$$

That is,

$$C(x) = \log A(x),$$

from which the text file below was composed. When input to `AutoGF`, the numbers of labeled connected graphs are obtained. The results are shown in Table 4.1 and the time performance is in Table 4.2. In Table 4.2 and time performance tables in following sections, i equals $vertices/10$, which is used to simplify the calculation. From Table 4.2, we can see that if i is the number of vertices then $time/i^4$ seems to approach a constant. So, the time complexity for labeled connected graphs could be $\Theta(n^4)$, or perhaps $\Theta(n^c)$ for some constant c near 4.

```
29
A(x) in def f(x) in def
def A(x)
for i in range(n):
   a=choose(i, 2)
   b=factorial(i)
   c=power(2, a)
   d=Ration(c)/Ration(b)
   result.append(d)
endef
def f(x)
for i in range(n):
   a=factorial(i)
   result.append(Ration(a))
```

```
endef
begin
C(x) = ln A(x)
c(x) = C(x) @ f(x)
c(x)
```

4.2 Labeled Blocks

Definition The removal of a vertex v from a graph G results in the subgraph $G - v$ of G consisting all vertices of G except v and all edges not incident with v. A *cutpoint* of a graph is one whose removal increases the number of components. A *block* or *nonseparable graph* is connected, nontrivial, and has no cutpoints (Harary and Palmer 1973, p. 9).

Let $B(x)$ denote the exponential generating function for labeled blocks and $C(x)$ denote the exponential generating function for connected labeled graphs. The following theorem (Harary and Palmer 1973, p. 10) describes the relationship between $B(x)$ and $C(x)$.

Theorem 2 *The two exponential generating functions $B(x)$ and $C(x)$ for labeled blocks and connected graphs are related by:*

$$logC'(x) = B'(xC'(x)).$$

So, we have

$$xC'(x) = xe^{B'(xC'(x))}.$$

Let $\varphi(x) =$ reversion of $xC'(x)$, then

$$x = \varphi(x)e^{B'(x)}$$

$$log\frac{x}{\varphi(x)} = B'(x)$$

Table 4.1: Number of Connected Labeled Graphs

Vertices	**Connected Labeled Graphs**
0	0
1	1
2	1
3	4
4	38
5	728
6	26704
7	1866256
8	251548592
9	66296291072
10	34496488594816
11	35641657548953344
12	73354596206766622208
13	301272202649664088951808
14	247164881103044373529089126 4
15	405276809377304802346097553448 96
16	132857895833578320100833898684542771 2
17	870896890524471828417913889890514009784 32
18	114164135204345223087886742857132479192446402 56
19	29929384116018180373700342801528939354584661726986 24
20	15692155707394063462565472103777685757658849832648044052 48
21	16454716025370648777224855178001761643740015163273062875613 10208
22	34508369722950116062601714914260936851437546115328069963470 23345844224
23	14473931784581530777452916362195345689326195578125463551466 449404195748970496
24	12141645838784034832247737828641414668703840762841807733278 35292186712271438605189 12
25	20370329409143419676922561585800800631483979568699568444273 55893688994716051486372603625472
26	68351532186533737864736355381396298734910952426503780423683 99073031877791537875686137879298939289 6
27	45869953864873439868450361909803259294922972126320661426113 60844233962960637520118252235915249481987129344
28	61565621838274124223450863197683805128241193119763036274703 37241742223953435431098610286958165669508558908114862 08

Table 4.2: Time Performance of Connected Labeled Graphs

i	Vertices	time (s)	$time/i^{3.5}$	$time/i^{4}$	$time/i^{4.5}$
1	10	0.17	0.1700	0.1700	0.1700
2	20	0.48	0.0424	0.0300	0.0212
3	30	1.53	0.0327	0.0189	0.0109
4	40	4.07	0.0318	0.0159	0.0079
5	50	9.24	0.0331	0.0148	0.0066
6	60	18.35	0.0347	0.0142	0.0058
7	70	33.64	0.0371	0.0140	0.0053
8	80	56.55	0.0391	0.0138	0.0049
9	90	90.32	0.0413	0.0138	0.0046

