COUNTING FUNCTIONS OF MAGIC LABELLINGS

A thesis presented to the faculty of
San Francisco State University
In partial fulfilment of
The Requirements for
The degree

Master of Science
In
Computer Science

by

Alex Plotitsa

San Francisco, California

May 2010

CERTIFICATION OF APPROVAL

I certify that I have read *COUNTING FUNCTIONS OF MAGIC LABELLINGS* by Alex Plotitsa and that in my opinion this work meets the criteria for approving a thesis submitted in partial fulfillment of the requirements for the degree: Master of Science in Computer Science at San Francisco State University.

Matthias Beck
Assistant Professor of Mathematics

Dragutin Petkovic
Professor of Computer Science

Rahul Singh
Associate Professor of Computer Science

COUNTING FUNCTIONS OF MAGIC LABELLINGS

Alex Plotitsa
San Francisco State University
2010

A magic labelling of a set system is a labelling of its points by distinct positive integers so that every set of the system has the same sum, the magic sum. The most famous class of examples are magic squares (the sets are the rows, columns, and diagonals of a matrix). It follows from a recent paper by Matthias Beck and Thomas Zaslavky that the number of n by n magic labellings is a quasipolynomial function of the magic sum, and also of an upper bound on the entries in the square. The contribution of this thesis is to develop software that allows computation of a large class of examples of generating functions for such counting functions.

The software will utilize previously developed programs THAC (developed at SFSU) and Latte (developed at UC Davis) to compute intermediate results required by the overall computation. The symbolic algebra program Maple (developed at the University of Waterloo, Canada) will be used for final algebraic manipulation to achieve the final result which is the generating function of a counting function.

While there are other methods to compute these types of counting functions, it is believed that the approach used in this thesis is novel and no such software exists.

ACKNOWLEDGMENTS

I dedicate this thesis to my father. I would like to acknowledge tremendous help from Dr. Matthias Beck. Without his perseverance and attention to detail, this work would not be possible. I would also like to thank my thesis committee members, Dr. Dragutin Petkovic and Dr. Rahul Singh, for their guidance. Dr. Barry Levine kept his watchful eye on my progress and I am thankful to him for that. Special thanks to my wife, Susanna and her mom Lusik for their dedication and support for this effort. And of course to my daughters, Michelle and Eileen, who are the source of my inspiration.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

A magic labelling of a matrix is a labelling of its entries by distinct non-negative integers so that the sum of entries in each column, row, and diagonal is the same and is called the magic sum. There are several examples, including magic squares. Semimagic squares are like the magic squares, but excluding the diagonal constraints. Another labelling is magilatin which is similar to a magic labelling, but values need to be distinct only within each column, row and diagonal . In the latin case, the only restriction is that each entry in a column or a row has to be distinct [5]. A counting function computes the number of magic labellings as a function of the magic sum (the affine case) or size of an individual entry in a labelling (the cubical case). A generating function $f(x)$ is a power series

$$f(x) = \sum_{n=0}^{\infty} a_n x^n$$

whose coefficients give the sequence $\{a_0, a_1, \ldots, a_n\}$.

The goal of this work is to develop an integrated software solution to illustrate computational feasibility of a novel theoretical approach developed by Matthias Beck and Thomas Zaslavsky in [5] to derive generating functions for magic counting functions. The theoretical approach is based on theory of hyperplane arrangements, Inside-out polytopes, and counting integer points inside polytopes. Our computational approach integrates previously developed programs (THAC [15] and Latte [9]) to compute intermediate building blocks used in the overall computation. The symbolic algebra program Maple [11] is used for algebraic manipulation to obtain the final result. The computational intent of this thesis is to write a standalone integrated software program (called CGF for Counting Function Generator) which uses and integrates THAC, Latte and Maple. Inputs to this program are the dimension of the magic square, the type of the labelling and whether it is affine or cubical. The output of the program is the generating function for this particular labelling. Several examples of counting functions were computed as the result of this thesis. These results revealed computational feasibility of the theoretical approach as well as uncovered some of its shortcomings.

This thesis consist of two parts. The Mathematics of Magic Squares chapter introduces the main underlying concepts and theory behind the computation. The Computational Approach and Implementation chapter describes design and implementation details as well as discusses results produced by this computation. The program was tested by comparing results with existing known results computed by other methods.

1.1 Historical Background

People have been interested in magic squares for a long time. Initial written accounts on the topic date back to almost 5,000 years ago. Chinese literature from as early as 2800 BC [16] contains the legend of the Lo Shu Turtle told in the Book of Rites which is one of five classical texts of ancient China. According to the legend, a great flood took place in China about 3,000 years ago. In order to placate nature, people offered sacrifices to the river. Each time they did that, a turtle with a curious file pattern on its shell emerged from the river. The pattern consisted of circular dots arranged in a 3 by 3 grid such that the number of dots in each row, column and both diagonals added to the number 15. Curiously, this was the same number of days in each of the 24 cycles in a Chinese solar year. Subsequently, people realized that the number of sacrifices should be 15 and that's how they subdued the river. The curious pattern was called the magic square and it plays a major role in the Chinese Art of Harmony called Feng Shui. According to [10] "Many Feng Shui formulas such as Flying Stars, I-Ching and Chinese astrology are based on the magic square." An example of a 3 by 3 magic square is the following pattern:

$$\begin{bmatrix} 4 & 9 & 2 \\ 3 & 5 & 7 \\ 8 & 1 & 6 \end{bmatrix}$$

Since the ancient times, fascination with magic squares continues. Magic squares are the centerpiece of the Tibetan Calendar. According to [8] "symbolically this magic square creates order amidst chaos, and equality in all dimensions." There are numerous sightings on the subject in both Muslem and Jewish numerology as well as Cornelius Agrippa's De Occulta Philosophica from 1510 where he describes spiritual powers of magic squares and produces some squares from order three to nine [1]. Despite its long lasting popularity, magic squares are still being researched today in both pure mathematics as well as popular mathematics and their related fields.

Chapter 2

The Mathematics of Magic Squares

This chapter will outline mathematical background necessary for the computational approach based on work of Matthias Beck and Thomas Zaslavsky in [5] . The problem of counting magic squares will be translated into a mathematical model using a linear-algebra approach. The problem will further be analyzed using the theory of inside-out polytopes and hyperplane arrangements. This analysis will outline the computational approach that will be discussed in subsequent chapters.

2.1 Generalizing the 3 by 3 example

Generalizing the 3 by 3 example introduced in Chapter 1 produces the following matrix:

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} \in \mathbb{Z}_{\geq 0}^{3^2}.$$

In the magic square's case, the constraint of each row's sum being equal to each column's sum and each diagonal's sum, can be represented by the following equations:

$$\begin{aligned} x_{11}+x_{12}+x_{13} &= x_{21}+x_{22}+x_{23} = x_{31}+x_{32}+x_{33} \\ = x_{11}+x_{21}+x_{31} &= x_{12}+x_{22}+x_{32} = x_{13}+x_{23}+x_{33} \\ &= x_{11}+x_{22}+x_{33} = x_{13}+x_{22}+x_{31}. \end{aligned}$$

Thus the magic subspace s is defined as follows:

$$s = \left\{ \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} \in \mathbb{R}^{3^2}_{\geq 0} : \begin{array}{r} x_{11}+x_{12}+x_{13} = x_{21}+x_{22}+x_{23} \\ = x_{31}+x_{32}+x_{33} = x_{11}+x_{21}+x_{31} \\ = x_{12}+x_{22}+x_{32} = x_{13}+x_{23}+x_{33} \\ = x_{11}+x_{22}+x_{33} = x_{13}+x_{22}+x_{31} \end{array} \right\}.$$

