Rigorous Experimental Mathematics Applied to the Goulden-Jackson Method , Construction of Symmetric Chains and the Sprague-Grundy Function

---

A Dissertation
Submitted to
the Temple University Graduate Board

---

in Partial Fulfillment
of the Requirements for the Degree of
DOCTOR OF PHILOSOPHY

---

by
Xiangdong Wen
May, 2005

ABSTRACT

Rigorous Experimental Mathematics Applied to the Goulden-Jackson Method , Construction of Symmetric Chains and the Sprague-Grundy Function

Xiangdong Wen

DOCTOR OF PHILOSOPHY

Temple University, May, 2005

Professor Shiferaw Berhanu, Chair

Professor Doron Zeilberger, Co-Chair

Experimental mathematics is a type of mathematical investigation in which computation is used to investigate mathematical structures and identify their fundamental properties and patterns. As in other experimental sciences, experimental mathematics can be used to make mathematical predictions which can then be verified or falsified on the bases of additional computational experiments.

In this thesis, we apply computer technology to three problems: 1. Discovering and proving Symmetric Chain Decompositions for Young's Lattices $L(3,n)$ and $L(4,n)$; 2. Making Extensions of the Goulden-Jackson Cluster method; 3. Finding and proving additive periods for the Sprague-Grundy function of Wythoff's game.

ACKNOWLEDGEMENTS

It's a great pleasure here to thank all the people who have been helping me for five and a half years in Temple University. I am deeply indebted to my thesis advisor Doron Zeilberger, who introduced me to the fascinating world of combinatorics and the exciting area of experimental mathematics. I want to thank Prof. Marvin Knopp for his invaluable comments and suggestions, and to the professors who taught me: Prof. Shiferaw Berhanu, Prof. Edward Letzter, Prof. Cristian Gutiérrez, Prof. Sinai Robins, Prof. Wei-Shih Yang, Prof. Longin Latecki, Prof. Eric Grinberg, Prof. Omar Hijab, Prof. Richard Beigel, Prof. Igor Rivin, Prof. Bruce Conrad and other Temple faculty and staff who helped me during these years.

In the end, I want to thank my wife who gave me constant support to accomplish this degree.

DEDICATION

To my wife and to my parents

TABLE OF CONTENTS

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Nowadays, with the development of computer technology and the enhancement of the work of mathematicians, a new approach to mathematics, using computing technology in mathematical research, which is often called experimental mathematics, has provided new and effective ways for problem-solving. Computers can serve as a "laboratory" for mathematicians, in which he/she can perform experiments such as analyzing examples, testing out new ideas, and/or searching for patterns.

Computers have been utilized by mathematicians to solve many problems which otherwise would be extremely time-consuming or impossible to solve. Mathematicians can use computers to prove theorems automatically, obtain proofs for theorems which are difficult to prove with traditional approaches (e.g. the Four-Color Problem [28]). Computer graphics are also useful in searching for patterns (e.g. Cellular Automata [27]). With the help of computers, mathematicians can observe

the structures of different systems with parameters changed, solve differential equations, and compute integrals.

Computers can perform two main functions in the field of mathematics: carring out numerical calculations and presenting new areas of research. Thanks to the efficiency of computers, a mathematician can gather a great amount of different data and facts concerning the problems of his/her interest. Moreover, with the aid of computers, mathematicians could carry out different “tests” to find the results and patterns of mathematical properties.

Experimental mathematics is a type of mathematical investigation in which computation is used to investigate mathematical structures and to identify their fundamental properties and patterns. As in other experimental sciences, experimental mathematics can be used to make mathematical predictions which can then be verified or falsified on the basis of additional computational experiments. Borwein and Bailey ([3],[23]) use the term “experimental mathematics” to mean the methodology of doing mathematics that includes the use of computation for:

- Gaining insight and intuition;
- Discovering new patterns and relationships;
- Using graphical displays to suggest underlying mathematical principles;
- Testing and especially falsifying conjectures;
- Exploring a possible result to see if it is worth a formal proof;

- Suggesting approaches for a formal proof;
- Replacing lengthy hand derivations with computer-based derivations;
- Confirming analytically derived results.

In addition, sometimes it is possible to do compeltely rigorous **computer generated research**, for example, in the Wilf-Zeilberger theory([20]). The present thesis is in a similar vein, but applied to different kinds of problems. In this thesis, we apply the computer generated research to three problems:

- Finding and proving the Symmetric Chain Decompositions for the Young's lattices;
- Making extensions of the Goulden-Jackson Cluster method;
- Finding and proving the additive periods for the Sprague-Grundy function.

1.1 Finding SCD for the Young's Lattices

The famous Young's partition lattice $L(m,n)$ consists of the set of integer-vectors

$$(a_1, a_2, \cdots, a_m), \quad 0 \leq a_1 \leq a_2 \leq \cdots \leq a_m \leq n,$$

with the order relation

$$\vec{a} \leq \vec{b} \quad \text{if} \quad a_i \leq b_i \quad \text{for} \quad i = 1, 2, \cdots, m.$$

The rank r is defined by

$$r(\vec{a}) = \sum_{i=1}^{m} a_i.$$

And a chain $\vec{v_1} \leq \vec{v_2} \leq \cdots \leq \vec{v_k}$ in $L(m,n)$ is called saturated if it skips no ranks and is called symmetric if

$$r(\vec{v_1}) + r(\vec{v_k}) = mn.$$

A Symmetric Chain Decomposition (SCD) of a poset is a way of expressing it as a disjoint union of saturated symmetric chains.

One of the major problems in order theory is the explicit construction of SCD for Young's Lattice for *all* m and n. In 1989, Kathy O'Hara ([19], see also [30]) astounded the combinatorial world by constructing SCD for the 'trivial extension' of $L(m,n)$, in which all partitions of one rank are related to the next; but the problem remains wide open for Young's lattice itself.

SCDs for L(4,n) and L(3,n) have been constructed by West([26]) and Lindström([15]). In this thesis we explicitly provide **complete** SCDs for $L(4,n)$ and $L(3,n)$, which were found with the assistance of our computer. And far more interestingly, The proof is *completely automatic* without any human help (except for writing the general Maple program).

We hope the present approach will ultimately lead to computer - generated or at least computer-assisted constructions of SCDs for $L(m,n)$, or at least for $L(5,n)$.

Meanwhile we are unable to do the case of $L(5, n)$. We also hope the present *methodology* will be useful for future attacks on this challenging and tantalizing problem.

1.2 Making Extensions of the Goulden-Jackson Cluster Method

Let $V = \{a_1, a_2, \cdots, a_d\}$ be a finite alphabet and B be a finite set of $words$ (on V). Suppose $q(n)$ is the total number of words with length n that avoid the words in B as factors. The aim is to find the generating function

$$f(t) = \sum_{n=0}^{\infty} q(n)t^n \tag{1.1}$$

in an efficient way.

The Goulden - Jackson cluster method ([11],[12]), which is widely used in solving this kind of problem, has been beautifully explained, extended, and implemented by J. Noonan and D. Zeilberger ([18]). However, their Maple packages require that the cardinality of the alphabet is a *numeric* argument rather than *symbolic*. In chapter 3 we extended the method into the latter case, thereby initiating the *Symbolic Goulden-Jackson Method*.

Another observation is that much of the information is lost in 1.1, since the

single variable t only accounts for the *length* . In order to keep track of the *order*, we set up the general generating function

$$g(a_1, a_2, \cdots, a_d) = \sum_{n=0}^{\infty} \sum_{w_1 w_2 \cdots w_n \in \mathcal{L}(B)} w_1 \cdot w_2 \cdot w_3 \cdot \cdots \cdot w_n \,, \tag{1.2}$$

where the operation "$\cdot$" is noncommutative. The generating function (1.2) preserves not only the information of letters in a word but also the order of the letters. Actually it contains all the information of the language $\mathcal{L}(B)$, all the words that avoid words in B as factors. In chapter 4, we develop the *noncommutative Goulden-Jackson method* to find a rational form of the function (1.2).

At Chapter 5 we apply the cluster method to the two dimensional cases.

1.3 Finding and Proving the Additive Periods for the Sprague-Grundy Function of Wythoff's Game

Generally experimental mathematics needs the following steps to find certain properties and rules for mathematical expressions: Given a $Problem(n)$, parameterized by integer n, apply a standard (or new) numerical algorithm to get the $Answers(n)$ for $n = 1, 2, \cdots, 100$. Then have the human, or much better still, the machine, guess the symbolic answer $Answer(n)$, for symbolic n. Finally, use the machine to prove the guess automatically. In chapter 6, we provide a way to use the

computer to find the additive periods for the Sprague-Grundy function of Wythoff's game and get an automatic proof.

CHAPTER 2

COMPUTER-GENERATED SCD FOR $L(3,n)$ AND $L(4,n)$

2.1 Introduction

Recall that the famous *Young's partition lattice* $L(m,n)$ consists of the set of integer-vectors

$$(a_1, a_2, \cdots, a_m), \quad 0 \le a_1 \le a_2 \le \cdots \le a_m \le n,$$

with the order relation

$$\vec{a} \le \vec{b} \quad \text{if} \quad a_i \le b_i \quad \text{for} \quad i = 1, 2, \cdots, m.$$

The **rank** r is defined by

$$r(\vec{a}) = \sum_{i=1}^{m} a_i.$$

And recall that a chain $\vec{v_1} \leq \vec{v_2} \leq \cdots \leq \vec{v_k}$ in $L(m,n)$ is called **saturated** if it skips no ranks and is called **symmetric** if

$$r(\vec{v_1}) + r(\vec{v_k}) = mn.$$

A **Symmetric Chain Decomposition**(SCD) of a poset is a way of expressing it as a disjoint union of saturated symmetric chains.

One of the major problems in order theory is the explicit construction of SCDs for Young's Lattice for *all* m and n. In 1989, Kathy O'Hara ([19], see also [30]) astounded the combinatorial world by constructing SCDs for the 'trivial extension' of $L(m,n)$, in which all partitions of one rank are related to the next; but the problem remains wide open for Young's lattice itself.

SCDs for L(4,n) and L(3,n) have been constructed by West([26]) and Lindström([15]). In this paper we explicitly provide **complete** SCDs for $L(4,n)$ and $L(3,n)$, which were found by the assistance of our computer. And far more interestingly, it is proved *completely automatically* without using any human help (except for writing the general Maple program). While our construction for $L(4,n)$ is not equivalent to West's construction, it is nevertheless of the similar format. On the other hand, our construction for $L(3,n)$ is more elegant than Lindström's, since

it is not split into even and odd cases.

We hope the present approach will ultimately lead to computer-generated, or at least computer-assisted, constructions of SCDs for $L(m, n)$, or at least for $L(5, n)$. Meanwhile we are unable to do the case of $L(5, n)$. We also hope the present *methodology* will be useful for future attacks on this challenging and tantalizing problem.