Applying `AutoGF` on the following text file can obtain the number of labeled blocks. The results are shown in Table 4.3 and the time performance is in Table 4.4. From Table 4.4, we see that $time/i^5$ seems to approach a constant. So, the time complexity for labeled blocks could be $\Theta(n^5)$, or perhaps $\Theta(n^c)$ for some constant c near 5.

```
29
A(x) in def f(x) in def a(x) = 1x^1
def A(x)
for i in range(n):
   a=choose(i, 2)
   b=factorial(i)
   c=power(2, a)
   d=Ration(c)/Ration(b)
   result.append(d)
endef
def f(x)
for i in range(n):
   a=factorial(i)
   result.append(Ration(a))
endef
begin
C(x) = ln A(x)
c(x) = C(x) @ f(x)
```

```
diffC(x) = diff C(x)
b(x) = a(x) * diffC(x)
fi(x) = reverse b(x)
fi1(x) = a(x) / fi(x)
diffB(x) = ln fi1(x)
B(x) = antidiff diffB(x)
b(x) = B(x) @ f(x)
b(x)
```

4.3 Connected Labeled Eulerian Graphs

Definition The *degree* of a vertex v in a graph G, denoted *deg* v, is the number of edges of G that are incident with v. If every vertex of G has even degree, G is called *even*. An eulerian graph is a connected, even graph (Harary and Palmer 1973, p. 11).

Let W_p denote the number of labeled, even graphs of order p. A rather surprising result occurs (Harary and Palmer 1973, p. 11).

Theorem 3 *The number of labeled, even graphs of order p equals the number of graphs of order $p-1$:*

$$W_p = 2^{\binom{p-1}{2}}.$$

Let $W(x)$ be the exponential generating function for labeled even graphs and $U(x)$ be the exponential generating function for labeled eulerian graph. That is,

$$W(x) = \sum_{p=1}^{\infty} 2^{\binom{p-1}{2}} x^p/p!$$

and

$$U(x) = \sum_{p=1}^{\infty} U_p x^p/p!.$$

Theorem 4 *The exponential generating function $U(x)$ for labeled eulerian graphs satisfies*

$$U(x) = log(W(x)+1).$$

Table 4.3: Number of Labeled Blocks

Vertices	**Labeled Blocks**
0	0
1	0
2	1
3	1
4	10
5	238
6	11368
7	1014888
8	166537616
9	50680432112
10	29107809374336
11	32093527159296128
12	68846607723033232640
13	290126947098532533378816
14	241768461252342560072113254 4
15	400135227025387809008038938818 56
16	131890599047043284183515841468098 3552
17	867294692013411766731398076352353096 47872
18	113898127669565192464993027060072099761493 44256
19	298903804440873945806253789778187016308202160 4720640
20	156807891747501546261703952529349290193445253718290 2878208
21	164481274687369706320630709943032493689609949596034673291 13 29280
22	345007690213289088905230743901921202654368150973462713779 78 32562704384
23	144721858359019110759393392724109391465266278188152050886 30 763041366328213504
24	121408469217491249023027086501246145917790377352817522231 40 413578895590527516016640 0
25	203696009323637737747864160956392662601021456622280300380 60 461804984013450297258011100446 72
26	683502081369038493825017186997932358355578188534419496302 40 001716817665982399618700131891184402432
27	458694740414074379652875819830752029537495656494318383970 55 009655279513554388385709803110777957962712350 72
28	615652750631022935834873077960908990801917446257759574422 48 647064205210942337728829962193306014297555041125820 0064

Table 4.4: Time Performance of Labeled Blocks

i	Vertices	time (s)	$time/i^4$	$time/i^5$	$time/i^6$
1	10	0.35	0.350	0.350	0.350
2	20	2.35	0.147	0.073	0.037
3	30	11.62	0.143	0.048	0.016
4	40	38.19	0.149	0.037	0.009
5	50	100.96	0.162	0.032	0.006
6	60	234.85	0.181	0.030	0.005
7	70	480.81	0.200	0.029	0.004

The number of connected labeled eulerian graphs can be obtained by applying `AutoGF` on the following text file. The results are shown in Table 4.5 and the time performance is in Table 4.6. From Table 4.6, we can see that $time/i^4$ seems to approach a constant. So, the time complexity for connected labeled eulerian graphs could be $\Theta(n^4)$, or perhaps $\Theta(n^c)$ for some constant c near 4.

```
30
W(x) in def f(x) in def
def W(x)
result.append(Ration(1))
for i in range(1, n):
   a=choose(i-1, 2)
   b=factorial(i)
   c=power(2, a)
   d=Ration(c)/Ration(b)
   result.append(d)
endef
def f(x)
for i in range(n):
   a=factorial(i)
   result.append(Ration(a))
endef
begin
U(x) = ln W(x)
```

```
u(x) = U(x) @ f(x)
u(x)
```