Since every sum of each row, column and diagonal is equal, to impose an upper bound on the magic sum (the affine case), an upper bound needs to be imposed on the sum of the first row only:

$$x_{11}+x_{12}+x_{13} = T.$$

Combining this restriction with the magic subspace s produces a polytope $\mathcal{P}$:

$$\mathcal{P} = s \cap \left\{ \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} \in \mathbb{R}^{3^2} : x_{11}+x_{12}+x_{13} = T \right\}.$$

There are two equivalent definitions of a polytope. A *vertex description* of a polytope is as follows: Given a set of points $\mathbf{v}_1, \ldots, \mathbf{v}_n$ in $\mathbb{R}^d$, the polytope is determined by all linear combinations $c_1\mathbf{v}_1+c_2\mathbf{v}_2+\ldots+c_n\mathbf{v}_n$, where the coefficients c_j are non-negative real numbers that satisfy the relation $c_1+c_2+\ldots+c_n = 1$. There is also an alternate, *hyperplane description*, which states that a polytope is a bounded intersection of finitely many half-spaces in $\mathbb{R}^d$ with a half space defined as $\{x \in \mathbb{R}^d : \mathbf{a} \cdot x \leq b\}$. An example of a polytope in 0 dimension is a point, in 1 dimension a line segment, in two dimensions a polygon, and so on. If there is a hyperplane H (defined below) such that $\mathcal{P} \cap H$ consists of a single point then this point is a *vertex* of $\mathcal{P}$. A polytope with integer vertices is called an integer polytope and one with rational vertices is called rational polytope. The polytope $\mathcal{P}$ above is a rational polytope [5].

In the cubical case an upper bound is imposed on the individual entries. Imposing t to be an upper bound on an individual entry results in this constraint:

$$x_{ij} \in [0, t].$$

From here on, our development of the mathematical foundation will concentrate on the affine case. The cubical case is developed similarly, the only difference is whether an upper bound is imposed on the magic sum or on an individual entry in a labelling.

According to the definition of a magic labelling, each coordinate must be distinct, i.e.,

$$x_{ij} \neq x_{km}.$$

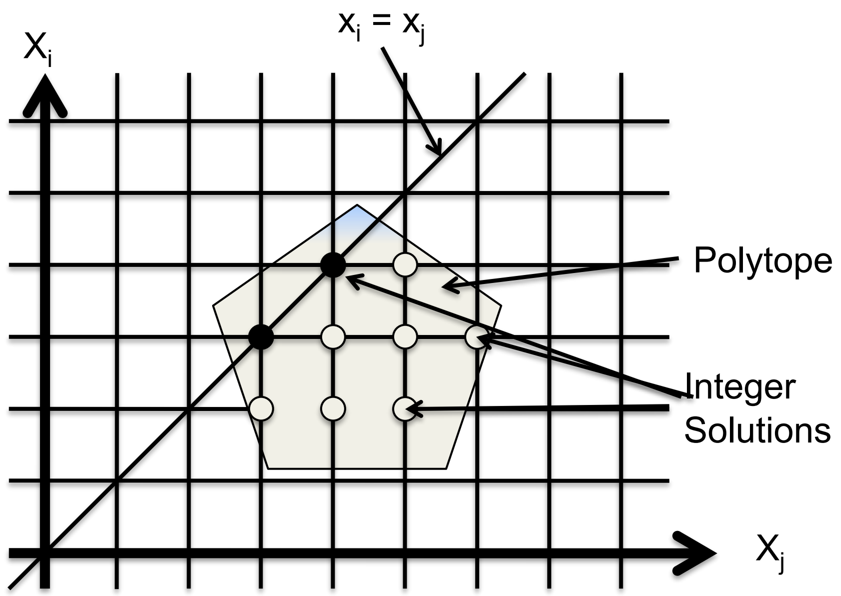
This imposes another constraint on the points. It implies that points with

$$x_{ij} = x_{km}$$

must be removed. This represents an equation of a hyperplane. A formal definition of a hyperplane H is as follows: $H = \{x \in \mathbb{R}^d : \mathbf{a} \cdot x = b\}$ for some $\mathbf{a} \in \mathbb{R}^d \backslash \{0\}, b \in \mathbb{R}$.

A 2-dimensional geometrical representation of a polytope $\mathcal{P}$, integer points and hyperplanes $x_{ij} = x_{km}$ is depicted in Figure 2.1.

Figure 2.1: Integer Points inside Polytope and Hyperplanes



In order to remove points pertaining to these hyperplanes, a notion of hyperplane arrangement is introduced. According to [13], a *hyperplane arrangement* is a finite set of hyperplanes. Let $\mathcal{H}$ denote the hyperplane arrangement in s that will capture the distinctness of the entries requirement [6]:

$$\mathcal{H} = \{(x_{11} = x_{12}) \cap s, (x_{11} = x_{21}) \cap s, \ldots, (x_{32} = x_{33}) \cap s\}.$$

So the formulation of the problem of determining a counting function for the number of magic labellings, once translated into mathematical representation, is converted into a problem of counting the number of non-negative integer points inside a polytope $\mathcal{P}$ excluding points pertaining to hyperplanes $x_{ij} = x_{kl}$. One of the applications of the theory of inside-out polytopes (defined later) does precisely that, it determines counting functions of distinct, positive integer points inside polytopes and uses hyperplane arrangements to exclude solutions pertaining to hyperplanes $x_{ij} = x_{km}$.

2.2 Integer Points Inside Polytopes

In order to count the number of integer points inside a polytope, an introduction of the concept of a lattice is in order. According to [2], it is a discrete collection of equally spaced integer points (a grid) in Euclidean space, mathematically represented as $\mathbb{Z}^d = \{(x_1, \ldots, x_d) : \text{all}\, x_k \,\epsilon\, \mathbb{Z}\}$. The integer points inside a polytope $\mathcal{P}$ are $\mathbb{Z}^d \cap \mathcal{P}$ and the number of such points is represented by $\#\,(\mathbb{Z}^d \cap \mathcal{P})$. Another useful concept in counting integer points inside polytopes is dilation. A polytope can be dilated or shrunk retaining its shape and proportions by a factor of t. A formal definition of dilation is as follows:

A *dilation* of a set $X \subseteq \mathbb{R}^d$ is any set $tX = \{tx : x \in X\}$ for a real number $t > 0$.

The lattice-point enumerator (counting function) of the t^{th} dilate of $\mathcal{P} \subset \mathbb{R}^d$ is denoted by

$$L_{\mathcal{P}}(t) := \#\,(t\mathcal{P} \cap \mathbb{Z}^d) = \#\,(\mathcal{P} \cap t^{-1}\mathbb{Z}^d).$$

It is also known as the *Ehrhart polynomial* if $\mathcal{P}$ is an integer polytope. If $\mathcal{P}$ is rational, than it is an *Ehrhart quasipolynomial* . A *quasipolynomial* is a function $q(t) = \sum_0^d c_i t^i$ with coefficients c_i that are periodic functions of t. In turn the Ehrhart generating function for $L_{\mathcal{P}}(t)$ is:

$$Ehr_{\mathcal{P}}(x) := 1 + \sum_{t \geq 1} L_{\mathcal{P}}(t) x^t. \tag{2.1}$$

An Ehrhart generating function is computed by a program called Latte [9] and is one of two core components in computing the counting function for a magic labelling. The second core component in this computation comes from hyperplane arrangements.