2.2 New SCDs for L(3,n) and L(4,n)

Theorem 1 *Table 2.1 and Table 2.2 give symmetric chain decompositions for*

$$L(3, n) \text{ and } L(4, n)$$

respectively, where i, j and k are generic non-negative integers and vertical dots represent that the only component that is not the same gets decreased by 1. For example,

$$(n - i - 3j, n - i - 2j, n - j) \ldots (i + j, n - i - 2j, n - j)$$

is a shorthand for the chain:

$$(n - i - 3j - a, n - i - 2j, n - j), a = 0 \ldots n - 2i - 4j.$$

Table 2.1: A complete SCD for L(3,n)

C_{ij} $2i+4j \le n$			D_{ij} $2i+4j+3 \le n$		
(n-i-3j,	n-i-2j,	n-j)	(n-i-3j-1,	n-i-2j-1,	n-j-1)
⋮					⋮
(i+j,	n-i-2j,	n-j)	(n-i-3j-1,	n-i-2j-1,	n-i-j-1)
	⋮		⋮		
(i+j,	i+2j,	n-j)	(j,	n-i-2j-1,	n-i-j-1)
		⋮		⋮	
(i+j,	i+2j,	i+3j)	(j,	i+2j+1,	n-i-j-1)
⋮					⋮
(j,	i+2j,	i+3j)	(j,	i+2j+1,	i+3j+2)

Proof: The chains are clearly saturated and symmetric. Thus we only need to prove that each vector in $L(m,n)$ $(m=3,4)$ appears only once in the tables. We introduce the commuting indeterminate $x_1, x_2, \cdots, x_m$ and t, and define the **weight** for a vector $\vec{a}=(a_1,a_2,\cdots,a_m)$ in $L(m,n)$ as the following:

$$\mathrm{w}(\vec{a};x_1,x_2,\cdots,x_m;t)=(x_0t)^{n-a_m}(x_1t)^{a_m-a_{m-1}}\cdots(x_{m-1}t)^{a_2-a_1}(x_mt)^{a_1}.$$

For a fixed m, it is easy to see that the total weight,

$$\sum_{n=0}^{\infty}\mathrm{w}(\vec{a}),\quad \text{where}\quad \vec{a}\in L(m,n),$$

is a generating function

$$G(t;x_1,x_2,\cdots,x_m)=\frac{1}{(1-x_0t)(1-x_1t)\cdots(1-x_mt)}.$$

Table 2.2: A complete SCD for L(4,n)

C_{ijk} $2i+2j+3k \leq n$			
(n-2k-2j-i,	n-2k-j-i,	n-k-j,	n-k)
⋮			
(k+i,	n-2k-j-i,	n-k-j,	n-k)
	⋮		
(k+i,	k+j+i,	n-k-j,	n-k)
		⋮	
(k+i,	k+j+i,	2k+j+i,	n-k)
			⋮
(k+i,	k+j+i,	2k+j+i,	2k+2j+i)
⋮			
(k,	k+j+i,	2k+j+i,	2k+2j+i)
	⋮		
(k,	k+j,	2k+j+i,	2k+2j+i)

D_{ijk} $2i+2j+3k+3 \leq n$			
(n-2k-2j-i-1,	n-2k-j-i-1,	n-k-j-1,	n-k)
		⋮	
(n-2k-2j-i-1,	n-2k-j-i-1,	n-k-i-j-1,	n-k)
			⋮
(n-2k-2j-i-1,	n-2k-j-i-1,	n-k-i-j-1,	n-k-i-1)
⋮			
(k,	n-2k-j-i-1,	n-k-i-j-1,	n-k-i-1)
	⋮		
(k,	k+j,	n-k-i-j-1,	n-k-i-1)
		⋮	
(k,	k+j,	2k+j+i+1,	n-k-i-1)
			⋮
(k,	k+j,	2k+j+i+1,	2k+2j+i+2)

Each term in the expanded power series of $G(t; x_1, x_2, \cdots, x_m)$ has coefficient 1 and corresponds to a unique vector in $L(m, n)$. On the other hand, for each vector in $L(m, n)$, there is a unique corresponding term in the power series of $G(t; x_1, x_2, \cdots, x_m)$. Therefore, to prove that each vector in $L(m, n)$ $(m = 3, 4)$ appears only once in the SCDs is the same as to prove that the total weights of the vectors in the given chains are $G(t; x_1, x_2, x_3)$ and $G(t; x_1, x_2, x_3, x_4)$ respectively. The part of summing over all the weights of the vectors is done by computer. ■

This method of proof can be applied to any conjectured SCD. The difficult part is to find such decompositions. Here we need human-computer interactions by using a modified greedy algorithm. Once it is found, the verification part is purely automatic by using the Maple program `Lmn`. `Lmn` can also be used to give completely automatic proofs of the validity of Lindström's and West's constructions. For general m,n, an explicit construction of SCDs of $L(m, n)$ is still an open problem.

2.3 The Maple Package

The summation of all the weights of the vectors in the given chains is automatically done by computer. The Maple package is available at

`http://www.math.temple.edu/~wen/lattice/ .`

After downloading the file to the local disk, type

```
read(‘‘Lmn");
```

in the Maple workspace. There is detailed on-line help on how to use the procedures in the package `Lmn`. Procedures to compute the total weights of the vectors in SCDs given by Lindström[15] and by West[26] are also included in the package `Lmn`.

CHAPTER 3

THE SYMBOLIC GJ CLUSTER METHOD

3.1 Introduction

Let V be a finite alphabet $v = \{a_1, a_2, \cdots, a_d\}$ and let B be a finite set of $words$ (on V). Suppose $q(n)$ is the total number of words with length n that avoid the words in B as factors. The aim is to find the generating function

$$f(t) = \sum_{n=0}^{\infty} q(n)t^n \tag{3.1}$$

in an efficient way.

The Goulden-Jackson cluster method([11],[12]), which is widely used in solving

these kinds of problems, has been beautifully explained, extended, and implemented by J. Noonan and D. Zeilberger ([18]). However, their Maple packages require that the cardinality of the alphabet is a *numeric* argument rather than *symbolic*. In this chapter we extended the method into the latter case, thereby initiating the *Symbolic Goulden-Jackson Method*.

3.2 Review of the Goulden-Jackson Cluster Method

Recall that a **factor** of the word $w_1 w_2 \cdots w_n$ is one of the words

$$w_i w_{i+1} \cdots w_{j-1} w_j, \quad 1 \leq i \leq j \leq n$$

that we shall denote by $[i, j]$. Two factors $[i, j]$ and $[i', j']$ **overlap** if they have at least one common letter.

The **length** of the word $w = w_1 w_2 \cdots w_n$ is $|w| = n$; and the **weight** of the word w is $weight(w) = t^{|w|} = t^n$. Obviously, the generating function (3.1) is the same as

$$f(t) = \sum_{w \in \mathcal{L}(B)} weight(w),$$

where $\mathcal{L}(B)$ is the set of all words over V that avoid the members of B as factors.

A word with some factors marked is called a **marked word**. Here we only consider the case when the marked factors are the words in B. A marked word can be written

in the following form:

$$(w; [i_1, j_1], [i_2, j_2], \cdots, [i_l, j_l]), \text{ where } [i_r, j_r], 1 \leq r \leq l \text{ are marked factors.}$$

For example, let $V = \{1, 2, 3\}$, $B = \{123, 231, 312\}$ and $w = 12312$. There are totally 2^3 marked words for w:

$$(12312;), \quad (12312; [1, 3]), \quad (12312; [2, 4]),$$
$$(12312; [3, 5]), \quad (12312; [1, 3], [2, 4]), \quad (12312; [1, 3], [3, 5]),$$
$$(12312, [2, 4], [3, 5]), \quad (12312; [1, 3], [2, 4], [3, 5]).$$

Define the $\overline{\mathbf{weight}}$ of a marked word w with marked factors S, $S \subset B$ as

$$\overline{weight}(w, S) = (-1)^{|S|} t^{|w|},$$

where $|S|$ is the cardinality of S.

Let V^* be the set of all words generated by V; and Let

$$B(w) := B \bigcap \bigcup_{0 \leq i \leq j \leq n} \{(w, [i, j])\}.$$

We have

Theorem 2 *:*

$$f(t) = \sum_{w \in \mathcal{L}(B)} weight(w) = \sum_{w \in V^*} \sum_{S \subset B(w)} \overline{weight}(w, S). \tag{3.2}$$

Proof: The basic idea in the proof is to use the inclusion-exclusion principle.

$$\begin{aligned}
f(t) &= \textstyle\sum_{w \in \mathcal{L}(B)} weight(w) \\
&= \textstyle\sum_{w \in V^*} weight(w) 0^{|B(w)|} \text{ define } 0^0 = 1 \\
&= \textstyle\sum_{w \in V^*} t^{|w|} [1 + (-1)]^{|B(w)|} \\
&= \textstyle\sum_{w \in V^*} t^{|w|} \sum_{S \subset B(w)} (-1)^{|S|} \\
&= \textstyle\sum_{w \in V^*} \sum_{S \subset B(w)} (-1)^{|S|} t^{|w|} \\
&= \textstyle\sum_{w \in V^*} \sum_{S \subset B(w)} \overline{weight}(w, S).
\end{aligned}$$

■

By the theorem, the calculation of the generating function (3.1) is then transfered to finding the generating function for the weighted marked words (3.2) which is much easier to weight-count by the Goulden-Jackson cluster method.

A **cluster** is a marked word

$$(w_1 w_2 \cdots w_n; [i_1(=1), j_1], [i_2, j_2], \cdots, [i_l, j_l(=n)]),$$

where $[i_k, j_k]$ overlaps with $[i_{k+1}, j_{k+1}]$ for all $k = 1, 2, \cdots, l-1$.

A nonempty marked word either ends with a letter that is not part of a cluster, or ends with a cluster. Peelingoff the maximal cluster, we get a shorter marked word. Thus we have the following decomposition

$$\mathcal{M} = \{empty - word\} \cup \mathcal{M}V \cup \mathcal{M}\mathcal{C}.$$

Taking weights on both sides and solving for $\overline{weight}(\mathcal{M})$, we have

$$f(t) = \overline{weight}(\mathcal{M}) = \frac{1}{1 - d\,t - \overline{weight}(\mathcal{C})} \quad . \tag{3.3}$$

The only step left is to find $\overline{weight}(\mathcal{C})$.

For a given word $w = w_1 w_2 \cdots w_n$, let $HEAD(w)$ be the set of all proper prefixes:

$$HEAD(w) := \{w_1 w_2 \cdots w_k | k = 1, 2, \cdots, n-1\},$$

and $TAIL(w)$ be the set of all proper suffixes

$$TAIL(w) := \{w_k w_{k+1} \cdots w_n | k = 2, 3, \cdots, n\},$$

and let

$$OVERLAP(u, v) := TAIL(u) \cap HEAD(v).$$

Let u/v denote the operation of chopping the head v from the word u. For example: $12321/12 = 321$. Let

$$(u : v) := \sum_{x \in OVERLAP(u,v)} \overline{weight}(v/x).$$

The set of clusters $\mathcal{C}$ can be partitioned into

$$\mathcal{C} = \bigcup_{v \in B} \mathcal{C}[v]\,,$$

where $\mathcal{C}[v]$, $v \in B$ is the set of clusters whose last entry is v.

Given a cluster in $\mathcal{C}[v]$, $v \in B$, it either consists of just v or chopping v results in a shorter cluster in $\mathcal{C}[u]$, $u \in B$ if $OVERLAP(u, v)$ is not empty. On the other hand, given a cluster in $\mathcal{C}[u]$, we can always reconstitute the bigger cluster in $\mathcal{C}[v]$ by adding some words in $\bigcup_{x \in OVERLAP(u,v)} \{v/x\}$. Hence, there exists a bijection:

$$\mathcal{C}[v] \leftrightarrow \{(v; [1, |v|])\} \bigcup_{u \in B} \mathcal{C}[u] OVERLAP(u, v).$$

Taking weights on both sides, we have:

$$\overline{weight}(\mathcal{C}[v]) = (-1)\,\overline{weight}(v) - \sum_{u \in B} (u : v)\,\overline{weight}(\mathcal{C}[u]). \tag{3.4}$$

This is a system of $|B|$ linear equations with $|B|$ unknowns $\overline{weight}(\mathcal{C}[v]), v \in B$.