4.4 Labeled Acyclic Digraphs

Definition A *walk* of length n in a digraph D is determined by its sequence of vertices $v_0, v_1, v_2, \cdots, v_n$ in which v_i is adjacent to v_{i+1} for $i < n$. A *closed walk* has the same first and last vertices. A *cycle* is a nontrivial closed walk with all vertices distinct except the first and last. An *acyclic* digraph has no cycles (Harary and Palmer 1973, p. 18).

Let $g(x)$ be the special exponential generating function (with weight $1/(n!2^{\binom{n}{2}})$ instead of $1/n!$) for the set of labeled acyclic digraphs. We have (Robinson 1973)

$$g(x) = \frac{1}{f(x)}$$

and

$$f(x) = \sum_{i=0}^{\infty} \frac{(-1)^i x^i}{i!2^{\binom{i}{2}}}$$

The number of labeled acyclic digraphs can be obtained by $g(x)@(2^{\binom{i}{2}}i!)$. Applying the following text file to `AutoGF` can obtain the number of labeled acyclic digraph. The results are shown in Table 4.7 and the time performance is in Table 4.8. From Table 4.8, we see that $time/i^5$ seems to approach a constant. So, the time complexity for labeled acyclic digraphs could be $\Theta(n^5)$, or perhaps $\Theta(n^c)$ for some constant c near 5.

```
26
f(x) in def fact(x) in def c(x) = 1
def f(x)
for i in range(n):
   a=choose(i, 2)
   b=factorial(i)
   c=power(2, a)
```

Table 4.5: Number of Connected Labeled Eulerian Graphs

Vertices	**Connected Labeled Eulerian Graphs**
0	0
1	1
2	0
3	1
4	3
5	38
6	720
7	26614
8	1858122
9	250586792
10	66121926720
11	34442540326456
12	35611003057733928
13	73321307277341501168
14	30120169035718709752896O
15	24713543216816059831023708864
16	40525241311304939167532163726672
17	13285387301673910087311002605214808832
18	87088366030214648381879697292246665507840
19	11416326635434367335852139273129517546817518976
20	29929270101711603018661428577262009759583612710657728
21	15692125799817962844344002675196573604155873125081512174O8
22	16454700339530451359024830804863915826828211432301128864387 78880
23	34508353271874795891874674233715177108905115177521001847552 72223751424
24	14473928334162515846638214596417526677787291548160298089549 521495020615267072
25	12141644391486445470209298866826569662207491374208788584136 18223590887633795308677l2
26	20370328195022402551394801101435954482828794989816814465049 69241249014873033618827837153280
27	68351530149540378819666723147229445654570198213092656662768 22525153317113101600234223134092097433O
28	45869953181365288368897400817381238796643968164071866862947 47872739237691976733834790416820083081968944128
29	61565621379577176099639635945335275528341816323598265597874 46294119459572909288745509008905709538154908655453614O8

Table 4.6: Time Performance of Connected Labeled Eulerian Graphs

i	Vertices	time (s)	$time/i^{3.5}$	$time/i^{4}$	$time/i^{4.5}$
1	10	0.17	0.1700	0.1700	0.1700
2	20	0.49	0.0433	0.0306	0.0217
3	30	1.58	0.0338	0.0195	0.0113
4	40	4.16	0.0325	0.0163	0.0081
5	50	10.07	0.0360	0.0161	0.0072
6	60	19.29	0.0365	0.0149	0.0061
7	70	34.14	0.0376	0.0142	0.0054
8	80	56.98	0.0393	0.0139	0.0049
9	90	95.68	0.0437	0.0146	0.0049
10	100	148.76	0.0470	0.0149	0.0047
11	110	219.13	0.0496	0.0150	0.0045

```
   d=b*c
   e=(-1) ** i
   f=Ration(e)/Ration(d)
   result.append(f)
endef
def fact(x)
for i in range(n):
   a=factorial(i)
   b=choose(i, 2)
   c=power(2, b)
   d=c*a
   result.append(Ration(d))
endef
begin
g(x) = c(x) / f(x)
r(x) = g(x) @ fact(x)
r(x)
```