2.3 Hyperplane Arrangements

A hyperplane arrangement will be used to ensure the distinctness requirement $x_{ij} \neq x_{km}$. The hyperplanes in a hyperplane arrangement may intersect each other (or they may not). Intersections may also intersect each other, thus producing intersections of intersections. All of these structures, hyperplanes and their intersections, are called *flats*. The collection of the flats in a hyperplane arrangement $\mathcal{H}$ is called an *intersection semilattice* of the arrangement and is defined as follows:

$$\mathcal{L}(\mathcal{H}) := \left\{ \bigcap \mathcal{S} : \mathcal{S} \subseteq \mathcal{H} \text{ and } \bigcap \mathcal{S} \neq \emptyset \right\}.$$

A very important data of each of these flats is its *Möbius value* which is described by the *Möbius function* which is used for assigning integer values to elements in a poset (partially-ordered set which is a set whose elements are related by some relation $\leq$). For r and s in a poset, the *Möbius function* is defined recursively by [4] :

$$\mu(r,s) := \begin{cases} 0 & \text{if } r > s, \\ 1 & \text{if } r = s, \\ -\sum\limits_{r \leq u < s} \mu(r,u) & \text{if } r < s. \end{cases}$$

Since in the semilattice of a hyperplane arrangement the flats are arranged by reverse inclusion, the Möbius function is defined as follows:

$$\mu(r,s) := \begin{cases} 0 & \text{if } r \subset s, \\ 1 & \text{if } r = s, \\ -\displaystyle\sum_{r \supseteq u \supset s} \mu(r,u) & \text{if } r \supset s. \end{cases}$$

where r and s are flats in the hyperplane arrangement $\mathcal{H}$.

Both of these entities of a hyperplane arrangement, flats and a Möbius value of each flat are used in computing a counting function for a magic labelling and are computed using the THAC [15] program.

2.4 Inside-Out Polytopes

The theoretical foundation of computing a counting function for magic labellings is provided by inside-out polytopes developed in [4]. This paper further combines concepts of counting integer points inside polytopes and hyperplane arrangements and outlines computationally manageable entities that can be computed by the programs Latte and THAC. An *inside-out polytope* $(\mathcal{P}, \mathcal{H})$ is a polytope $\mathcal{P}$ dissected by a hyperpane arrangement $\mathcal{H}$. The goal is to count points of a discrete lattice inside the polytope, but not in any of the hyperplanes. The intersection semilattice $\mathcal{L}(\mathcal{H})$ within a polytope $\mathcal{P}$ is:

$$\mathcal{L}(\mathcal{P}^\circ, \mathcal{H}) := \left\{ \mathcal{P}^\circ \cap \bigcap \mathcal{S} : \mathcal{S} \subseteq \mathcal{H} \right\} \setminus \{\emptyset\},$$

where $\mathcal{P}^\circ$ denotes the interior of $\mathcal{P}$. The counting function of integer points inside of a polytope $\mathcal{P}$ that do not belong to any hyperpanes in a hyperplane arrangement $\mathcal{H}$ is defined as follows:

$$E^\circ{}_{\mathcal{P}^\circ, \mathcal{H}}(t) = \#\left([\mathcal{P}^\circ \setminus \bigcup \mathcal{H}] \cap t^{-1}\mathbb{Z}^d\right).$$

It is the Erhart quasipolynomial of the inside-out polytope $(\mathcal{P}, \mathcal{H})$. $E^{\circ}{}_{\mathcal{P}^{\circ},\mathcal{H}}(t)$ can be computed in terms of ordinary (non inside-out) counting functions [4, Theorem 4.2]:

$$E^{\circ}{}_{\mathcal{P}^{\circ},\mathcal{H}}(t) = \sum_{u \epsilon \mathcal{L}(\mathcal{P}^{\circ},\mathcal{H})} \mu(\hat{0}, u) L_{\mathcal{P}^{\circ} \cap u}(t), \tag{2.2}$$

where $\hat{0}$ denotes the minimum element of $\mathcal{L}(\mathcal{P}^{\circ}, \mathcal{H})$, which is the underlying space; in our case it is s. Equation 2.1 allows us to represent the counting function $L_{\mathcal{P}^{\circ} \cap u}(t)$ in terms of a generating function $Ehr_{\mathcal{P}^{\circ} \cap u}(t)$. Since our computation is in terms of an open Ehrhart generating function $Ehr_{\mathcal{P}^{\circ}}(t)$ and Latte produces a closed Ehrhart generating function $Ehr_{\mathcal{P}}(t)$, one more transformation, given in [4] is in order:

$$Ehr_{\mathcal{P}^{\circ}}(x) = (-1)^{1+\dim \mathcal{P}} Ehr_{\mathcal{P}}(1/x). \tag{2.3}$$

Together these two equations provide a computational recipe for using the programs Latte and THAC in order to compute counting functions for magic labellings. Latte computes the generating functions $Ehr_{\mathcal{P}}(x)$ and THAC computes the individual flats u of $\mathcal{L}(\mathcal{P}^{\circ}, \mathcal{H})$ together with their Möbius value $\mu(\hat{0}, u)$.

2.5 Specific Counting Functions

Equation 2.2 provides a general recipe for determining counting functions for magic labellings. This section outlines counting functions for specific types of magic labellings that are computed as a result of this thesis. The first one, the 3 by 3 magic affine squares, will contain an example of its polytope, its hyperplane arrangement and its generating function to facilitate better understanding of the underlying computational primitives.

2.5.1 3 by 3 Magic Affine

Let $MA_{3,3}(t)$ denote the number of 3 by 3 magic squares with magic sum t. It is a specific case of $E^{\circ}{}_{\mathcal{P}^{\circ},\mathcal{H}}(t)$ where the polytope $\mathcal{P}$ is given by:

$$\begin{aligned}
x_{11}+x_{12}+x_{13}&=1\\
x_{21}+x_{22}+x_{23}&=1\\
x_{31}+x_{32}+x_{33}&=1\\
x_{11}+x_{21}+x_{31}&=1\\
x_{12}+x_{22}+x_{23}&=1\\
x_{13}+x_{23}+x_{33}&=1\\
x_{11}+x_{22}+x_{33}&=1\\
x_{13}+x_{22}+x_{31}&=1
\end{aligned}$$

and the hyperplane arrangement $\mathcal{H}$ is given by:

$$x_{ij} \neq x_{km} \text{ for all } (i,j) \neq (k,m) \text{ and } 1 \leq i,j,k,m \leq 3.$$

The generating function for $\mathbf{MA}_{3,3}(x)$ can be represented in the following derivation:

$$\begin{aligned}
\mathbf{MA}_{3,3}(x) &= \sum_{t\geq 1} MA_{3,3}(t)x^t\\
&= \sum_{t\geq 1} E^{\circ}{}_{\mathcal{P}^{\circ},\mathcal{H}}(t)x^t \qquad \text{using equation 2.2:}\\
&= \sum_{t\geq 1}\sum_{u\epsilon\mathcal{L}(\mathcal{P}^{\circ},\mathcal{H})} \mu(\hat{0},u)L_{\mathcal{P}^{\circ}\cap u}(t)x^t\\
&= \sum_{u\epsilon\mathcal{L}(\mathcal{P}^{\circ},\mathcal{H})} \mu(\hat{0},u)\sum_{t\geq 1} L_{\mathcal{P}^{\circ}\cap u}(t)x^t\\
&= \sum_{t\geq 1} \mu(\hat{0},u)\, Ehr_{\mathcal{P}^{\circ}\cap u}(x).
\end{aligned}$$

where $\mu(\hat{0}, u)$ are computed by THAC and $Ehr_{\mathcal{P}^\circ \cap u}(x)$ are computed by Latte.

2.5.2 3 by 3 Magic Cubical

Let $MC_{3,3}(t)$ denote the number of 3 by 3 magic squares with t being an upper bound on an individual entry in the matrix, and $\mathbf{MC}_{3,3}(x)$ denote its generating function.

2.5.3 3 by 3 Semimagic Affine

Let $SA_{3,3}(t)$ denote the number of 3 by 3 semimagic squares with magic sum t, and $\mathbf{SA}_{3,3}(x)$ denote its generating function.