After solving these equations we can simply obtain $\overline{weight}(\mathcal{C})$ by:

$$\overline{weight}(\mathcal{C}) = \sum_{v \in B} \overline{weight}(\mathcal{C}[v]).$$

Because $\overline{weight}(\mathcal{C})$ is independent of the cardinality of the alphabet, the symbolic Goulden Jackson can be easily implemented.

3.3 Symmetric Cases

Given an alphabet $V = \{1, 2, 3\}$, let us find the generating function for the number of words which do not have three consecutive different letters as factors, i.e.

$$B = \{123, 132, 213, 231, 312, 321\}.$$

By the original Goulden-Jackson cluster method, we need to set up and solve a system of $|B| = 6$ linear equations with six unknowns $\overline{weight}(\mathcal{C}[v])$, $v \in B$:

$$\begin{cases} \overline{weight}(\mathcal{C}[123]) &= -t^3 - t^2\,\overline{weight}(\mathcal{C}[312]) - t^2\,\overline{weight}(\mathcal{C}[321]) \\ & \quad -t\,\overline{weight}(\mathcal{C}[231]) \\ \overline{weight}(\mathcal{C}[132]) &= -t^3 - t^2\,\overline{weight}(\mathcal{C}[231]) - t^2\,\overline{weight}(\mathcal{C}[213]) \\ & \quad -t\,\overline{weight}(\mathcal{C}[321]) \\ \overline{weight}(\mathcal{C}[213]) &= -t^3 - t^2\,\overline{weight}(\mathcal{C}[312]) - t^2\,\overline{weight}(\mathcal{C}[321]) \\ & \quad -t\,\overline{weight}(\mathcal{C}[132]) \\ \overline{weight}(\mathcal{C}[231]) &= -t^3 - t^2\,\overline{weight}(\mathcal{C}[123]) - t^2\,\overline{weight}(\mathcal{C}[132]) \\ & \quad -t\,\overline{weight}(\mathcal{C}[312]) \\ \overline{weight}(\mathcal{C}[312]) &= -t^3 - t^2\,\overline{weight}(\mathcal{C}[213]) - t^2\,\overline{weight}(\mathcal{C}[231]) \\ & \quad -t\,\overline{weight}(\mathcal{C}[123]) \\ \overline{weight}(\mathcal{C}[321]) &= -t^3 - t^2\,\overline{weight}(\mathcal{C}[123]) - t^2\,\overline{weight}(\mathcal{C}[132]) \\ & \quad -t\,\overline{weight}(\mathcal{C}[213]) \end{cases}.$$

By the symmetry of B, all the clusters $\mathcal{C}[v], \quad v \in B$, have the same generating function $\overline{weight}(\mathcal{C}[123])$. Thus we can reduce these six equations to only one equation:

$$\overline{weight}(\mathcal{C}[123]) = -t^3 - 2t^2\,\overline{weight}(\mathcal{C}[123]) - t\,\overline{weight}(\mathcal{C}[123]).$$

After solving it, we have

$$\overline{weight}(\mathcal{C}) = 6\,\overline{weight}(\mathcal{C}[123]) = \frac{-6t^3}{1+2t^2+t},$$

and

$$f(t) = \frac{1}{1-3t-\frac{-6t^3}{1+2t^2+t}} = -\frac{2t^2+t+1}{t^2+2t-1}.$$

Assuming the cardinality of the alphabet V is a symbol d, $V = \{1, 2, 3, \cdots d\}$, let us find the generating function for the number of words which do not have three consecutive different letters as factors, i.e.

$$B = \{123, 124, 125, \cdots, d(d-1)(d-3), d(d-1)(d-2)\}.$$

By the original Goulden-Jackson cluster method, we need to set up and solve a system of $|B| = d(d-1)(d-2)$ linear equations. Using the symmetry of B, we only need to set up and solve one equation:

$$\begin{aligned}\overline{weight}(\mathcal{C}[123]) &= -t^3 - (d-1)(d-2)t^2\,\overline{weight}(\mathcal{C}[123]) \\ &\quad -(d-2)t\,\overline{weight}(\mathcal{C}[123]).\end{aligned}$$

Thus,

$$\begin{aligned}\overline{weight}(\mathcal{C}) &= d(d-1)(d-2)\,\overline{weight}(\mathcal{C}[123]) \\ &= \frac{-d(d-1)(d-2)t^3}{1+(d-1)(d-2)t^2+(d-2)t},\end{aligned}$$

and

$$\begin{aligned}f(t) &= \frac{1}{1-dt-\frac{-d(d-1)(d-2)t^3}{1+(d-1)(d-2)t^2+(d-2)t}} \\ &= \frac{(-d^2+3d-2)t^2+(-d+2)t-1}{(d-2)t^2+2t-1}.\end{aligned}$$

In general, if the set B is invariant under the action of the symmetric group, there exists a more efficient way to find the generating function (3.1).

Two words u, v are **equivalent**, $u \equiv v$ if there exists a permutation λ such that $\lambda(u) = v$. By symmetry, all the elements in the equivalence class of v have the same cluster generating function $\overline{weight}(\mathcal{C}[v])$.

Define the **dimension** of a letter v, $dim(v)$ as the number of different letters appearing in v. Then there are $\binom{d}{dim(v)}$ different words in the equivalence class of v.

Suppose the words set B is partitioned into different equivalence classes

$$B_1, B_2, B_3, \cdots, B_k,$$

and $b_1, b_2, b_3, \cdots, b_k$ are the representatives respectively. Define $(b_i : B_j) := \sum_{b \in B_j} (b_i : b)$. Then the system (3.4) can be reduced to:

$$\overline{weight}(\mathcal{C}[b_i]) = -\overline{weight}(b_i) - \sum_{j=1}^{k} (b_i : B_j)\overline{weight}(\mathcal{C}[b_j]),\ i = 1, \cdots, k. \quad (3.5)$$

This is a system of k linear equations with k unknowns

$$\overline{weight}(\mathcal{C}[b_i]), \quad i = 1, 2, \cdots, k.$$

Remember that k is the number of different equivalence classes in B. There are many fewer equations and many fewer unknowns than in the original Goulden-Jackson cluster method, and thus everything is much more efficient. After solving the system, we can obtain $\overline{weight}(\mathcal{C})$ by

$$\overline{weight}(\mathcal{C}) = \sum_{i=1}^{k} \binom{d}{dim(b_i)} \overline{weight}(\mathcal{C}[b_i]). \quad (3.6)$$

Given $u = u_1 u_2 u_3 \cdots u_n$, let

$$H_i(u) := u_1 u_2 \cdots u_i$$

and

$$T_i(u) := u_{n-i+1} u_{n-i+2} \cdots u_{n-1} u_n,$$

where $0 \leq i \leq n$. It is easy to obtain

$$(b_i : B_j) = \sum_{m=1}^{min(|b_i|,|b_j|)-1} I(T_m(b_i) \equiv H_m(b_j)) \binom{d}{dim(b_j) - dim(H_m(b_j))} t^{|b_j|-m},$$

where

$$I(T_m(b_i) \equiv H_m(b_j)) = \begin{cases} 1, & \text{if } T_m(b_i) \equiv H_m(b_j), \\ 0, & \text{otherwise.} \end{cases}$$

In the two examples below, the first can still be done with the unextended Goulden-Jackson, since the number of letters is numeric, 3, but the second one requires the new extension, since the number of letters is d, i.e. a *symbol*.

Example 1: Let $V = \{1, 2, 3\}$. Find the generating function for the number of words which have neither three consecutive different letters nor three consecutive same letters as factors, i.e.

$$B = \{123, 132, 213, 231, 312, 321, 111, 222, 333\}.$$

The set B is invariant under the symmetric group; and it can be partitioned into two equivalence classes:

$$B_1 = \{123, 132, 213, 231, 312, 321\},\ B_2 = \{111, 222, 333\}.$$

By the system (3.5) and the equation (3.6), we have

$$\begin{cases} \overline{weight}(\mathcal{C}[123]) &= -t^3 - 2t^2\,\overline{weight}(\mathcal{C}[123]) - t\,\overline{weight}(\mathcal{C}[123]) \\ & \quad -t^2\,\overline{weight}(\mathcal{C}[111]) \\ \overline{weight}(\mathcal{C}[111]) &= -t^3 - 2t^2\,\overline{weight}(\mathcal{C}[123]) - t^2\,\overline{weight}(\mathcal{C}[111]) \\ & \quad -t\,\overline{weight}(\mathcal{C}[111]) \end{cases},$$

and

$$\overline{weight}(\mathcal{C}) = 6\,\overline{weight}(\mathcal{C}[321]) + 3\,\overline{weight}(\mathcal{C}[111]).$$

Solving the system, finally we get

$$\begin{aligned} f(t) &= \frac{1}{1 - 3t - \overline{weight}(\mathcal{C})} \\ &= \frac{1}{1 - 3t - [6\,\overline{weight}(\mathcal{C}[321]) + 3\,\overline{weight}(\mathcal{C}[111])]} \\ &= -\frac{3t^2 + t + 1}{2t - 1}. \end{aligned}$$

Example 2: Let $V = \{1, 2, 3, \cdots, d\}$. Find the generating function for the number of words which have neither three consecutive different letters nor three consecutive same letters as factors.

By (3.5) and (3.6), we have

$$\begin{cases} \overline{weight}(\mathcal{C}[123]) &= -t^3-(d-1)(d-2)t^2\,\overline{weight}(\mathcal{C}[123]) \\ & \quad -(d-2)t\,\overline{weight}(\mathcal{C}[123])-t^2\,\overline{weight}(\mathcal{C}[111]) \\ \overline{weight}(\mathcal{C}[111]) &= -t^3-(d-1)(d-2)t^2\,\overline{weight}(\mathcal{C}[123]) \\ & \quad -t^2\,\overline{weight}(\mathcal{C}[111])-t\,\overline{weight}(\mathcal{C}[111]) \end{cases},$$

and

$$\overline{weight}(\mathcal{C})=d(d-1)(d-2)\,\overline{weight}(\mathcal{C}[321])+d\,\overline{weight}(\mathcal{C}[111]).$$

Finally,

$$\begin{aligned} f(t) &= \frac{1}{1-dt-\overline{weight}(\mathcal{C})} \\ &= \frac{(-d^2+2d)t^3+(-d^2+2d-1)t^2+(1-d)t-1}{(d-1)t^2+t-1}. \end{aligned}$$

3.4 Finite Memory Self-Avoiding Walks

The set of so-called *self-avoiding walks* ([16])can be viewed as a set of words over the alphabet

$$V=\{1,-1,2,-2,\cdots,d,-d\},$$

which avoid as factors the words with as many i's as $-i$'s for *each* i between 1 and d. In other words, it is a set of words that avoid the 'bad factors' in

$$B = \{[1,-1],[1,2,-1,-2],[1,2,3,-1,-2,-3],\cdots\}$$

and all their images under the action of the group of signed permutations. J. Noonan ([17]) has a detailed discussion about the finite memory self-avoiding walks for the memory up to 8. We have implemented the procedures for symmetric cases under signed permutations too. Using our Maple package, we can automatically get the formula of the generating functions for 2-step, 4-step and 6-step memory self-avoiding walks. For 8-step memory self-avoiding walks, the package set up a system of 112 linear equations but our own computer was not big enough to solve it.