Table 4.7: Number of Labeled Acyclic Digraphs

Vertices	**Labeled Acyclic Digraphs**
0	1
1	1
2	3
3	25
4	543
5	29281
6	3781503
7	1138779265
8	783702329343
9	1213442454842881
10	4175098976430598143
11	31603459396418917607425
12	521939651343829405020504063
13	186766007444320351866648169267 21
14	14394281410443983349417907198395351 03
15	23772526555341035499218021828637671925350 5
16	83756670773733320287699303047996412235223138303
17	62707921196923889899446452602494921906963551482675201
18	99421195322159515895228914592354524516555026878588305014783
19	33277190122710759173617757331126112588358307625842190258354 6773505
20	23448804510510889881525598552290991888990811922342912987958 03236068491263
21	34698768283588750028759328430181088222313944540438601719027 559113446586077675521
22	10758229217257614936529561793276243265737276628091852181040 900005005595275116934951075 83
23	69743329837281492647141549700245804876504274990515985894109 106401549811985510951501377122074625
24	94357834486618508938161152848393652139840190649830971425387 76232515861210619999018077882548839455391743
25	26595736077837381797589415611381864213634144630228868053872 37791664160209221179044776702988190938181603615047681

Table 4.8: Time Performance of Labeled Acyclic Digraphs

i	Vertices	time (s)	$time/i^{4.5}$	$time/i^{5}$	$time/i^{5.5}$
1	10	0.17	0.1700	0.1700	0.17
2	20	1.06	0.0468	0.0331	0.0234
3	30	5.35	0.0381	0.0220	0.0127
4	40	19.65	0.0384	0.0192	0.0096
5	50	57.94	0.0415	0.0185	0.0083
6	60	146.23	0.0461	0.0188	0.0077
7	70	329.00	0.0518	0.0196	0.0074

4.5 Labeled Strong Digraphs

Definition A digraph G is *strongly connected* or *strong* if for every ordered pair $u, v \in V(G)$ there is a u, v-path in G (West 1996, p. 88).

Let $D(x)$ be the special exponential generating function for the set of all digraphs and $S(x)$ be the special exponential generating function for the set of all strong digraphs. The special exponential generating function is weighted by $(1/(i!2^{\binom{i}{2}}))_{i\geq 0}$ instead of $(1/i!)_{i\geq 0}$. Thus

$$D(x) = \sum_{i\geq 0} \frac{2^{\binom{i}{2}} x^i}{i!},$$

since there are $4^{\binom{i}{2}}$ labeled unrestricted digraphs with i vertices. The following formula describes the relationship between $D(x)$ and $S(x)$ (Robinson, 1973):

$$D(x) = \frac{1}{e^{-S(x)}}.$$

So, the following two formulas describe how to obtain $S(x)$:

$$E(x) = (\frac{1}{D(x)})@(2^{\binom{i}{2}})_{i\geq 0},$$

$$S(x) = -logE(x).$$

The following text file can be used to obtain the numbers of labeled strong digraphs. The result is shown in Table 4.9 and the time performance is in Table 4.10. From Table 4.10, we can see that $time/i^4$ is seems to approach a constant. So, the time complexity for labeled strong digraphs could be $\Theta(n^4)$, or perhaps $\Theta(n^c)$ for some constant c near 4.

```
24
D(x) in def fact(x) in def c(x) = 1 a(x) in def f(x) in def
def D(x)
for i in range(n):
   a=choose(i, 2)
   b=factorial(i)
   c=power(2, a)
   f=Ration(c)/Ration(b)
   result.append(f)
endef
def fact(x)
for i in range(n):
   a=choose(i, 2)
   c=power(2, a)
   result.append(Ration(c))
endef
def a(x)
for i in range(n):
   result.append(Ration(-1))
endef
def f(x)
for i in range(n):
   a=factorial(i)
   result.append(Ration(a))
endef
begin
d(x) = c(x) / D(x)
E(x) = d(x) @ fact(x)
e(x) = ln E(x)
S(x) = e(x) @ a(x)
r(x) = S(x) @ f(x)
r(x)
```