2.5.4 3 by 3 Semimagic Cubical

Let $SC_{3,3}(t)$ denote the number of 3 by 3 semimagic squares with t being an upper bound on an individual entry in the matrix, , and $\mathbf{SC}_{3,3}(x)$ denote its generating function.

2.5.5 3 x 3 Magilatin Affine

Let $LA_{3,3}(t)$ denote the number of 3 by 3 magilatin squares with magic sum t, and $\mathbf{LA}_{3,3}(x)$ denote its generating function.

2.5.6 3 x 3 Magilatin Cubical

Let $LC_{3,3}(t)$ denote the number of 3 by 3 magilatin squares with t being an upper bound on an individual entry in the matrix, and $\mathbf{LC}_{3,3}(x)$ denote its generating function.

2.5.7 2 x 3 Magilatin Cubical

Let $LC_{2,3}(t)$ denote the number of 2 by 3 magilatin squares with t being an upper bound on an individual entry in the matrix, and $\mathbf{LC}_{2,3}(x)$ denote its generating function.

2.5.8 3 x 3 Latin

Let $L_{3,3}(t)$ denote the number of 3 by 3 latin squares with t being an upper bound on an individual entry in the matrix, and $\mathbf{L}_{3,3}(x)$ denote its generating function.

Chapter 3

Computational Approach and Implementation

This chapter describes the computational approach which is the main contribution of this thesis. It includes design and implementation details. Block diagrams and Unified Modeling Language (UML) will be used to summarize and communicate the design of the implementation.

3.1 High Level Requirements

The requirement of the programming part of the thesis is to develop a stand-alone program to compute generating functions of counting functions for the following magic labellings:

- 3 by 3 Magic Affine
- 3 by 3 Magic Cubical
- 3 by 3 Semimagic Affine
- 3 by 3 Semimagic Cubical
- 3 by 3 Magilatin Affine

- 3 by 3 Magilatin Cubical
- 2 by 3 Magilatin Cubical
- 3 by 3 Latin

A user of the program would be anyone who would want to compute generating functions of counting functions for the above labellings.

Inputs to this program are the dimension of the matrix (-*r* option for number of rows and -*c* option for number of columns), the type of labelling (-*l* option with these labellings: magic, semimagic, magilatin, latin) and the type of computation required (-*t* option with these types: affine or cubical). For a 3 by 3 magic affine case, program invocation with input would look like this:
cgf -r3 -c3 -lmagic -taffne .
The program's output is the generating function of the counting function of the specified labelling. It calculates the number of labellings as a function of the magic sum (the affine case) or the size of an individual entry (the cubical case). For the above 3 by 3 magic affine case it will be:

$$\frac{8x^{15}(2x^3+1)}{(1-x^3)(1-x^6)(1-x^9)}.$$

In developing this software we had to integrate results produced the THAC and Latte as well as use symbolic algebra program Maple perform necessary algebraic manipulations.

3.2 THAC

Tool for Hyperplane Arrangement Characterization (THAC) [15] is an algorithm and program (written in Java) developed as part of a Master's Thesis at San Francisco State University by Eric Etu under Professor Matthias Beck. Its primary intent is to compute the characteristic polynomial of a hyperplane arrangement. THAC's intermediate results, namely a hyperplane arrangement's flats and and their Möbius values are used as computational building blocks.

3.3 Latte

Latte [9] (written in C++) is a computer software dedicated to the problems of counting and detecting lattice points inside convex polytopes, and the solution of integer programs. Latte contains the first ever implementation of Barvinok's algorithm. Latte stands for "Lattice point Enumeration" [9]. The program was developed at the University of California, Davis. Latte is used to compute the generating function of an individual flat produced by the THAC program.

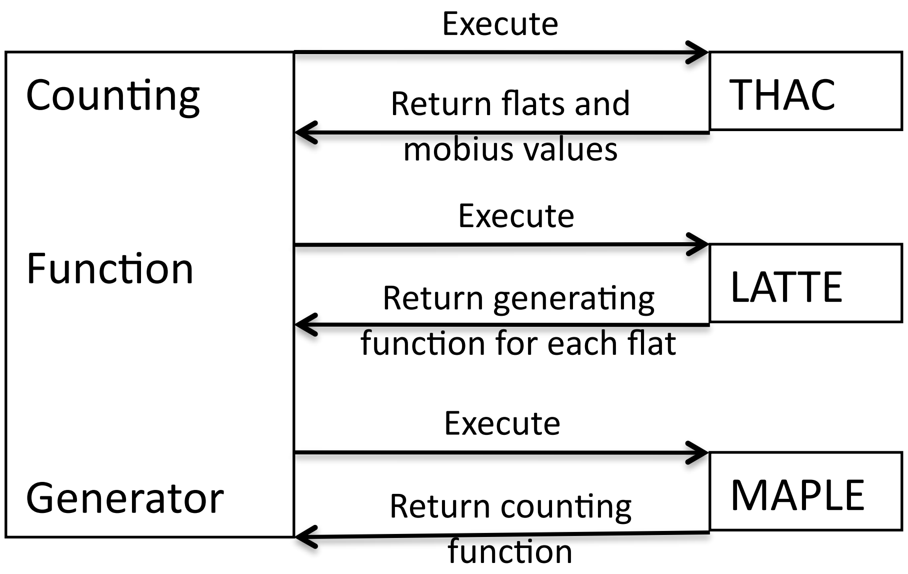
3.4 Maple

Maple [11] (written in C) is a general-purpose symbolic algebra program. It was originally developed at the University of Waterloo in Waterloo, Ontaria, Canada. It has since been commercialized and is distributed by a company called Maplesoft. Maple is used for summing individual generating functions for each flat as well as performing substitutions and algebraic simplifications.

3.5 High Level Approach and Architecture

Section 2.4 presented the building blocks for our computational approach. In particular equation (2.2) outlined individual items that need to be computed and summed. Equation (2.3) is required to make up for the fact that Latte calculates closed generating functions and we need its open counterpart in our computation. We will use the THAC program go compute individual flats u and their Möbius value $\mu(\hat{0}, u)$ of the intersection semilattice $\mathcal{L}(\mathcal{P}^{\circ}, \mathcal{H})$ given by the hyperplane arrangement $\mathcal{H}$ and the subspace s. Then individual flats are fed into the Latte program in the proper format. Latte outputs the generating functions $E_{\mathcal{P}\cap u}(x)$. The Maple program [11] is used for the transformation depicted in 2.3 then multiplying by Möbius value and then summing all individual generating functions together. Figure 3.1 depicts a high level block diagram and information flow for this computation. The Counting Function Generator (GFC) program was developed in conjunction with this thesis. THAC, Latte, and Maple are third party software used for this computation. They were executed as standalone programs. Communication between these programs was achieved by placing their output files on a filesystem and using them as inputs for the next stage of computation.

Figure 3.1: Block Diagram



Additional programs were required to validate some results and verify THAC's output in some cases. Those programs will be explained in the upcoming chapters.

3.5.1 Counting Function Generator

The Counting Function Generator (CGF) acts as a controller of the whole system. It controls other programs as well as the data flow between them by processing a program's output, preparing an input for the next program and executing that program with its input. It starts by executing the THAC program. THAC requires two files as inputs, the hyperplane arrangement $\mathcal{H}$ and the subspace s. As an output, it produces a file with individual flats and their Möbius values. CGF parses this output and creates a node for each of the flats. After creating a file for each node in the format that is suitable for Latte, it executes Latte with that file. Latte returns a generating function for that particular node. After all individual generating functions are computed, a Maple input file is created. The purpose of this file is to capture the algebraic manipulations in 2.3 and to sum all of the counting functions together. Then Maple is executed with this file which produces the final counting function.