3.5 The Maple Package

All the procedures are included in the package "SYMBOLIC_GJ", downloadable from the web address:

`http://www.math.temple.edu/~wen/gj/SYMBOLIC_GJ .`

The main procedures take the cardinality of the alphabet as symbolic input. Moreover the package can be used to compute generating functions for the symmetric cases and for the finite memory self-avoiding walks.

CHAPTER 4

THE NONCOMMUTATIVE GJ CLUSTER METHOD

4.1 Introduction

Let $V = \{a_1, a_2, \cdots, a_d\}$ be a finite alphabet with d letters. Let B be a finite set of $words$ (on V) and let $\mathcal{L}(B)$ denote the set of all words (language) over V that avoid the members of B as factors. Suppose $q(n)$ is the total number of words of length n that avoid the words in B as factors. Then using the Goulden-Jackson cluster method ([11],[12] [18]), we can efficiently find the generating function

$$f(t) = \sum_{n=0}^{\infty} q(n)\, t^n \,. \tag{4.1}$$

However much information is lost, since the single variable t accounts only for the *length* . In order to keep track of the *order*, we set up the general generating function

$$g(a_1, a_2, \cdots, a_d) = \sum_{n=0}^{\infty} \sum_{w_1 w_2 \cdots w_n \in \mathcal{L}(B)} w_1 \cdot w_2 \cdot w_3 \cdot \cdots \cdot w_n \,, \tag{4.2}$$

where the operation "$\cdot$" is noncommutative. The generating function (4.2) preserves not only the information of letters in a word but also the order of the letters. Actually it contains all the information of the language $\mathcal{L}(B)$. In this paper, we develop a *noncommutative Goulden-Jackson method* to find a rational form of the function (4.2).

4.2 The Noncommutative Goulden-Jackson Cluster Method

The noncommutative Goulden - Jackson cluster method is analogous to the normal Goulden-Jackson cluster method except that the definitions of the weight of a word are different. Amazingly the procedures developed in [18] and [24] can all be applied to the noncommutative case. Here we provide an analogous version to make it self-contained.

A **factor** of the word $w_1 w_2 \cdots w_n$ is one of the words $w_i w_{i+1} \cdots w_{j-1} w_j$, $1 \leq i \leq j \leq n$, that we shall denote by $[i, j]$. Two factors $[i, j]$ and $[i', j']$ **overlap** if they

have at least one common letter. The **weight** of a word $w = w_1\, w_2\, w_3 \cdots w_n$ is the noncommutative product $W(w) = w_1 \cdot w_2 \cdot \cdots \cdot w_n$. Summing weights of all the words in $\mathcal{L}(B)$,

$$g(a_1, a_2, \cdots, a_d) = \sum_{w \in \mathcal{L}(B)} W(w), \tag{4.3}$$

gives us the general generating function (4.2).

A word with some factors marked is called a **marked word**. A marked word can be written in the form:

$$(w; [i_1, j_1], [i_2, j_2], \cdots, [i_l, j_l]), \text{ where } [i_r, j_r], 1 \leq r \leq l, \text{ are marked factors.}$$

A **cluster** is a marked word

$$(w_1 w_2 \cdots w_n; [i_1(=1), j_1], [i_2, j_2], \cdots, [i_l, j_l(=n)]),$$

where $[i_k, j_k]$ overlaps with $[i_{k+1}, j_{k+1}]$ for all $k = 1, 2, \cdots, l-1$. The set of clusters $\mathcal{C}$ can be partitioned into

$$\mathcal{C} = \bigcup_{u \in B} \mathcal{C}[u],$$

where $\mathcal{C}[u]$, $u \in B$, is the set of clusters whose first entry is u.

Define the **weight**,$\overline{W}$, of a marked word $w = w_1 w_2 \cdots w_n$ with marked factors

S, $S \subset B$, as

$$\overline{W}(w, S) = (-1)^{|S|} w_1 \cdot w_2 \cdot w_3 \cdot \cdots \cdot w_n = (-1)^{|S|} W(w),$$

where $|S|$ is the cardinality of S.

Let

$$B(w) := B \bigcap \bigcup_{0 \le i \le j \le |w|} \{(w, |i, j|)\}.$$

Using the inclusion-exclusion principle, we have:

$$\begin{aligned}
g(a_1, a_2, \cdots, a_d) &= \textstyle\sum_{w \in \mathcal{L}(B)} W(w) \\
&= \textstyle\sum_{w \in V^*} W(w) 0^{|B(w)|} \qquad (\text{ Define } 0^0 = 1) \\
&= \textstyle\sum_{w \in V^*} W(w) [1 + (-1)]^{|B(w)|} \\
&= \textstyle\sum_{w \in V^*} W(w) \sum_{S \subset B(w)} (-1)^{|S|} \\
&= \textstyle\sum_{w \in V^*} \sum_{S \subset B(w)} (-1)^{|S|} W(w) \\
&= \textstyle\sum_{w \in V^*} \sum_{S \subset B(w)} \overline{W}(w, S).
\end{aligned}$$

Hence the generating function (4.2) is exactly the same as the generating function for the weighted marked word.

A nonempty marked word either starts with a letter that is not part of a cluster, or starts with a cluster. Peelingoff the maximal cluster, we get a shorter marked

word. Thus we have the following decomposition

$$\mathcal{M} = \{empty\_word\} \cup \mathcal{M}V \cup \mathcal{M}\mathcal{C}.$$

Taking weights on both sides and solving it for $\overline{W}(\mathcal{M})$, we have

$$g(a_1, a_2, \cdots, a_d) = \overline{W}(\mathcal{M}) = \frac{1}{1 - (a_1 + a_2 + \cdots + a_d) - \overline{W}(\mathcal{C})} \quad . \tag{4.4}$$

For a given word $w = w_1 w_2 \cdots w_n$, let $HEAD(w)$ be the set of all proper prefixes:

$$HEAD(w) := \{w_1 w_2 \cdots w_k | k = 1, 2, \cdots, n-1\},$$

and $TAIL(w)$ be the set of all proper suffixes

$$TAIL(w) := \{w_k w_{k+1} \cdots w_n | k = 2, 3, \cdots, n\},$$

and

$$OVERLAP(u, v) := TAIL(u) \cap HEAD(v).$$

Let u/v denote the operation of chopping the head v off the word u, and let

$$(u : v) := \sum_{x \in OVERLAP(u,v)} \overline{W}(v/x).$$

Given a cluster in $\mathcal{C}[u]$, $u \in B$, it either consists of just u or chopping its head u results in a shorter cluster in $\mathcal{C}[v]$, $v \in B$ if $OVERLAP(u, v)$ is not empty. On the other hand, given a cluster in $\mathcal{C}[v]$, we can always reconstitute the bigger cluster in $\mathcal{C}[u]$ by adding some words in $\bigcup_{x \in OVERLAP(u,v)} \{v/x\}$. Hence, there exists a bijection:

$$\mathcal{C}[u] \leftrightarrow \{(u; [1, |u|])\} \bigcup_{v \in B} \mathcal{C}[u] OVERLAP(u, v).$$

Taking weights on both sides, we have:

$$\overline{W}(\mathcal{C}[u]) = (-1)\, \overline{W}(u) - \sum_{v \in B} (u : v) \cdot \overline{W}(\mathcal{C}[v]). \tag{4.5}$$

This is a system of $|B|$ linear equations with $|B|$ unknowns

$$\overline{W}(\mathcal{C}[v]), \quad v \in B.$$

The classical method cannot be used to solve the system for the product "$\cdot$" in the equation is noncommutative. Fortunately all the unknowns in the equations are at the rightmost part of the operand "$\cdot$" . We can use multiplication on the left to extract out the unknowns and then use the Gaussian elimination method to solve the system. This procedure can be done automatically by computer.

After solving these equations we can obtain $\overline{W}(\mathcal{C})$ by:

$$\overline{W}(\mathcal{C}) = \sum_{v \in B} \overline{W}(\mathcal{C}[v]).$$

The symbolic method ([24]) can also be implemented because $\overline{W}(\mathcal{C})$ is independent of the cardinality of the alphabet.

Example 1: Let $V = \{a_1, a_2\}$, find the generating function for the language which does not have words in $B = \{a_1a_1a_1\}$ as factors. Using (4.5) we set up a system of one equation:

$$\overline{W}(\mathcal{C}[a_1a_1a_1]) = -a_1^3 - a_1 \cdot \overline{W}(\mathcal{C}[a_1a_1a_1]) - a_1^2 \cdot \overline{W}(\mathcal{C}[a_1a_1a_1]).$$

After solving it, we have

$$\overline{W}(\mathcal{C}[a_1a_1a_1]) = -\frac{1}{1 + a_1 + a_1^2} \cdot a_1^3,$$

and

$$g(a_1, a_2) = \frac{1}{1 - a_1 - a_2 + \frac{1}{1+a_1+a_1^2} \cdot a_1^3}.$$

This can be simplified further as

$$g(a_1, a_2) = \frac{1}{(1 + a_1 + a_1^2) \cdot (1 - a_1 - a_2) + a_1^3} \cdot (1 + a_1 + a_1^2)\,.$$

Example 2: Let $V = \{a_1, a_2\}$ and $B = \{a_1a_2, a_2a_2\}$. Find the generating function of the language which does not have words in B as factors. Using (4.5) we set up a linear system of two equations:

$$\begin{cases} \overline{W}(\mathcal{C}[a_1a_2]) &= -a_1 \cdot a_2 - a_1 \cdot \overline{W}(\mathcal{C}[a_2a_2]) \\ \overline{W}(\mathcal{C}[a_2a_2]) &= -a_2^2 - a_2 \cdot \overline{W}(\mathcal{C}[a_2a_2]) \end{cases}.$$

After solving it, we have

$$\begin{cases} \overline{W}(\mathcal{C}[a_2a_2]) &= -\frac{1}{1+a_2} \cdot a_2^2 \\ \overline{W}(\mathcal{C}[a_1a_2]) &= -a_1 \cdot a_2 + a_1 \cdot \frac{1}{1+a_2} \cdot a_2^2 \end{cases}.$$

And finally we get

$$g(a_1, a_2) = \frac{1}{1 - a_1 - a_2 + \frac{1}{1+a_2} \cdot a_2^2 + a_1 \cdot a_2 - a_1 \cdot \frac{1}{1+a_2} \cdot a_2^2}.$$

4.3 Avoiding Pairs

One of the most interesting problems is to find the generating function for the language which avoids some pairs of letters as factors. [9] studies the problems of avoiding reflexive and acyclic relation pairs. These kinds of problems can be solved as a special case, using the Goulden-Jackson method. But there also exist a direct way to set up equations and thus to calculate the generating function: Divide the

whole language into d sublanguages in which the words start with the same letter. The generating functions for each subgroup satisfy a system of linear equations which can be solved by the revised Gaussian elimination method. And hence we can get the generating function for the language by summing up the generating functions for the sublanguages.

Here we give an example to illustrate how it works.

Example 3: Given an alphabet $V = \{a_1, a_2\}$, and a set of pairs

$$B = \{a_1a_1, a_2a_2\}.$$

Find the generating function of the language that avoids words in B as factors.