Table 4.9: Number of Labeled Strong Digraphs

Vertices	**Labeled Strong Digraphs**
0	0
1	1
2	1
3	18
4	1606
5	565080
6	734774776
7	3523091615568
8	63519209389664176
9	4400410978376102609280
10	1190433705317814685295399296
11	1270463864957828799318424676767488
12	538106796682625513245961168151135932953 6
13	907657888394030904572441289513074133718834944 00
14	610906446282154570404642603246573776322476063573288857 6
15	164249420920095915258592567599391151659433404720112110263 2675328
16	176512244340798844580503355255335526607690756298732729736 0836809051559936
17	75846144198058811756804332211790789408476795856235771995725 3581786882082364309504 0
18	130334499991586909527420498909537734590625269179659951103626888752043097479468849813827977216
19	895768038022758319622887534696406840807475012662988597483163983687225234069527076226938380360833322188824624375108553869772331593489405576878844787393033664278117500624391522547930813616531548620861855114570205522165765006243915225479308136165315486208618551145702055221657620707576795790195142721569210634436344018502593560160171191588440833903486828501437858808457787221705290533079569526240346112000
22	1190827581060915090283626443408959287332763159167976860776946422376490007400140895593655763479186824298483861059776375654138985221547334041623
23	2094947013107452045047680816196488106043224498351106478076378607150625190473998586219691425517859805606183649452249422561158013258438444361512647713095686

Table 4.10: Time Performance of Labeled Strong Digraphs

i	Vertices	time (s)	$time/i^{3.5}$	$time/i^{4}$	$time/i^{4.5}$
1	10	0.22	0.2200	0.2200	0.2200
2	20	0.90	0.0796	0.0562	0.0398
3	30	2.86	0.0612	0.0353	0.0204
4	40	7.91	0.0618	0.0309	0.0154
5	50	18.17	0.0650	0.0291	0.0130
6	60	37.17	0.0703	0.0287	0.0117
7	70	67.73	0.0746	0.0283	0.0107
8	80	112.72	0.0778	0.0275	0.0097
9	90	185.61	0.0849	0.0283	0.0094
10	100	292.88	0.0926	0.0293	0.0093

4.6 Labeled Trees and Forests

Definition A *tree* is a connected acyclic graph. A *forest* is an acyclic graph (West 1996, p. 51).

The exponential generating function for labeled trees is:

$$t(x) = \sum_{k \geq 1} \frac{k^{k-2} x^k}{k!}.$$

The relationship between the exponential generating function for labeled trees $t(x)$ and the exponential generating function for labeled forests $f(x)$ is:

$$f(x) = e^{t(x)}.$$

Applying the Hadamard product of $t(x)$ and $f(x)$ with $i!$ for $i \geq 0$ will produce the numbers of labeled trees and labeled forests. The following text file can be used to obtain the number of labeled forests and the number of labeled trees can be obtained by substituting $f(x)$ with $t(x)$ in the second to last line and removing the third to last line which calculates $f(x)$.

The number of labeled trees are shown in Table 4.11 and time used to obtain these numbers are shown in Table 4.12. From Table 4.12, we can see that $time/i^3$ seems to approach a constant. So, the time complexity for labeled trees could be $\Theta(n^3)$, or perhaps $\Theta(n^c)$ for some constant c near 3.

The number of labeled forests are shown in Table 4.13 and time performance of labeled forests is shown in Table 4.14. From Table 4.14, we can see that $time/i^4$ seems to approach a constant. So, the time complexity for labeled trees could be $\Theta(n^4)$, or perhaps $\Theta(n^c)$ for some constant c near 4.

```
30
t(x) in def fact(x) in def
def t(x)
result.append(Ration())
result.append(Ration(1))
b=0L
for i in range(2, n):
   a=i-2
   b=power(i, a)
   c=factorial(i)
   d=Ration(b)/Ration(c)
   result.append(d)
endef
def fact(x)
for i in range(n):
   a=factorial(i)
   result.append(Ration(a))
endef
begin
f(x) = e ^ t(x)
r(x) = f(x) @ fact(x)
r(x)
```

Table 4.11: Number of Labeled Trees

Vertices	**Labeled Trees**
0	0
1	1
2	1
3	3
4	16
5	125
6	1296
7	16807
8	262144
9	4782969
10	100000000
11	2357947691
12	61917364224
13	1792160394037
14	56693912375296
15	1946195068359375
16	72057594037927936
17	2862423051509815793
18	121439531096594251776
19	5480386857784802185939
20	262144000000000000000000
21	13248496640331026125580781
22	705429498686404044207947776
23	39471584120695485887249589623
24	2315513501476187716057433112576
25	142108547152020037174224853515625
26	9106685769537214956799814036094976
27	608266787713357709119683992618861307
28	42277452950578284263485622772148731904
29	3053134545970524535745336759489912159909