3.5.2 Computational Complexity

The Counting Function Generator program has three inputs: a type of a labelling (magic, semimagic or magilatin), whether our computation is affine or cubical, and the dimension of the matrix. It turns out that the dimension of the matrix is the determining factor in assessing computational complexity of CFG. Let the dimension of the matrix be d by d. Then the number of hypeplanes, let's call it n, in the hyperplane arrangement is $n = \binom{d^2}{2} = d^2(d^2-1)/2$. According to [15] the number of flats produced by THAC is $O(2^n)$ and its overall computational complexity is $O(8^n)$. Also, according to [7], Latte's computational complexity is polynomial in bit size of its inputs $A_{i,j}, b_i$ (Latte's inputs will be discussed later) if the dimension is fixed, otherwise it is exponential. Since in our case the dimension d^2 is not fixed, the computational complexity of Latte will also be exponential. Since both THAC and Latte are underlying computational facilities of CGF, and their computational complexity is exponential in d, therefore the computational complexity of CGF is also exponential in d.

3.5.3 Performance Issues

An obvious bottleneck as seen from the previous section is computational complexity of the THAC program. Both the 4 by 4 affine and cubical cases were started and then stopped after about 2 months of computation. In the cubical case the computation was stopped in Dimension 5 while it was intersecting 68568 flats, and in the affine case it was stopped in Dimension 4 where it was intersecting 137086 flats. Computations were performed on a machine with 4 Intel(R) Xeon(TM) 3.00GHz CPU 's running Ubuntu 8.04.4 LTS Linux. This empirically proves a theoretical result predicted on page 107 of [15]: "... (THAC) performance seems to be limited by the CPU, not by memory". One of the remedies suggested in [15] is the parallel computing approach by distributing computation to other computers. Another approach is described in another Master's thesis by Andrew van Herrick [17]. There mixed integer programming together with lattice points counts in rational semi-open polytopes of fixed dimension is used to compute the counting functions for 4 by 4 magic and semimagic squares for affine and cubical cases.

3.6 Implementation

3.6.1 Software Development Environment

Java language was used to develop the Counting Function Generator program. The following versions were used for Development and Runtime environments:

Development:

```
java version ”1.5.0_16”
The Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_16-b06-275)
Java HotSpot(TM) Client VM (build 1.5.0_16-132, mixed mode, sharing)
```

Runtime:

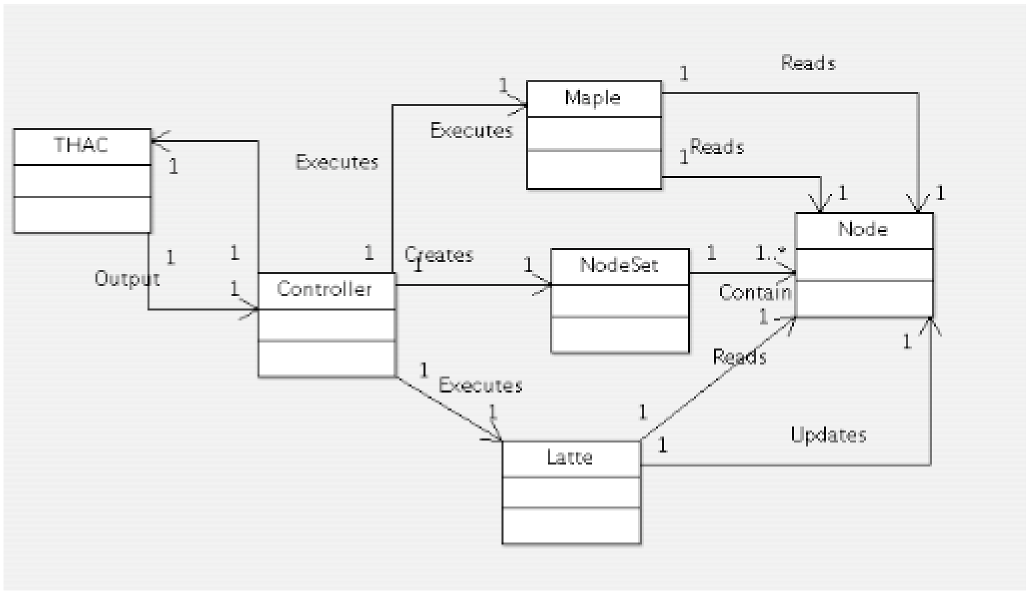
```
java version ”1.5.0_09”
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_09-b01)
Java HotSpot(TM) Server VM (build 1.5.0_09-b01, mixed mode)
```

The software was written using the Eclipse Integrated Development Environment version 3.2.2 .
The software requires compiling. Due to its complicated prerequisite requirements (THAC, Latte, Maple) and limitations because of its computational complexity, the program is not meant for wide distribution.

3.6.2 Class Diagram

A class diagram describes relationship between classes in an implementation. A high-level class diagram for CGF is depicted in the Figure 3.2. The Controller class is responsible for controlling program execution and data flow between other parts of the program. The THAC, Latte, and Maple classes are responsible for preparing the environment and executing their respective programs. In addition, the Latte and Maple classes prepare input files prior to executing their programs. The NodeSet class is responsible for parsing THAC's output and creating a set of nodes (the Node class) for each of the flats in THAC's output. The NodeSet and Node classes are the two most important classes in the CGF program and will be described in more detail.

Figure 3.2: Class Diagram



3.6.3 Sequence Diagram

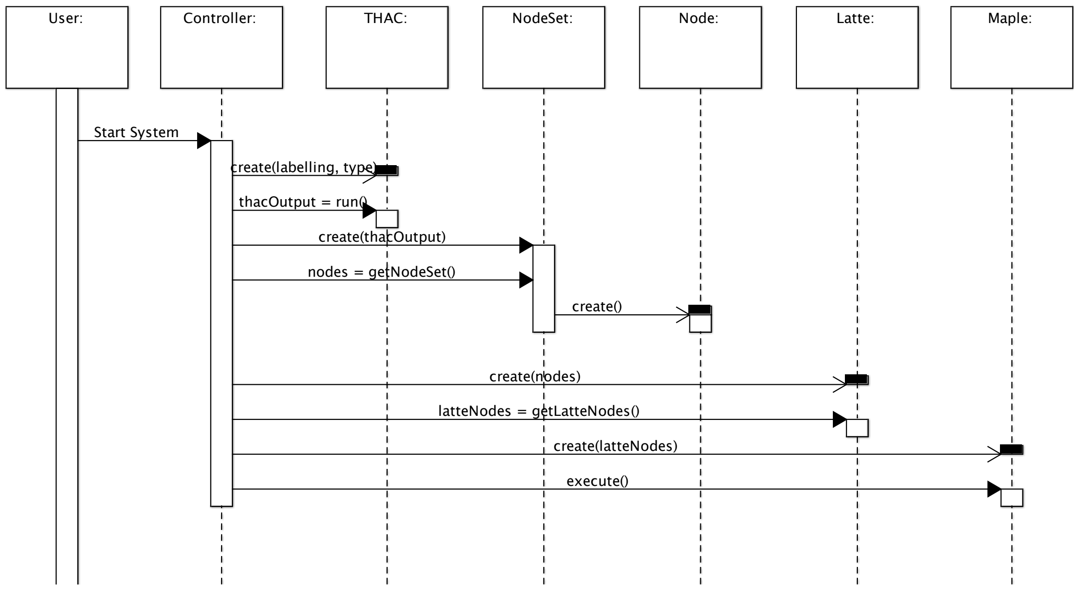
A sequence diagram is used to show how objects interact in a given situation. This diagram is used to represent dynamic interaction of collaborating objects with respect to time. An important characteristic of a sequence diagram is that time passes from top to bottom : the interaction starts near the top of the diagram and ends at the bottom.