Let $C[a_i]$ be the sublanguage in which each word starts with the letter a_i,$i = 1, 2$. Then

$$C[a_1] = \{[1]\} \bigcup \{[1] \cdot C[a_2]\}.$$

Taking weight on both sides we have

$$\overline{W}(\mathcal{C}[a_1]) = a_1 + a_1 \cdot \overline{W}(\mathcal{C}[a_2]).$$

In the same way we can get another equation:

$$\overline{W}(\mathcal{C}[a_2]) = a_2 + a_2 \cdot \overline{W}(\mathcal{C}[a_1]).$$

Thus we can see all the unknowns are at the rightmost part of the operand "$\cdot$". Using multiplication on the left and also the Gaussian elimination method, we can solve the system:

$$\begin{cases} \overline{W}(\mathcal{C}[a_1]) &= a_1 \cdot [1 - \frac{1}{x_1 - \frac{1}{a_2}} \cdot (1 + a_1)] \\ \overline{W}(\mathcal{C}[a_2]) &= -\frac{1}{a_1 - \frac{1}{a_2}} \cdot (1 + a_1) \end{cases}.$$

Finally the generating function can be calculated by

$$f(a_1, a_2) = 1 + \overline{W}(\mathcal{C}[a_1]) + \overline{W}(\mathcal{C}[a_2]).$$

4.4 The Maple Package

All the procedures are included in the package "NONCOMM_GJ", downloadable from the web address:

`http://www.math.temple.edu/~wen/gj/noncomm/ .`

Users can get detailed online help when opening the package with Maple. The direct way to set up and solve the linear system, to get the generating function for avoiding pairs, is also included in the package.

CHAPTER 5

APPLICATION OF THE GJ CLUSTER METHOD IN TWO DIMENSIONAL CASES

5.1 Introduction

This chapter was initially inspired by the game tic-tac-toe, which is a two player game with pencils and paper. Each player may, in turn, put symbols ‘O’ or ‘X’ (one player can only put one kind of them) in a 3×3 grid. The first player who get 3 symbols in a row (vertically, horizontally or diagonally) wins. Played professionally, the game is a draw. The question is how many end positions there are which are draws. We then can extend this question to the generalized case: Given a finite

alphabet (e.g. ‘O’ and ‘X’), using them to fill out a $m \times n$ grid, and avoiding some certain one dimensional patterns (e.g. ‘OOO’ and ‘XXX’) either horizontally, vertically or diagonally, how many possibilities for the filling are there? Fortunately, We can use the extended Goulden-Jackson to solve this kind of problem.

5.2 Two Dimensional Cases

With the powerful tool of the Goulden - Jackson Cluster method, and an implementation of the method ([18]), we can solve the problem by constructing the new *vector alphabet* and *vector bad words*. The $m \times n$ rectangle can be viewed as an array of n m-dimensional vectors and here we look at each m dimensional vector as a *vector letter*. Altogether there are d^m vector letters. Unfortunately, when the length of some bad patterns is less than or equal to m, some vector letters are not good because they may contain one or more bad patterns as a subword. So we need to subtract all the vector letters which contain bad patterns, which then leaves us the reduced vector alphabet. Next, we use these vector letters to construct vector words of length m, and find all the bad vector words, and finally apply the Goulden-Jackson method to find the generating function.

5.3 Finding the Vector Alphabet

Simply using the permutation, we can get all the vector letters. Each vector letter can be judged according to whether it contains a bad pattern. We then collect all the vector letters which do not contain bad patterns and construct the vector alphabet we need.

5.4 Finding the Bad Vector Words

Again, using the permutation, we can find all the vector words with certain lengths (equal to the length of the bad patterns). Next, we can judge them one by one to see whether the patterns appear horizontally or in the two diagonals. Finally, we collect all the vector words which have at least one bad pattern and those will constitute the bad words set.

5.5 Applying the Goulden-Jackson Method

The original method has already been described by Nononn and Zeilberger [18]. Thus, we use their package to complete the rest of the computation.

5.6 The Maple Package

The Maple package which computes the vector alphabet and the bad vector words is downloadable from:

`http://www.math.temple.edu/~wen/gj/twodim/ .`

Detailed on - line help is available.

CHAPTER 6

FINDING AND PROVING THE ADDITIVE PERIODS FOR THE SG FUNCTION OF WYTHOFF'S GAME

6.1 Introduction

Experimental mathematics is a subarea of mathematics which uses computation to find certain properties and rules for mathematical expressions. Generally it needs the following steps: Given a $Problem(n)$, parameterized by integer n, apply a standard (or new) numerical algorithm to get the $Answers(n)$ for $n = 1, 2, \cdots, 100$.

Then have the human, or much better still, the machine, guess the symbolic answer $Answer(n)$ for symbolic n. Finally, use the machine to prove the guess automatically. This chapter will discuss how to use the computer to guess the additive periods of the Sprague - Grundy function of Wythoff's game ([13],[10],[22],[29]) and get an automatic computer proof.

6.2 Wythoff's Game

Wythoff's Game (also called Wythoff's Nim) is an impartial 2-player game played with 2 piles of counters. Each player may, in turn, remove any number of counters from either pile, or remove the same number of counters from both piles. The player who removes the last counter wins.

Let's use a pair (m, n) to describe a position of the game, where m and n correspond to the numbers of counters in each pile. The legal moves can be written as:

$$(m,n) \rightarrow \begin{cases} (i,n) & 0 \leq i < m\,; \\ (m,i) & 0 \leq i < n\,; \\ (m-i,n-i) & 0 < i \leq \min(m,n)\,. \end{cases}$$

For each position, we can assign a Grundy value:

$$\mathcal{G}(X) = \mathrm{mex}\{\mathcal{G}(Y) : Y \text{ is reachable from } X\}$$

with the initial condition

$$\mathcal{G}(0,0) = 0.$$

Let $\mathrm{mex}(S)$ be the smallest non-negative integer which is not in S (a finite set of nonnegative integers). For example:

$$\mathrm{mex}(1,2,3,4) = 0\,,$$

$$\mathrm{mex}(0,1,2,3) = 4\,,$$

$$\mathrm{mex}(0,1,4,5) = 2 \quad \text{and}$$

$$\mathrm{mex}(2) = 0\ .$$

The Grundy value for the position (m,n) may be defined recursively by

$$\mathcal{G}(m,n) = \mathrm{mex}\left(\begin{array}{ll} \{\mathcal{G}(i,n), & 0 \le i < m\} \quad \bigcup \\ \{\mathcal{G}(m,i), & 0 \le i < n\} \quad \bigcup \\ \{\mathcal{G}(m-i,n-i) & 0 < i \le \text{ min } (m,n)\} \end{array}\right).$$

The first few values of the Sprague-Grundy function are listed in Table 6.1. The set of losing positions consists of the zero positions of the Sprague-Grundy function $\mathcal{G}(m,n)$.

Dress, et. al. ([8]) have proved the additive periodicity of rows of the Sprague-

Table 6.1: Values of Sprague - Grundy Function

m / n	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	0	4	5	3	7	8	6	10
2	2	0	1	5	3	4	8	6	7	11
3	3	4	5	6	2	0	1	9	10	12
4	4	5	3	2	7	6	9	0	1	8

Grundy function of a class of Nim-like games which includes Wythoff's game. Howard A. Landman [14] gave a Simple FSM-Based Proof of the additive periodicity of the Sprague-Grundy Function.

In this chapter, we provide a method using the computer to *find* and *verify* automatically the additive periods for the Sprague-Grundy function.

6.3 Computer Guessing

For integers i and j, we can use the computer to generate values

$$\mathcal{G}(m,n),\ 0 \le m \le i \,\text{and}\, 0 \le n \le j.$$

If one pile is fixed, let's say m is fixed, then each non - negative integer will eventually appear in the sequence $\mathcal{G}(m,n), n = 0,1,2,\cdots,\infty$. Thus each row is a permutation of the nonnegative integers.

Using the computer, we can generate a sequence of Grundy values, and check whether the sequence of some values show additive periodicity. That is, for $p =$

$1, 2, \cdots$, we can check whether

$$\mathcal{G}(m, n + k) = \mathcal{G}(m, n) + k$$

is true for

$$n = N_0, N_0 + 1, \cdots, \text{ where } N_0 = 1, 2, \cdots .$$

If it always gives the answer *true* for some p and N_0, we can conjecture that the additive period is p and the starting position is N_0.

Thus the computer can give the guessed additive period p and the start position N_0 after which it shows periodicity. Now it can also prove the conjecture automatically.

6.4 Computer Proof

We know that ([14])

$$n - 2m \leq \mathcal{G}(m, n) \leq n + m.$$

Let's define

$$\text{mexg}(S, k) := \text{mex}\{S \bigcup \{0, 1, 2, \cdots, k\}\}.$$

Then

$$\mathcal{G}(m,n) = \mathrm{mexg}(T, n+m-1),$$

where

$$\begin{aligned} T &= \{\mathcal{G}(i,n), \quad 0 < i \le m\} && (6.1)\\ &\bigcup \{\mathcal{G}(m,n-i), \quad 0 < i \le 3m\} && (6.2)\\ &\bigcup \{\mathcal{G}(m-i,n-i) \quad 0 < i \le m\}\,. && (6.3) \end{aligned}$$

Thus, we can use the computer to automatically check whether

$$\mathcal{G}(m,N+p) = \mathcal{G}(m,N) + p \text{ for } N = N_0, N_0+1, \cdots, N_0+p.$$

If all are true, then by the induction it's true for all integers N, $N \ge N_0$.

6.5 The Maple Package

The maple package is downloadable from the web address:

`http://www.math.temple.edu/~wen/wythoff .`

REFERENCES

[1] Bailey D.H. and Borwein J.M., “Sample Problems of Experimental Mathematics.” 22 Sept. 2003. http://crd.lbl.gov/ dhbailey/expmath/expmath-probs.pdf.

[2] Blass U. and Fraenkel A.S. , *“The Sprague-Grundy function of Wythoff’s game”*, Theoret. Comput. Sci. (Math Games), 311-333, **75**(1990).

[3] Borwein J. , and Bailey D., “Mathematics by Experiment: Plausible Reasoning in the 21st Century”. Natick, MA: A. K. Peters, 2003.

[4] Connell I.G., *“A generalization of Wythoff’s game”*, Canad, Math Bull, 181-190, **2**(1959).

[5] Conway J.H. and Guy R.K., “The book of numbers”, *Copernicus, Springer, New York.* (1996).

[6] Coxeter H.S.M., *“ The golden section, phyllotaxis and Wythoff’s game”*, Scripta Math., 135-143, **19**(1953).

[7] Dollhopf J.,Goulden I., C. Greene “Words avoiding a reflexive acyclic relation”, preprint (2004)

[8] Dress A., Flammenkamp A. and Pink N., *“Additive periodicity of the Sprague-Grundy function of certain Nim games”*, Adv. in Appl. Math. 249-270, **22**(1999).

[9] Dollhopf J., Goulden I. and Greene C., “Words avoiding a reflexive acyclic relation”, preprint, 2004

[10] Fraenkel A.S. and Borosh I., *“ A generalization of Wythoff’s game”*, J. Combin. Theory, (Ser. A),175-191, **15**(1973).

[11] Goulden I. and Jackson D.M., *An inversion theorem for cluster decompositions of sequences with distinguished subsequences*, J. London Math. Soc.(2)**20** (1979), 567-576.

[12] Goulden I. and Jackson D.M., *”Combinatorial Enumeration”*, John Wiley, 1983, New York.

[13] Grundy P.M., *“Mathematics and Games”*, Eureka, 9-11, **27**(1964), originally published: ibid., 6-8, **2**(1939).

[14] Landman H. “A simple FSM-based proof of the additive periodicity of the Sprague-Grundy function of Wythoff’s Game” *More Games of No Chance* v.42 (2002)

[15] Lindström B., “A partition of L(3,n) into saturated chains”, *European J. Comb*, 61-63,1(1980).

[16] Madras N.,and Slade G., “The Self avoiding Walk”, Birkhauser, Boston (1993).