Table 4.12: Time Performance of Labeled Trees

i	Vertices	time (s)	$time/i^{2.5}$	$time/i^{3}$	$time/i^{3.5}$
5	50	0.35	0.00626	0.00280	0.00125
10	100	1.17	0.00370	0.00117	0.00037
15	150	3.31	0.00380	0.00098	0.00025
20	200	7.15	0.00400	0.00089	0.00020
25	250	14.01	0.00448	0.00090	0.00018
30	300	22.67	0.00460	0.00084	0.00015
35	350	39.21	0.00541	0.00091	0.00015
40	400	53.44	0.00528	0.00084	0.00013
45	450	81.86	0.00603	0.00090	0.00013
50	500	109.69	0.00621	0.00088	0.00012
55	550	147.76	0.00659	0.00089	0.00012
60	600	192.66	0.00691	0.00089	0.00011
65	650	248.87	0.00731	0.00091	0.00011

Table 4.13: Number of Labeled Forests

Vertices	**Labeled Forests**
0	1
1	1
2	2
3	7
4	38
5	291
6	2932
7	36961
8	561948
9	10026505
10	205608536
11	4767440679
12	123373203208
13	3525630110107
14	110284283006640
15	3748357699560961
16	137557910094840848
17	5421179050350334929
18	228359487335194570528
19	10239206473040881277575
20	4869097448625766542836 16
21	2447669761084907491190037 1
22	1296922170326967017021456192
23	722423439462504747653752160 97
24	42204086040527950506306939376 00
25	25802582394869095934016499242 3001
26	16476325133131206856388531345000832
27	10968815430248987996907754154748767 11
28	76004217718178366542848556101866327168
29	54730089071627094555282589309724028768 75

Table 4.14: Time Performance of Labeled Forests

i	Vertices	time (s)	$time/i^{3.5}$	$time/i^{4}$	$time/i^{4.5}$
1	10	0.20	0.2000	0.2000	0.2000
2	20	0.55	0.0486	0.0344	0.0243
3	30	1.89	0.0404	0.0233	0.0135
4	40	4.78	0.0373	0.0187	0.0093
5	50	10.51	0.0376	0.0168	0.0075
6	60	20.29	0.0383	0.0157	0.0064
7	70	37.07	0.0391	0.0148	0.0056
8	80	59.37	0.0406	0.0143	0.0051
9	90	91.68	0.0419	0.0140	0.0047
10	100	137.21	0.0434	0.0137	0.0043
11	110	207.04	0.0469	0.0141	0.0043
12	120	297.39	0.0497	0.0143	0.0041

Chapter 5

Applications of AutoGF to Counting Integer Partitions

Definition A *partition* of a positive integer n is a representation of n as a sum of positive integers

$$n = r_1 + r_2 + \cdots + r_k,$$

where $r_1 \geq r_2 \geq \cdots \geq r_k \geq 1$. The numbers $r_1, r_2, \cdots, r_k$ are the *parts* of the partition and thus the above is a partition of n into k parts (Wilf 1990, p. 91).

There are 7 partitions of 5, namely $5 = 5, = 4+1, = 3+2, = 3+1+1, = 2+2+2, = 2+1+1+1, = 1+1+1+1+1$. The number of partitions of n is denoted by $p(n)$. The following theorem describes the generating function of $p(n)$ (Hall 1986, p. 34).

Theorem 5 *The ordinary generating function of* $p(n)$*,*

$$P(x) = 1 + p(1)x + p(2)x^2 + \cdots + p(n)x^n + \cdots$$

is given by

$$P(x) = \prod_{i=1}^{\infty} \frac{1}{1 - x^i}.$$

The text file below is used to calculate $p(n)$ based on the theorem. The results are shown in Table 5.1 and the time performance is in Table 5.2. From Table 5.2, we can see that $time/i^3$ seems roughly to approach a constant for unrestricted integer partitions. So, the time complexity for unrestricted integer partitions could be $\Theta(n^3)$, or perhaps $\Theta(n^c)$ for some constant c near 3.

```
31
A(x) in def B(x) = 1
def A(x)
coeff1=[]
coeff2=[]
for i in range(n):
   coeff1.append(Ration())
   coeff2.append(Ration())
   result.append(Ration())
result[0]=Ration(1)
result[1]=Ration(-1)
i=2
while i<=30:
   for l in range(n):
      coeff1[l]=Ration()
      coeff2[l]=Ration()
   coeff2[0]=Ration(1)
   coeff2[i]=Ration(-1)
   for j in range(n):
      coeff1[j]=result[j]
   for a in range(n):
       b=a+1
       s=Ration()
       for c in range(b):
          s=s+coeff1[c]*coeff2[a-c]
       result[a]=s
   i=i+1
endef
begin
C(x) = B(x) / A(x)
C(x)
```