The sequence diagram depicted in Figure 3.3 represents program flow and interaction between objects in CGF:

- An end user executes the program supplying the labelling and the type, either affine or cubical with a dimension of the labelling.

- Controller creates a THAC object and executes a THAC run. THAC creates its output file.

Figure 3.3: Sequence Diagram



- Controller creates a NodeSet object and passes THAC's output file as a parameter to NodeSet's constructor.
- NodeSet parses THAC's output and creates a Node object for every flat found in the file
- Controller creates a Latte object and passes a set of created nodes as a parameter to Latte's constructor.
- For each node in the set of nodes, Latte creates an input file for the Latte program and executes the program.
- Latte object captures generating function for each node and adds it to the Node object
- Controller creates a Maple object and passes the set of updated nodes as a param-

eter to Maple's constructor.
- For each node in the set of nodes, Maple object extracts the node's generating function, performs necessary operations and creates a Maple input file. Maple object then executes the Maple program with its input file.

3.6.4 NodeSet Class

The goal of the NodeSet Class is to parse THAC'S output into a set of nodes that represents each flat in the output. Each node is represented by the Node object and NodeSet is a collection called Set of such objects.

A typical THAC output file looks like this:

```
DIMENSION = 4 (1 flats)
----------------


1
5 10
1.0 0.0 0.0 0.0 -1.0 -1.0 -1.0 0.0 0.0 0.0
1.0 0.0 0.0 0.0 0.0 0.0 0.0 -1.0 -1.0 -1.0
1.0 -1.0 0.0 0.0 -1.0 0.0 0.0 -1.0 0.0 0.0
1.0 0.0 -1.0 0.0 0.0 -1.0 0.0 0.0 -1.0 0.0
1.0 0.0 0.0 -1.0 0.0 0.0 -1.0 0.0 0.0 -1.0

DIMENSION = 3 (18 flats)
----------------


-1
```

```
6 10
1.0 0.0 0.0 0.0 -1.0 -1.0 -1.0 0.0 0.0 0.0
1.0 0.0 0.0 0.0 0.0 0.0 0.0 -1.0 -1.0 -1.0
1.0 -1.0 0.0 0.0 -1.0 0.0 0.0 -1.0 0.0 0.0
1.0 0.0 -1.0 0.0 0.0 -1.0 0.0 0.0 -1.0 0.0
1.0 0.0 0.0 -1.0 0.0 0.0 -1.0 0.0 0.0 -1.0
0.0 1.0 -1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0

-1
6 10
1.0 0.0 0.0 0.0 -1.0 -1.0 -1.0 0.0 0.0 0.0
1.0 0.0 0.0 0.0 0.0 0.0 0.0 -1.0 -1.0 -1.0
1.0 -1.0 0.0 0.0 -1.0 0.0 0.0 -1.0 0.0 0.0
1.0 0.0 -1.0 0.0 0.0 -1.0 0.0 0.0 -1.0 0.0
1.0 0.0 0.0 -1.0 0.0 0.0 -1.0 0.0 0.0 -1.0
0.0 1.0 0.0 -1.0 0.0 0.0 0.0 0.0 0.0 0.0
```

It is grouped by dimensions which contains flats and their data. The first line contains the Möbius value. In the dimension 4 above, the Möbius value is 1. The next lines describe the flat itself. It is the standard format to represent a flat used in both THAC and Latte programs. Flats are represented by a system of linear equations $Ax = b$, where $A \in \mathbb{Z}^{mxd}, A = (a_{ij})$, and $b \in \mathbb{Z}^m$. The lines after the Möbius value are formatted as follows:

$$\begin{array}{cc} m & (d+1) \\ b & (-A) \end{array}$$

Thus the first step of parsing this output is to parse it into dimensions, which is what the parseIntoDimensions() method does:

```
void parseIntoDimensions() {
    try {
        BufferedReader br =
                new BufferedReader(new FileReader(inputFile));
        String line = null;
        StringBuffer allText = new StringBuffer("");
        while((line = br.readLine()) != null){
                allText.append(line);
                allText.append("\n");
         }
        Pattern pt  = Pattern.compile("DIMENSION");
        String [] dimTokens = pt.split(allText);
        for(int i = 1; i < dimTokens.length; i++) {
                parseDimIntoFlats(dimTokens[i]);
        }
        }
        catch (Exception ex) {
                ex.printStackTrace();
                System.exit(-1);
        }
}
```

Each dimension, which is contained in an array of dimension tokens dimTokens[], is then passed to the method parseDimIntoFlats(), which in turn parses it into flats:

```
void parseDimIntoFlats(String dim) {
        int numFlats = 0;
        Pattern pt = Pattern.compile("^\n",Pattern.MULTILINE);
        String [] flatTokens = pt.split(dim);
        numFlats = parseDimensionLine(flatTokens[0]);
        if (numFlats != 0) {
                for (int i = 1; i < flatTokens.length
                        && i <= numFlats; i++){
                        parseFlatIntoNode(flatTokens[i]);
                }
        }
}
```

This method starts out by splitting the dimension token into an array of flat tokens flatTokens[] and then passes each of them to the parseFlatIntoNode() method which in turn creates a node of object Node from a flat:

```
void parseFlatIntoNode(String flat) {
        Node nd = new Node(dimension);
        Pattern pt = Pattern.compile("\n");
        String [] flatTokens = pt.split(flat);
        nd.setMobius(parseMobius(flatTokens[0]));
        parseEqnsVars(flatTokens[1]);
        nd.setNumEqns(numEqns);
        nd.setNumVars(numVars);
        colNumber = 0;
        rowNumber = 0;
        tempFlat = new int [numEqns][numVars];
        for (int j = 2; j < flatTokens.length; j++){
                parseRowFlat(flatTokens[j]);
                rowNumber++;
        }
        nd.setFlat(tempFlat);
        nd.setSerialNumber(nodeSerialNumber);
        nodeSerialNumber++;
        ns.add(nd);
}
```

This method uses helper methods to parse Möbius values: parseMobius() extracts the number of equations and variables required for Latte computation: parseEqnsVars(). A flat is also parsed into its constituents and is recreated in the Node class. parseRowFlat() is used for this purpose. The flat will be used as a part of the Latte input.

Finally each node is added to the NodeSet object.

3.6.5 Node Class

The Node Class is used to contain information about each flat parsed from the THAC output file. It contains its associated dimension and Möbius value used in the final Maple computation. It also contains the number of variables and equations as well as each equation in a flat used for Latte computation. It has Set and Get helper methods for these values. This class contains methods to create an input file for Latte as well as to store the generating function created by Latte for each flat. This is what a typical Latte input file looks like for the affine case:

```
8 10
3 0 0 0 -1 -1 -1 0 0 0
3 0 0 0 0 0 0 -1 -1 -1
3 -1 0 0 -1 0 0 -1 0 0
3 0 -1 0 0 -1 0 0 -1 0
3 0 0 -1 0 0 -1 0 0 -1
3 -1 0 0 0 -1 0 0 0 -1
3 0 0 -1 0 -1 0 -1 0 0
```

```
0 1 -1 0 0 0 0 0 0 0
linearity 8 1 2 3 4 5 6 7 8
nonnegative 9 1 2 3 4 5 6 7 8 9
```