[17] Noonan J., “ New Upper Bounds for The Connective Constants of Self-Avoiding Walks”, *J. Stat. Phys.*, 91(1998), 871-888.

[18] Noonan J. and Zeilberger D., “ The Goulden-Jackson Cluster Method: Extensions, Applications, and Implementations”, *J. Difference Eq. Appl.*, 5(1999), 355-377.

[19] O’Hara K.M. , “Unimodality of Gaussian coefficients: a constructive proof”, *J. Combin. Theory Ser. A* 29-52, 53 (1990).

[20] Petkovsek M., Wilf H. and Zeilberger D. “A=B” AK Peters, Ltd., of Wellesley, Mass

[21] Proctor R., “Solution of two difficult combinatorial problems with linear algebra”, *American Mathematics Monthly,* 721-734,89(1982).

[22] Sprague R., “*Uber mathematische Kampfspiele* ” Tohoku Math. J, 438-444, **41**(1935-36).

[23] Weisstein E. “Experimental Mathematics” From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/ExperimentalMathematics.html

[24] Wen X., "The symbolic Goulden-Jackson cluster method" *J. Difference Eq. Appl.*, vol. 11, No. 2(2005),173-179.

[25] Wen X., "Computer-generated symmetric chain decompositions for $L(4,n)$ and $L(3,n)$" *Adv. Appl. Math.*, v. 33 (2004) 409-412.

[26] West D.B., "A Symmetric chain decomposition of L(4,n)", *European J. Comb*, 379-383,1(1980).

[27] Wolfram S., "A New Kind of Science" Wolfram Media, Inc., 2002

[28] Biggs N., Lloyd E. and Wilson R., "Graph Theory 1736-1936" (Oxford, 1986).

[29] Wythoff W.A., *"A modification of the game of Nim"*, Nieuw Arch. Wisk, 199-202, **7**(1907).

[30] Zeilberger D., "Kathy O'Hara's Constructive proof of the Unimodality of the Gaussian Polynomials", *American Mathematical Monthly,* 590-602,96(1989).

[31] Zeilberger D., Computerized Deconstruction *Adv. Appl. Math.* v. 31 (2003), 532-543.

APPENDIX A

SELECTED MAPLE CODE

A.1 SCDs For Young's Lattices

```
Sum_of_one_Segment:=proc(n,arr1,arr2,n_lowerbound,n_upperbound,num_pars)
  local s,y,size,a,b,r_upper,z,c;
  size:=nops(arr1);
  a:=[n-arr1[size],seq((arr1[size-t]-arr1[size-t-1]),t=0..size-2),arr1[1]];
  b:=[n-arr2[size],seq((arr2[size-t]-arr2[size-t-1]),t=0..size-2),arr2[1]];
  for s from 1 to size+1 do
    if (a[s]-b[s]<>0)then r_upper:=a[s]-b[s];
    fi;
  od;
  for s from 1 to size+1 do
    if (a[s]-b[s]<>0)then z[s]:=r*(a[s]-b[s])/(r_upper);
    else z[s]:=0;
    fi;
  od;
  for s from 1 to size+1 do
    c[s]:=b[s]+z[s];
  od;

  if num_pars=4 then
    normal(sum(sum(sum(sum(sum(sum(expression(c,size+1),r=0..infinity),
    n=n_lowerbound..n_upperbound),j=0..infinity),i=0..infinity),
      k=0..infinity),l=0..infinity)
    -sum(sum(sum(sum(sum(expression(c,size+1),r=r_upper..infinity),
      n=n_lowerbound..n_upperbound),j=0..infinity),i=0..infinity),
      k=0..infinity),l=0..infinity));
  elif num_pars=3 then
    normal(sum(sum(sum(sum(sum(expression(c,size+1),r=0..infinity),
      n=n_lowerbound..n_upperbound),j=0..infinity),i=0..infinity),
        k=0..infinity)
          -sum(sum(sum(sum(sum(expression(c,size+1),r=r_upper..infinity),
            n=n_lowerbound..n_upperbound),j=0..infinity),
              i=0..infinity),k=0..infinity));
  elif num_pars=2 then
```

```
    normal(sum(sum(sum(sum(expression(c,size+1),r=0..infinity),
      n=n_lowerbound..n_upperbound),j=0..infinity),i=0..infinity)
        -sum(sum(sum(sum(expression(c,size+1),r=r_upper..infinity),
          n=n_lowerbound..n_upperbound),j=0..infinity),i=0..infinity));
  elif num_pars=1 then
    normal(sum(sum(sum(expression(c,size+1),r=0..infinity),
      n=n_lowerbound..n_upperbound),i=0..infinity)
        -sum(sum(sum(expression(c,size+1),r=r_upper..infinity),
          n=n_lowerbound..n_upperbound),i=0..infinity));
  elif num_pars=0 then
    normal(sum(sum(expression(c,size+1),r=0..infinity),
      n=n_lowerbound..n_upperbound)-sum(sum(expression(c,size+1),
        r=r_upper..infinity),n=n_lowerbound..n_upperbound));
  fi:
end:

expression:=proc(k,size)
  local t,s;
  t:=1;
  for s from 0 to size-1 do
    t:=t*x[s]^k[s+1];
  od;
  t;
end:

Sum_of_one_chain:=proc(n,arr,n_lower_bound,n_upper_bound,num)
  local s,t,y,z,result;
  result:=0;
  t:=arr[1];t[1]:=t[1]+1;
  result:=result
    +Sum_of_one_Segment(n,t,arr[1],n_lower_bound,n_upper_bound,num);
  for s from 1 to nops(arr)-1 do
    result:=result+Sum_of_one_Segment(n,arr[s],arr[s+1],
      n_lower_bound,n_upper_bound,num);
  od;
  normal(result);
end:

sumall_4n_1:=proc()
  local a,b:
  a:=Sum_of_one_chain(n,[
    [n-2*k-2*j-i-1,n-2*k-j-i-1,n-k-j-1,n-k],
    [n-2*k-2*j-i-1,n-2*k-j-i-1,n-k-i-j-1,n-k],
    [n-2*k-2*j-i-1,n-2*k-j-i-1,n-k-i-j-1,n-k-i-1],
    [k,          n-2*k-j-i-1,n-k-i-j-1,n-k-i-1],
    [k,          k+j,      n-k-i-j-1,n-k-i-1],
    [k,          k+j,      2*k+j+i+1,n-k-i-1],
    [k,k+j,2*k+i+j+1,2*k+2*j+i+2]],2*i+2*j+3*k+3,infinity,3):

  b:=Sum_of_one_chain(n,[
    [n-2*k-2*j-i,n-2*k-j-i,n-k-j,n-k],
    [k+i,n-2*k-j-i,n-k-j,n-k],
    [k+i,k+j+i,n-k-j,n-k],
    [k+i,k+j+i,2*k+j+i,n-k] ,
    [k+i,k+j+i,2*k+j+i,2*k+2*j+i],
    [k,k+j+i,2*k+j+i,2*k+2*j+i],
    [k,k+j,2*k+i+j,2*k+2*j+i]],2*i+2*j+3*k,infinity,3):
  normal(a+b);
end:

sumall_4n_2:=proc()
  local a,b:
  a:=Sum_of_one_chain(n,[
```

```
    [n-2*k-2*j-i,n-2*k-j-i,n-k-j,n-k],
    [n-2*k-2*j-i,n-2*k-j-i,n-k-i-j,n-k],
    [n-2*k-2*j-i,n-2*k-j-i,n-k-i-j,n-k-i],
    [k,          n-2*k-j-i,n-k-i-j,n-k-i],
    [k,          k+j,      n-k-i-j,n-k-i],
    [k,          k+j,      2*k+j+i,n-k-i],
    [k,k+j,2*k+i+j,2*k+2*j+i]],2*i+2*j+3*k,infinity,3):

  b:=Sum_of_one_chain(n,[
    [n-2*k-2*j-i-2,n-2*k-j-i-1,n-k-j,n-k],
    [k+i+1,n-2*k-j-i-1,n-k-j,n-k],
    [k+i+1,k+j+i+1,n-k-j,n-k],
    [k+i+1,k+j+i+1,2*k+j+i+1,n-k] ,
    [k+i+1,k+j+i+1,2*k+j+i+1,2*k+2*j+i+1],
    [k,k+j+i+1,2*k+j+i+1,2*k+2*j+i+1],
    [k,k+j+1,2*k+i+j+1,2*k+2*j+i+1]],2*i+2*j+3*k+3,infinity,3):
  normal(a+b);
end:

sumall_3n_1:=proc()
  local a,b:
  a:=Sum_of_one_chain(n,[
    [n-i-3*j,n-i-2*j,n-j],
    [i+j,n-i-2*j,n-j] ,
    [i+j,i+2*j,n-j],
    [i+j,i+2*j,i+3*j],
    [j,i+2*j,i+3*j]],2*i+4*j,infinity,2):
  b:=Sum_of_one_chain(n,[
    [n-i-3*j-1,n-i-2*j-1,n-j-1],
    [n-i-3*j-1,n-i-2*j-1,n-i-j-1] ,
    [j,n-i-2*j-1,n-i-j-1],
    [j,i+2*j+1,n-i-j-1],
    [j,i+2*j+1,i+3*j+2]], 2*i+4*j+3,infinity,2):
  normal(a+b);
end:

sumall_3n_2:=proc()
  local a,b:
  a:=Sum_of_one_chain(n,[
    [n-i-3*j-2,n-i-2*j-1,n-j],
    [i+j+1,n-i-2*j-1,n-j] ,
    [i+j+1,i+2*j+1,n-j],
    [i+j+1,i+2*j+1,i+3*j+1],
    [j+1,i+2*j+1,i+3*j+1]],2*i+4*j+3,infinity,2):
 b:=Sum_of_one_chain(n,[
    [n-i-3*j,n-i-2*j,n-j],
    [n-i-3*j,n-i-2*j,n-i-j] ,
    [j,n-i-2*j,n-i-j],
    [j,i+2*j,n-i-j],
    [j,i+2*j,i+3*j]], 2*i+4*j,infinity,2):
  normal(a+b);
end:

sumall_2n:=proc()
  local a:
  a:=Sum_of_one_chain(n,[
    [n-i,n-i],
    [i,n-i],
    [i,i]],2*i,infinity,1);
end:

sumall_1n:=proc()
  local a:
```

```
  a:=Sum_of_one_chain(n,[[n],
                     [0]],0,infinity,0);
end:

sumall_west :=proc()
  local a,b;
  a:=Sum_of_one_chain(n,[
    [n-3*i-j-2,n-2*i-j-1,n-i,n],
    [j+1,n-2*i-j-1,n-i,n] ,
    [j+1,i+j+1,n-i,n],
    [j+1,i+j+1,2*i+j+1,n],
    [0,i+j+1,2*i+j+1,n],
    [0,i+j+1,2*i+j+1,3*i+j+1],
    [0,i+1,2*i+j+1,3*i+j+1]],3*i+2*j+3,infinity,2):
  b:=Sum_of_one_chain(n,[
    [n-3*i-j,n-2*i-j,n-i,n],
    [n-3*i-j,n-2*i-j,n-i-j,n] ,
    [0,n-2*i-j,n-i-j,n],
    [0,n-2*i-j,n-i-j,n-j],
    [0,i,n-i-j,n-j],
    [0,i,2*i+j,n-j],
    [0,i,2*i+j,3*i+j]],3*i+2*j,infinity,2):
  normal((a+b)/(1-x[0]*x[4]));
end:

Sum_of_one_Segment_lind:=proc(n,arr1,arr2,init_n,init_i)
  local s,y,size,a,b,r_upper,z,c,expre;
  size:=nops(arr1);
  a:=[n-arr1[size],seq((arr1[size-t]-arr1[size-t-1]),
    t=0..size-2),arr1[1]];
  b:=[n-arr2[size],seq((arr2[size-t]-arr2[size-t-1]),
    t=0..size-2),arr2[1]];
  for s from 1 to size+1 do
    if (a[s]-b[s]<>0)then r_upper:=a[s]-b[s];
    fi;
  od;
  for s from 1 to size+1 do
    if (a[s]-b[s]<>0)then z[s]:=normal(r*(a[s]-b[s])/(r_upper));
    else z[s]:=0;
    fi;
  od;
  for s from 1 to size+1 do
    c[s]:=b[s]+z[s];
  od;
  expre:=subs(n=2*k+init_n,expression(c,size+1));
  r_upper:=subs(n=2*k+init_n,r_upper);
  if(init_i<>infinity) then
    normal(sum(sum(sum(expre,r=0..infinity),
      k=i+1-init_n..infinity),i=init_i..infinity)
        -sum(sum(sum(expre,r=r_upper..infinity),
          k=i+1-init_n..infinity),i=init_i..infinity));
  else
    normal(sum(sum(expre,r=0..infinity),k=0..infinity)
      -sum(sum(expre,r=r_upper..infinity),k=0..infinity));
  fi;
end:

Sum_of_one_chain_lind:=proc(n,arr,init_n,init_i)
  local s,t,y,z,result;
  result:=0;
  t:=arr[1];t[1]:=t[1]+1;
  result:=result+Sum_of_one_Segment_lind(n,t,arr[1],init_n,init_i);
  for s from 1 to nops(arr)-1 do
```

```
    result:=result
      +Sum_of_one_Segment_lind(n,arr[s],arr[s+1],init_n,init_i);
  od;
  normal(result);
end:

sumall_lind:=proc()
  local oc124,oc35,ec1245,ec36,ed124,ed35,ec;
  oc124:=Sum_of_one_chain_lind(n,[[n-2*i,n,n],[1,2*i+1,n],[0,2*i,n],
    [0,i+1,n-i+1],[0,i,n-i],[0,i,i]],1,0);
  oc35:=Sum_of_one_chain_lind(n,[[n-2*i-1,n,n],[0,2*i+1,n],
    [0,2*i,n-1],[0,i+1,n-i]],1,0);
  ec1245:=Sum_of_one_chain_lind(n,[[n-2*i-1,n,n],[1,2*i+2,n],
    [1,2*i+1,n-1],[0,2*i,n-1],[0,i,n-i-1],[0,i,n-i-2],[0,i,i]],0,0);
  ec36:=Sum_of_one_chain_lind(n,[[n-2*i,n,n],[1,2*i+1,n],
    [0,2*i+1,n-1],[0,i+1,n-i-1]],0,0);
  ed124:=Sum_of_one_chain_lind(n,[[n-2*i,n-1,n-1],[1,2*i,n-1],
    [1,2*i-1,n-2],[1,i,n-i-1],[1,i,n-i-2],[1,i,i]],0,1);
  ed35:=Sum_of_one_chain_lind(n,[[n-2*i+1,n-1,n-1],[2,2*i,n-1],
    [1,2*i,n-2],[1,i+1,n-i-1]],0,1);
  ec:=Sum_of_one_chain_lind(n,[[0,n,n],[0,0,n]],0,infinity);
  normal((oc124+oc35)/(1-x[0]*x[3])
    +(ec+ec1245+ec36+ed124+ed35)/(1-x[0]^2*x[3]^2));
end:
```