A partition is said to be into *distinct parts* if the parts are all different. There are 3 partitions into distinct parts of 5, namely $5 = 5, = 4 + 1, = 3 + 2$. The generating function for partitions into distinct parts is (Hall 1986, Chapter 4, Problem 3, p. 46)

$$D(x) = \prod_{i=1}^{\infty}(1 + x^i).$$

The following text file is used to calculate the numbers of partitions into distinct parts. The results are shown in Table 5.3 and the time performance is in Table

Table 5.1: Number of Unrestricted Partitions

n	**Unrestricted Partitions** $p(n)$
0	1
1	1
2	2
3	3
4	5
5	7
6	11
7	15
8	22
9	30
10	42
11	56
12	77
13	101
14	135
15	176
16	231
17	297
18	385
19	490
20	627
21	792
22	1002
23	1255
24	1575
25	1958
26	2436
27	3010
28	3718
29	4565
30	5604

Table 5.2: Time Performance of Unrestricted Partitions

i	n	time (s)	$time/i^{2.5}$	$time/i^{3}$	$time/i^{3.5}$
1	10	0.31	0.310	0.310	0.3100
2	20	1.38	0.244	0.173	0.1220
3	30	4.63	0.297	0.171	0.0990
4	40	10.08	0.315	0.158	0.0788
5	50	18.64	0.333	0.149	0.0667
6	60	32.46	0.368	0.150	0.0614
7	70	54.80	0.423	0.160	0.0604
8	80	73.82	0.408	0.144	0.0510
9	90	109.68	0.451	0.150	0.0502
10	100	147.06	0.465	0.147	0.0465
11	110	193.54	0.482	0.145	0.0438

5.4. From Table 5.4, we can see that $time/i^3$ roughly approaches a constant for the distinct integer partitions. So, the time complexity for distinct integer partitions could be $\Theta(n^3)$ or $\Theta(n^c)$ for some constant c near 3.

```
31
A(x) in def
def A(x)
coeff1=[]
coeff2=[]
for i in range(n):
   coeff1.append(Ration())
   coeff2.append(Ration())
   result.append(Ration())
result[0]=Ration(1)
result[1]=Ration(1)
i=2
while i <=30:
   for l in range(n):
      coeff1[l]=Ration()
      coeff2[l]=Ration()
   coeff2[0]=Ration(1)
   coeff2[i]=Ration(1)
```

```
   for j in range(n):
      coeff1[j]=result[j]
   for a in range(n):
       b=a+1
       s=Ration()
       for c in range(b):
          s=s+coeff1[c]*coeff2[a-c]
       result[a]=s
   i=i+1
endef
begin
A(x)
```

Table 5.3: Number of Partitions with Distinct Parts

n	**Partitions with Distinct Parts** $D(n)$
0	1
1	1
2	1
3	2
4	2
5	3
6	4
7	5
8	6
9	8
10	10
11	12
12	15
13	18
14	22
15	27
16	32
17	38
18	46
19	54
20	64
21	76
22	89
23	104
24	122
25	142
26	165
27	192
28	222
29	256
30	296

Table 5.4: Time Performance of Partitions into Distinct Parts

i	n	time (s)	$time/i^{2.5}$	$time/i^{3}$	$time/i^{3.5}$
1	10	0.36	0.36	0.36	0.36
2	20	1.35	0.239	0.169	0.119
3	30	4.57	0.293	0.169	0.0977
4	40	9.87	0.308	0.154	0.0771
5	50	19.14	0.342	0.153	0.0684
6	60	32.24	0.366	0.149	0.0609
7	70	51.20	0.395	0.149	0.0564
8	80	77.13	0.426	0.151	0.0533
9	90	110.87	0.456	0.152	0.0507
10	100	152.57	0.482	0.153	0.0482
11	110	196.32	0.489	0.147	0.0445

Chapter 6

Conclusion

We have developed an automated system, `AutoGF`, to calculate coefficients of generating functions. The user of `AutoGF` must analyze the generating function problem and express it as a series of operations on basic generating functions which are defined in the system or supplied by the user. The user then creates a text file listing these operations and passes the text file to `AutoGF`. In turn, `AutoGF` tokenizes and analyzes the text file, computes the requested coefficients and writes them to an output file.