The lines before "linearity" and "nonegative" lines describe the flat using the aforementioned format. The fist line contains the number of equations (8) and the number of variables (9) plus 1 (10). Then there are the equations of a flat in the format $b(-A)$. The linearity line specifies how many and which equations are equalities rather than inequalities; all of our equations are equalities. The nonnegative line specifies which variables are non negative; all of our variables are non negative. There is a difference how a Latte input file is created depending on whether we are in an affine or cubical case. In the cubical case an upper bound is established on an individual entry. This constraint is added to the equations part of Latte's input. The computeEquationsLine() method creates Latte's input:

```
private StringBuffer computeEquationsLine() {
    StringBuffer equationsLine = new StringBuffer("");
     for (int i = 0; i < this.flat.length; i++) {
          for (int j = 0; j < this.flat[0].length; j++) {
                equationsLine.append((flat[i][j]) + " ");
          }
          equationsLine.append("\n");
     }
       if(GCF.type.compareToIgnoreCase("Cubical") == 0) {
          if (GCF.dimension_row == 2
               && GCF.dimension_column == 3) {
                equationsLine.append("1 -1 0 0 0 0 0\n");
                equationsLine.append("1 0 -1 0 0 0 0\n");
                equationsLine.append("1 0 0 -1 0 0 0\n");
                equationsLine.append("1 0 0 0 -1 0 0\n");
                equationsLine.append("1 0 0 0 0 -1 0\n");
                equationsLine.append("1 0 0 0 0 0 -1\n");
          }
       //This is the 3 by 3 case
          if (GCF.dimension_row == 3
          && GCF.dimension_column == 3) {
             equationsLine.append("1 -1 0 0 0 0 0 0 0 0 \n");
              equationsLine.append("1 0 -1 0 0 0 0 0 0 0 \n");
              equationsLine.append("1 0 0 -1 0 0 0 0 0 0 \n");
              equationsLine.append("1 0 0 0 -1 0 0 0 0 0 \n");
              equationsLine.append("1 0 0 0 0 -1 0 0 0 0 \n");
              equationsLine.append("1 0 0 0 0 0 -1 0 0 0 \n");
              equationsLine.append("1 0 0 0 0 0 0 -1 0 0 \n");
              equationsLine.append("1 0 0 0 0 0 0 0 -1 0 \n");
              equationsLine.append("1 0 0 0 0 0 0 0 0 -1 \n");
          }
     }
     return equationsLine;
}
```

In the cubical case, the number of equations in the first line also changes. This is reflected in the in the computeFirstLine() method:

```
private StringBuffer computeFirstLine() {
   StringBuffer firstLine = new StringBuffer("");
   int numEqs = 0;
   if (GCF.type.compareToIgnoreCase("Affine") == 0) {
      numEqs = this.getNumEqns();
   }
   else {
        numEqs = this.getNumEqns() + (
        GCF.dimension_row * GCF.dimension_column);
    }
    int numVars = this.getNumVars();
    firstLine.append(numEqs + " " + numVars + "\n");
   return firstLine;
}
```

This is what Latte's input looks like for the cubical case:

```
16 10
0 1 1 1 0 0 0 -1 -1 -1
0 0 1 1 -1 0 0 -1 0 0
0 1 0 1 0 -1 0 0 -1 0
0 1 1 0 0 0 -1 0 0 -1
0 0 1 1 0 -1 0 0 0 -1
0 1 1 0 0 -1 0 -1 0 0
0 1 -1 0 0 0 0 0 0 0
1 -1 0 0 0 0 0 0 0 0
1 0 -1 0 0 0 0 0 0 0
1 0 0 -1 0 0 0 0 0 0
1 0 0 0 -1 0 0 0 0 0
1 0 0 0 0 -1 0 0 0 0
1 0 0 0 0 0 -1 0 0 0
1 0 0 0 0 0 0 -1 0 0
1 0 0 0 0 0 0 0 -1 0
1 0 0 0 0 0 0 0 0 -1
linearity 7 1 2 3 4 5 6 7
nonnegative 9 1 2 3 4 5 6 7 8 9
```

3.6.6 Latte Class

The Latte class is responsible for executing the Latte program, capturing its result and updating the Node object. It uses Java's ProcessBuilder class to execute external programs like Latte. Besides returning a generating function for a flat, Latte may return two special cases: "Integrally empty polytope" or "The number of lattice points is 1". In the "Integrally empty polytope" case the number of lattice points is zero. We dilate the polytope and recompute until we get a result. This is what the dilateNode() method does:

```
public void dialateNode(int dFactor) {
        System.out.println("Dialating Node");
        setDilFactor(dFactor);
        for(int i = 0; i < flat.length; i++) {
                if(flat[i][0] != 0) {
                flat[i][0] = flat[i][0] + 1 ;
                }
        }
}
```

We need to record the dilation factor for further computation in Maple. It is achieved with the setDilFactor(dFactor) line. If the number of lattice points is 1, then the generating function is $1/(1-x)$, because the generating function is given by 2.1. In our case, $L_{\mathcal{P}}(t)$ is 1. Therefore the generating function becomes

$$Ehr_{\mathcal{P}}(x) = \sum_{t\geq 0} L_{\mathcal{P}}(t)x^t = \sum_{t=0}^{\infty} 1{\cdot}x^t = \frac{1}{1-x}$$

Once the generating function is computed, it is parsed and a node is updated.

3.6.7 Maple Class

The Maple class is responsible for performing necessary substitutions for each node (equation 2.3), multiplying by a Möbius value and adding all of the nodes together (equation 2.2). A typical Maple input file for one node looks as follows:

```
(1)  VAR1 := (t^3+1)/(-1+t)/(-1+t^3);
(2)  VAR1:=subs(t=1/(x^3), VAR1);
(3)  VAR1:=(-1)^2*VAR1;
(4)  VAR1:=-1*VAR1;
```

Line 1 is the generating function produced by Latte. Line 2 takes care of the dilating factor (3), line 3 performs substitution in 2.3, and line 4 does the multiplication by Möbius value (-1). Once these manipulations are performed for each node, they are summed up and simplified:

```
Result:=VAR0 + VAR1 + VAR2 + VAR3 + VAR4 + VAR5 + VAR6 + VAR7 +
           + VAR8 + VAR9;
Result:=simplify(Result);
```

3.7 Usage and Results

This section contains the results achieved using the computational approach described above. Results were validated against previously computed generating functions described in [5] except where specifically noted. There is also another paper [3] which describes computation of similar generating functions, but using a different approach.

3.7.1 3 x 3 Magic Affine

The CGF program was invoked in the following manner: *cgf -r3 -c3 -lmagic -taffine*. It produced the following output:

$$\mathbf{MA}_{3,3}(x) = \frac{8x^{15}(2x^3+1)}{(1-x^3)(1-x^6)(1-x^9)}.$$

Correctness of the result was checked against [5].

3.7.2 3 x 3 Magic Cubical

The CGF program was invoked in the following manner: *cgf -r3 -c3 -lmagic -tcubical.* It produced the following output:

$$\mathbf{MC}_{3,3}(x) = \frac{8x^{10}(2x^2+1)}{(1+x^2+x^3+x^4+x^5+x)^3(x^4-1)(x-1)}$$

Correctness of the result was checked against [5].

3.7.3 3 x 3 Semimagic Affine

The CGF program was invoked in the following manner: *cgf -r3 -c3 -lsemimagic -taffine*. It produced the following output:

$$\mathbf{SA}_{3,3}(x) = \frac{72x^{15}\left\{\begin{array}{l} 18x^{21}+5x^{20}+15x^{19}+11x^{17}-8x^{16}+1x^{15}-23x^{14}-13x^{13} \\ -22x^{12}-9x^{11}-16x^{10}+1x^{9}-3x^{8}+7x^{7}+7x^{6}+9x^{5}+7x^{4} \\ +6x^{3}+4x^{2}+2x+1 \end{array}\right\}}{(1-x^3)^2(1-x^4)(1-x^5)(1-x^6)(1-x^7)(1-x^8)}$$

Correctness of the result was checked against [5].