A.2 Symbolic Goulden-Jackson Method

```
# This part is for SSGJ
# findequ(n,u,seqv,s) get the equations for string u with strings seqv.
# usage: findequ(n,[1,2,3],[[1,2,3],[1,1]],s)

findequ:=proc(n,u,seqv,s)
  local i,j,h,t,x,y,m,term,equ,k,v,vset,yset:
  equ:=-s^(nops(u)):

  for h from 1 to nops(seqv) do
    v:=op(h,seqv):

    for i from 1 to nops(u) do
      x:=[op(i..nops(u),u)]:
      if nops(v)> nops(u)-i+1 then
        y:=[op(1..nops(u)-i+1,v)]:
        if isostring(x,y) then
          vset:=convert(v,set);
          yset:=convert(y,set);
          m:=nops(convert(v,set) minus yset);
          k:=nops(yset);
          term:=C(op(v)):
          for j from 1 to m do
            term:=term*(n-k-j+1):
          od:
          equ:=equ-term*s^(nops(u)-nops(y)):
        fi:
      fi:
    od:
  od:
  C(op(u))=equ:
end:
```

```
# kernalarr(Arr) clean Arr with only one copy of iso string left for
# each type. usage:kernalarr([[1,2,3],[3,2,1],[1,1]]);

kernalarr:=proc(Arr)
  local i,j,ans,isin:
  ans:=[];
  for i from 1 to nops(Arr) do
    isin:=0:
    for j from 1 to nops(ans) do
      if isostring(op(j,ans),op(i,Arr)) then
        isin:=1;
      fi:
    od:
    if isin=0 then
      ans:=[op(ans),op(i,Arr)]:
    fi:
  od:
  ans:
end:

# IsIsoSubString(arr1,arr2) test whether arr2 is a sub string of arr1.
# usage:IsIsoSubString([1,2,3,4],[4,3,2]);

IsIsoSubString:=proc(arr1,arr2)
  local i,j;
  if nops(arr2)>nops(arr1) then
    return false:
  fi:

  for i from 1 to nops(arr1)-nops(arr2)+1 do
    if isostring(arr2,[op(i..i+nops(arr2)-1,arr1)]) then
      return true:
    fi:
  od:
  return false:
end:

# GetOffIsoSupstring(arr) clean the string arr.
# usage:GetOffIsoSupstring([[1,2,2,3],[2,2]]);

GetOffIsoSupstring:=proc(arr)
  local i,j,k,isbigstring,ans;
  ans:=[]:
  for i from 1 to nops(arr) do
    isbigstring:=0:
    for j from 1 to nops(arr) do
      if i<>j then
        if IsIsoSubString(op(i,arr),op(j,arr)) then
          isbigstring:=1:
        fi:
      fi:
    od:
    if isbigstring=0 then
      ans:=[op(ans),op(i,arr)]:
    fi:
  od:
  ans:
end:

# SSGJ(n,mistake,s), symbolic symetric goulden Jackson, where n and s
# are symboles, mistake are arrays of mistakes,
# usage:SSGJ(n,[[1,2,3],[1,1]],s);
```

```
SSGJ:=proc(n,mistake,s)
  local i,j,k,eq,var,res,lu,v,zm,coef,ans,mistakes:
  mistakes:=kernalarr(mistake);
  mistakes:=GetOffIsoSupstring(mistakes):
  eq:={}:
  var:={}:
  for i from 1 to nops(mistakes) do
    eq:= eq union {findequ(n,op(i,mistakes),mistakes,s)}:
    var:= var union {C(op(op(i,mistakes)))}:
  od:

  var:=solve(eq,var):
  lu:=1-s*n:

  for i from 1 to nops(mistakes) do
    v:=op(i,mistakes):
    zm:=nops(convert(v,set)):
    coef:=1:
    for j from 1 to zm do
      coef:=coef*(n-j+1):
    od:
    lu:=lu-coef*subs(var,C(op(v))):
  od:
  ans:=normal(1/lu):
  collect(numer(ans),s)/collect(denom(ans),s) :
end:

# This part is for SGJ
# SGJ, symbolic goulden-jackson cluster method.
# usage:SGJ(n,{[1,2,3],[1,1]},s);

SGJ:=proc(n,MISTAKES1,s)
  local v,eq, var,i,lu,C,MISTAKES,ans:
  MISTAKES:=Hakten(MISTAKES1):
  eq:={}:
  var:={}:
  for i from 1 to nops(MISTAKES) do
    v:=op(i,MISTAKES):
    eq:= eq union {findeqz(v,MISTAKES,C,s)}:
    var:=var union {C[op(v)]}:
  od:
  var:=solve(eq,var):
  lu:=1-n*s:
  for i from 1 to nops(MISTAKES) do
    v:=op(i,MISTAKES):
    lu:=lu-subs(var,C[op(v)]):
  od:
  ans:=normal(1/lu):
  collect(numer(ans),s)/collect(denom(ans),s) :
end:

# findeqz sets up the equ C[v]= s+t*Sum_u overlap(u,v,x) *C[u]
# usage: findeqz([1,2,3],{[1,2,3],[1,1,1]},C,s);

findeqz:=proc(v,MISTAKES,C,s)
  local eq,i,u:
  eq:=-1:
  for i from 1 to nops(v) do
    eq:=eq*s:
  od:
  for i from 1 to nops(MISTAKES) do
    u:=op(i,MISTAKES):
    eq:=eq-overlapz(u,v,s)*C[op(u)]:
```

```
  od:
  C[op(v)]-eq=0:
end:

# overlapz is a procedure that given two words u and v, and a variable s
# computes the weight-enumerator of all v\suffix(u),
# for all suffixes of u that are prefixes of v, but with uniform weight s
# usage: overlapz([1,2,3,4],[3,4,5,6],s);

overlapz:=proc(u,v,s)
  local i,j,lu,gug:
  lu:=0:
  for i from 2 to nops(u) do
    for j from i to nops(u) while (j-i+1<=nops(v)
      and op(j,u)=op(j-i+1,v))do
    od:
    if j-i=nops(v) and u<>v then
      ERROR(v,‘is a subword of‘,u,‘illegal input‘):
    fi:
    if j=nops(u)+1 and (i>1 or j>2) then
      gug:=1:
      gug:=gug*s^(nops(v)-(j-i)):
      lu:=lu+gug:
    fi:
  od:
  lu:
end:

# Hakten1(B) removes all superflous words
# usage: hakten1({[1,2,3],[0,1,2,3,4]});

hakten1:=proc(B)
  local w,i:
  for i from 1 to nops(B) do
    w:=op(i,B):
    if superflous(B,w)=1 then
      RETURN(B minus {w}):
    fi:
  od:
  B:
end:

# issubword(u,v) returns 1 if v is a subword of u, otherwise 0
# usage: issubword([1,2,3,4],[2,3,4]);

issubword:=proc(u,v)
  local i,j:
  for i from 1 to nops(u) do
    for j from i to nops(u) while (j-i+1<=nops(v) and op(j,u)=op(j-i+1,v))
         do:
    od:
    if j-i=nops(v)  then
      RETURN(1):
    fi:
  od:
  0:
end:

# This part is for self avoiding walk.
# saw_isostring(u,v),judges whether u and v are isostrings,
# if u and v are iso return true, otherwise false.
# usage: saw_isostring([1,2,-1,-2],[2,1,-2,-1]);
```

```
saw_isostring:=proc(u,v)
  local i,j,t:
  if(nops(u)<>nops(v))then
    return false;
  fi:

  for i from 1 to nops(u) do
    t:={op(i,v)};
    for j from i+1 to nops(u)do
      if op(i,u)=op(j,u) then
        t:=t union {op(j,v)};
      fi:
      if op(i,u)=-op(j,u) then
        t:=t union {-op(j,v)};
      fi:
    od:
    if nops(t)>=2 then return false;
    fi:
  od:

  for i from 1 to nops(v) do
    t:={op(i,u)};
    for j from i+1 to nops(v)do
      if op(i,v)=op(j,v) then
        t:=t union {op(j,u)};
      fi:
      if op(i,v)=-op(j,v) then
        t:=t union {-op(j,u)};
      fi:
    od:
    if nops(t)>=2 then return false;
    fi:
  od:
  return true:
end:

# saw_findequ(n,u,seqv,s) get the equations for string u with strings
# seqv. usage: saw_findequ(n,[1,2,3],[[1,2,3],[1,1]],s)

saw_findequ:=proc(n,u,seqv,s)
  local i,j,h,t,x,y,m,term,equ,k,v,vset,yset:
  equ:=-s^(nops(u)):
  for h from 1 to nops(seqv) do
    v:=op(h,seqv):
    for i from 1 to nops(u) do
      x:=[op(i..nops(u),u)]:
      if nops(v)> nops(u)-i+1 then
        y:=[op(1..nops(u)-i+1,v)]:
        if saw_isostring(x,y) then
          m:=nops(saw_kernal(v))-nops(saw_kernal(y));
          k:=nops(saw_kernal(y));
          term:=C(op(v)):
          for j from 1 to m do
            term:=term*(n-k-j+1)*2:
          od:
          equ:=equ-term*s^(nops(u)-nops(y)):
        fi:
      fi:
    od:
  od:
  C(op(u))=equ:
end:
```

```
# saw_kernalarr(Arr) clean Arr with only one copy of iso string left for
# each type. usage:saw_kernalarr([[1,2,3],[3,2,1],[1,1]]);

saw_kernalarr:=proc(Arr)
  local i,j,ans,isin:
  ans:=[];
  for i from 1 to nops(Arr) do
    isin:=0:
    for j from 1 to nops(ans) do
      if saw_isostring(op(j,ans),op(i,Arr)) then
        isin:=1;
      fi:
    od:
  if isin=0 then
    ans:=[op(ans),op(i,Arr)]:
  fi:
  od:
  ans:
end:

#saw_kernal(Arr):

saw_kernal:=proc(Arr)
  local i,j,k,ans;
  ans:=[]:
  for i from 1 to nops(Arr) do
    k:=0;
    for j from 1 to nops(ans) do
      if op(i,Arr)=op(j,ans) or op(i,Arr)=-op(j,ans) then
        k:=1;
      fi:
    od:
    if k=0 then
      ans:= [op(ans),op(i,Arr)]:
    fi:
  od:
  ans:
end:

# saw_IsIsoSubString(arr1,arr2) test whether arr2 is a sub string of arr1.
# usage:saw_IsIsoSubString([1,2,3,4],[4,3,2]);

saw_IsIsoSubString:=proc(arr1,arr2)
  local i,j;
  if nops(arr2)>nops(arr1) then
    return false:
  fi:
  for i from 1 to nops(arr1)-nops(arr2)+1 do
    if saw_isostring(arr2,[op(i..i+nops(arr2)-1,arr1)]) then
      return true:
    fi:
  od:
  return false:
end:

#saw_GetOffIsoSupstring(arr) clean the string arr.
#usage:saw_GetOffIsoSupstring([[1,2,2,3],[2,2]]);

saw_GetOffIsoSupstring:=proc(arr)
  local i,j,k,isbigstring,ans;
  ans:=[]:
  for i from 1 to nops(arr) do
    isbigstring:=0:
```

```
    for j from 1 to nops(arr) do
      if i<>j then
        if saw_IsIsoSubString(op(i,arr),op(j,arr)) then
          isbigstring:=1:
        fi:
      fi:
    od:
    if isbigstring=0 then
     ans:=[op(ans),op(i,arr)]:
    fi:
  od:
  ans:
end:

# saw_SSGJ(n,mistake,s), symbolic symetric goulden Jackson,
# where n and s are symboles
# mistake are arrays of mistakes,
# usage:saw_SSGJ(n,[[1,-1],[1,2,-1,-2]],s);

SAW_SSGJ:=proc(n,mistake,s)
  local i,j,k,eq,var,res,lu,v,zm,coef,ans,mistakes:
  mistakes:=saw_kernalarr(mistake);
  mistakes:=saw_GetOffIsoSupstring(mistakes):
  eq:={}:
  var:={}:
  for i from 1 to nops(mistakes) do
    eq:= eq union {saw_findequ(n,op(i,mistakes),mistakes,s)}:
    var:= var union {C(op(op(i,mistakes)))}:
  od:
  var:=solve(eq,var):
  lu:=1-2*s*n:
  for i from 1 to nops(mistakes) do
    v:=op(i,mistakes):
    zm:=nops(saw_kernal(v)):
    coef:=1:
    for j from 1 to zm do
      coef:=coef*(n-j+1)*2:
    od:
    lu:=lu-coef*subs(var,C(op(v))):
  od:
  ans:=normal(1/lu):
  collect(numer(ans),s)/collect(denom(ans),s) :
end:
```