Exponential generating functions have been widely applied to count classes of labeled graphs and digraphs. The numbers of graphs or digraphs with different numbers of vertices can be found by applying the Hadamard product of the factorial number series to the coefficients of exponential generating functions. In this thesis, `AutoGF` is used to count the numbers of connected labeled graphs, labeled blocks, connected labeled eulerian graphs, labeled acyclic digraphs, labeled strong digraphs, and labeled forests. The time performances of these graph enumerations are measured and their time complexities are estimated. The time complexities are summarized in Table 6.1. The time used to count the number of graphs of a given type is determined by the number of operations and time used for each operation. The latter depends on the implementation of arithmetic operations and the size and growth rate of the numbers. From Table 6.1, we can see that the time complexities for these graph enumerations appear to lie in the range $\Theta(n^4)$ to $\Theta(n^5)$ except for labeled trees. The latter seems to be in the class $\Theta(n^3)$, presumably because it only requires simple manipulations of numbers of size $\Theta(nlogn)$ instead of $\Theta(n^2)$.

Table 6.1: Time Complexity of Graph Enumerations

Graphs	Time Complexity
Connected Labeled Graph	$\Theta(n^4)$
Labeled Blocks	$\Theta(n^5)$
Connected Labeled Eulerian Graph	$\Theta(n^4)$
Labeled Acyclic Digraphs	$\Theta(n^5)$
Labeled Strong Digraphs	$\Theta(n^4)$
Labeled Trees	$\Theta(n^3)$
Labeled Forests	$\Theta(n^4)$

Table 6.2: Time Complexity of Integer Partition Enumerations

Partition Type	Time Complexity
General Partition	$\Theta(n^3)$
Distinct Partition	$\Theta(n^3)$

Generating functions can also be used to count partitions of integers. In this thesis, `AutoGF` is used to count general partitions and partitions into distinct parts. Time complexities are estimated and summarized in Table 6.2. Both types of partitions can apparently be counted in time complexity $\Theta(n^3)$.

There are two obvious directions for future improvements to `AutoGF`. One would be to allow for arbitrary polynomials with rational coefficients to be used as coefficients of generating functions. At present, only generating functions with rational coefficients can be calculated and manipulated by `AutoGF`. Arbitrary rational polynomial coefficients would allow for applications in which there is more than one parameter to keep track of, such as graphs or digraphs by numbers of edges or numbers of components as well as by numbers of vertices.

The second direction would be to allow for problems defined implicitly by equations, which would then be automatically expressed in terms of fundamental operations on basic series.

Bibliography

D. M. Beazley, *Python Essential Reference,* 2nd ed., New Riders, Indianapolis, IN (2001).

R. P. Brent and H. T. Kung, Fast Algorithms for Manipulating Formal Power Series, *J. Assoc. Comput. Mach.* **25** (1978) 581-595.

R. L. Finney, M. D. Weir, and F. R. Giordano, *Thomas's Calculus, Part I,* 10th ed., Addison Wesley, Reading, MA (2000).

M. Hall, Jr., *Combinatorial Theory,* 2nd ed., John Wiley & Sons, New York (1986).

F. Harary and E. M. Palmer, *Graphical Enumeration,* Academic Press, New York (1973).

D. E. Knuth, *The Art of Computer Programming, Volume 2, Seminumerical Algorithms,* 3rd ed., Addison Wesley, Reading, MA (1998).

A. Lessa, *Python Developer's Handbook,* Sams Publishing,Indianapolis, IN (2001).

R. W. Robinson, Counting Labeled Acyclic Digraphs, *New Directions in the Theory of Graphs* (F. Harary, ed.) Academic Press, New York (1973) 239-273.

A. Tucker, *Applied Combinatorics,* 3rd ed., John Wiley & Sons, New York (1995).

D. B. West, *Introduction to Graph Theory,* Prentice Hall, Upper Saddle River, NJ (1996).

H. S. Wilf, *Generatingfunctionology,* Academic Press, San Diego, CA (1990).