3.7.4 3 x 3 Semimagic Cubical

The CGF program was invoked in the following manner: *cgf -r3 -c3 -lsemimagic -tcubical*. It produced the following output:

$$\mathbf{SC}_{3,3}(x) = \frac{72x^{10}(18x^9 + 46x^8 + 69x^7 + 74x^6 + 65x^5 + 46x^4 + 26x^3 + 11x^2 + 4x + 1)}{(1 - x^2)^2(1 - x^3)^2(1 - x^4)(1 - x^5)}$$

Correctness of the result was checked against [5].

3.7.5 3 x 3 Magilatin Affine

Due to errors in THAC, we could not compute this generating function. The flaw was discovered when the THAC program did not produce the correct characteristic polynomial for the hyperplane arrangement. Further investigation revealed that the flats produced by THAC were not unique (linearly independent) as they should have been. This was checked by computing each flat's rank. Then flats were combined and the rank computed again. The rank of combined flats should have been larger than the rank of the individual flats if those flats were unique. For some of these combinations, the ranks of combined flats were the same as the ranks of the individual flats which implied that the flats were not unique. Since all flats had to be unique in the hyperplane arrangement, this proved that THAC did not compute the flats correctly.

3.7.6 3 x 3 Magilatin Cubical

The CGF program was invoked in the following manner: *cgf -r3 -c3 -lmagilatin -tcubical*. It produced the following output:

$$\mathbf{LC}_{3,3}(x) = \frac{12x^4 \left\{ \begin{array}{l} 79x^{15} + 190x^{14} + 260x^{13} + 250x^{12} + 211x^{11} + 179x^{10} \\ + 181x^9 + 198x^8 + 210x^7 + 181x^6 + 125x^5 + 61x^4 \\ + 22x^3 + 8x^2 + 4x + 1 \end{array} \right\}}{(1-x^4)(1-x^5)(1-x^3)^2(1-x^2)^2}$$

Correctness of the result was checked against [5].

3.7.7 2 x 3 Magilatin Cubical

The quasipolynomial for this counting function is given by Example 4.3 in [5]. The generating function was computed with Maple using the equation (2.1). Maple produced the following result:

$$\mathbf{LC}_{2,3}(x) = \frac{12x^8}{(x-1)^2(x^2-1)^2(x^2+1)}$$

The CGF program was invoked in the following manner: *cgf -r2 -c3 -lmagilatin -tcubical* and produced the same result as Maple.

3.7.8 3 x 3 Latin

The CGF program was invoked in the following manner: *cgf -r3 -c3 -llatin -tcubical*. It produced the following output:

$$\mathbf{L}_{3,3}(x) = \frac{12x^4(1597x^6 + 7374x^5 + 12315x^4 + 7420x^3 + 1455x^2 + 78x + 1)}{(x-1)^2(x^2-2x+1)^4}$$

Since there was no previous result available for comparison, in order to validate the result, a Taylor series expansion of the result was taken around $x = 0$. A coefficient of a term in the expansion is equal to the number of labellings and a power of the term is a value of the cubical parameter. Subsequently, a brute force calculation was used to calculate a number of labellings for a particular cubical value. Labellings were calculated up to the cubical value of 7. This matched the first four terms of the Taylor series expansion:

$$12x^4 + 1056x^5 + 27480x^6 + 317760x^7.$$

3.8 Conclusion

The contribution of this work was to create a program that would illustrate theoretical results of computing a counting function for a number of magic labellings as a function of the magic sum or size of an individual entry in a labelling. Such a program, Counting Function Generator (CGF), was written in the Java language and tested against previously available results. Additionally, a previously unknown counting function was computed for the 3 by 3 latin case. CGF utilized previously existing programs THAC, Latte, and Maple in order to produce its computations. The purpose of CGF was to utilize results produced by these programs and facilitate data-flow management between them. CGF can potentially be useful to other researchers who study problems in the field of magic squares and integer points enumeration inside polytopes by computing other examples which could provide answers or raise more questions in the field.

Along the way, different obstacles were uncovered and resolved. A flaw in THAC was discovered during the 3 by 3 semimagic cubical case computation. The flaw was resolved by the original developer of the program which facilitated successful computation. Unfortunately another flaw in THAC prevented a successfully computing the 3 by 3 semimagic affine case. There were several cases where the Latte program produced unanticipated results. Following advice of one of the Latte developers, we were able to correctly interpret results and make necessary adjustments in the program. Those were the “Integrally empty polytope” and the “The number of lattice points is 1” cases.

Computational complexity is one of the major limitations of the program. Due to its underlying dependency on the THAC program whose computational complexity is exponential in the dimension of the matrix, the computational complexity of CGF is also exponential. This was evident in an attempt to compute the 4 by 4 affine and cubical cases which had to be halted after 2 months of computation. Another limitation is the program’s sheer dependency on its constituents, THAC, Latte and Maple. These programs have their own limitations and they also evolve. This could potentially introduce an unpredictable result in the final computation.

Bibliography

[1] M. Beck, M. Cohen, J. Cuomo, and P. Gribelyuk, *The Number of Magic Squares, Cubes, and Hypercubes*. American Mathematical Monthly 110 (2003), no 8, 707-717.

[2] M. Beck, Sinai Robbins *Computing the Continuous Discretely*. Springer (2007).

[3] M. Beck, A. Van Herick, *Enumeration of 4x4 Magic Squares*, preprint (arXiv:0907.3188) (2009); to appear in Mathematics in Computation

[4] M. Beck, T. Zaslavsky, *Inside-out polytopes*. Advances in Math, 205 (1) (2006), 134-162.

[5] M. Beck, T. Zaslavsky, *An Enumerative Geometry for Magic and Magilatin Labellings*. Annals of Combinatorics 10, no. 4 (2006), 395-413.

[6] M. Beck, T. Zaslavsky, *Six Little Squares and How Their Numbers Grow*, preprint (arXiv:1004.0282v1).

[7] Jesus A. De Loera, Raymond Hemmecke, Jeriemiah Tauzer, Ruriko Yoshida, *Effective Lattice Point Counting in Rational Convex Polytopes*, March 10, 2003

[8] Jain, *Jain MatheMagics*, http://www.jainmathemagics.com/page/1/default.asp, February 2007

[9] Latte (Lattice Point Enumeration), University of California Davis, http://www.math.ucdavis.edu/ latte/ , October 2007

[10] Magic's Existence, Feng Shui, Art and Well Being, http://www.magicl-existence.com/ , February 2007

[11] Maple, Symbolic Algebra Program, University of Waterloo, Canada

[12] A. F. Möbius , *Über eine besondere Art von Umkehrung der Reihen.* J. Reine Angew. Math. 9, 105-123, 1832.

[13] R. P. Stanley, *An introduction to hyperplane arrangements*, Geometric Combinatorics (E. Miller, V. Reiner, and B. Sturmfels, eds.), IAS/Park City Mathematics Series, vol.13, American Mathematical Society, Providence, RI, 2007, pp. 389-496.

[14] J. Steiner, *Einige Satze über die Teilung der Ebene und des Raumes*, J. Reine Angew. Math. 1 (1826), 349-364.

[15] THAC (Tool for Hyperplane Arrangement Characterization), San Francisco State University, http://sourceforge.net/projects/thac/ , October 2007

[16] "The History of Feng Shui", http://www.squidoo.com/fengshuistyle/ , February 2007

[17] A. Van Herrick, *Theoretical and computational methods for lattice point enumeration in inside-out polytopes*, MA thesis, San Francisco State University, 2007. Available at http://math.sfsu.edu/beck/teach/masters/andrewv.pdf.

[18] T. Zaslavsky, Facing up to arrangements: *Face count formulas for partitions of space by hyperplanes*, Mem. Amer. Math. Soc. 154 (1975).