A.3 Noncommutative Goulden-Jackson Method

```
NonCommGJ:=proc(n,MISTAKES,s)
  local equ,var,lu,i,v:
  equ:=GetEquGJ(MISTAKES,C,s):
  var:=GetVarGJ(MISTAKES,C):
  var:=MySolve(equ,var):
  lu:=1:
  for i from 1 to n do
    lu:=lu-s[[i]]:
  od:

  for i from 1 to nops(MISTAKES) do
    v:=op(i,MISTAKES):
    lu:=lu - subs({op(var)},C[v]):
```

```
  od:
  (1/lu):
end:

GetEquGJ:=proc(B,C,s)
  local u,v,p,tempequ,ans,i,j,k:
  ans:=[]:
  for i from 1 to nops(B) do
    u:=op(i,B):
    p:=1:
    for j from 1 to nops(u) do
      p:=p . s[[op(j,u)]]:
    od:
    tempequ:=C[u]+p:
    for j from 1 to nops(B) do
      v:=op(j,B):
      tempequ:=tempequ + NonCommOverlap(u,v,s) . C[v]:
    od:
    ans:=[op(ans),tempequ]:
  od:
  ans:
end:

GetVarGJ:=proc(B,C)
  local i,ans:
  ans:=[]:
  for i from 1 to nops(B) do
    ans:=[op(ans),C[op(i,B)]]:
  od:
  ans:
end:

NonCommOverlap:=proc(u,v,s)
  local i,j,lu,gug,k:
  lu:=0:
  for i from 2 to nops(u) do
    for j from i to nops(u) while (j-i+1<=nops(v)
      and op(j,u)=op(j-i+1,v))do
      :
    od:
    if j-i=nops(v) and u<>v then
      ERROR(v,‘is a subword of‘,u,‘illegal input‘):
    fi:

    if j=nops(u)+1 and (i>1 or j>2) then
      gug:=1:
      for k from 1 to i-1 do
        gug:=gug . s[[op(k,u)]]:
      od:
      lu:=lu+gug:
    fi:
  od:
  lu:
end:

GetEquAP:=proc(s,pairs,A,x)
  local i,j,result,equ:
  result:=[];
  for i from 1 to nops(s) do
    equ:=x[[s[i]]]:
    for j from 1 to nops(s) do
      equ:=equ+x[[s[i]]] . A[[s[j]]]:
    od:
```

```
    for j from 1 to nops(pairs) do
      if op(1,op(j,pairs))= s[i] then
        equ:=equ-x[[s[i]]] . A[[op(2,op(j,pairs))]]:
      fi:
    od:
    result:= [op(result), equ-A[[s[i]]]]:
  od:
  result:
end:

GetVarAP:=proc(s,A)
  local i,j,result:
  result:=[]:
  for i from 1 to nops (s) do
    result:=[op(result),A[[s[i]]]]
  od:
  result:
end:

HaveElement:=proc(expression,elem)local i:
  if nops(expression) = 1 then
    if expression = elem then return(true):
      else return(false):
    fi:
  fi:
  for i from 1 to nops(expression) do
    if HaveElement(op(i,expression),elem) then return(true): fi:
  od:
  return(false):
end:

MyExpand:=proc(expr)
  local i,j,ans,leftpart,rightpart,coeffleft,coeffright,temp;
  if op(0,expr) = '+' then
    ans:=0:
    for i from 1 to nops(expr) do
      ans:=ans+MyExpand(op(i,expr)):
    od:
    return(ans):
  fi:
  if op(0,expr) = '.' then
    for i from 1 to nops(expr) do
      if op(0,op(i,expr))='+' then
        leftpart:=1:
        for j from 1 to i-1 do
          leftpart:= leftpart . op(j,expr):
        od:
        rightpart:=1:
        for j from nops(expr) to i+1 by -1 do
          rightpart:= op(j,expr) . rightpart:
        od:
        ans:=0;
        for j from 1 to nops(op(i,expr)) do
          ans := ans +MyCoeff(leftpart) *MyCoeff( op(j,op(i,expr)))
            * MyCoeff (rightpart)*MyExpand(MyNoCoeff(leftpart)
              . MyNoCoeff( op(j,op(i,expr))) .MyNoCoeff (rightpart)) :
        od:
        return(ans):
      fi:
    od:
    ans:=1:
    for i from 1 to nops(expr) do
      ans:=ans . MyExpand(op(i,expr)):
```

```
    od:
    return (ans):
  fi:

  if op(0,expr) = '*' then
    if type(op(1,expr),complex) then
      rightpart:=1:
      for i from 2 to nops(expr) do
        rightpart:=rightpart * op(i,expr):
      od:
      return (op(1,expr) * MyExpand(rightpart)):
    fi:
  fi:


  if op(0,expr) = '^' then
    if nops(op(1,expr))=1 then
      return (expr):
    fi:
    if op(2,expr)=-1 then
      temp:=op(1,expr):
      ans:=1:
      if op(0,temp)='.' then
        for i from nops(temp) to 1 by -1 do
          ans:=ans . (1/MyExpand(op(i,temp))):
        od:
        return(ans):
      fi:
    fi:
    if op(2,expr)< 0 then
      return (expr):
    fi:
    if type(op(2,expr),even) then
      return (MyExpand(MySquare(op(1,expr))^(op(2,expr)/2))):
    fi:
    if type(op(2,expr),odd) then
      return (MyExpand((MySquare(op(1,expr))^((op(2,expr)-1)/2)).
        op(1,expr))):
    fi:
  fi:
  return(expr);
end:

AvoidPairs:=proc(s,p,A,x)
  local i, equ,var,ans,res,tempvar,tempequ:
  equ:=GetEquAP(s,p,A,x):
  var:=GetVarAP(s,A):
  ans:=MySolve(equ,var);
  res:=[]:

  for i from 1 to nops(ans) do
    tempvar:=op(1,op(i,ans)):
    tempequ:=op(2,op(i,ans)):
    res:=[op(res), tempvar
      =MyExplicitForm(MyCombineCoeff(MyCombine(tempequ)))]:
    #MyPositiveExponentForm##
  od:
  res:
end:

MyCombineOneVar:=proc(equ,var)
  local i,j,k,HaveVar, NoVar;
  equ:=MyExpand(equ):
```

```
  HaveVar:=0:
  NoVar:=0:
  for i from 1 to nops(equ) do
    if HaveElement(op(i, equ),var) then
      HaveVar:=HaveVar+op(i,equ):
    else NoVar:=NoVar+op(i,equ):
    fi:
  od:
end:

MySquare:=proc(expr)
  local i,j,ans:
  ans:=0;
  if op(0,expr) =‘+‘ then
    for i from 1 to nops(expr) do
      for j from 1 to nops(expr) do
        ans:=ans+ op(i, expr) . op(j, expr):
      od:
    od:
    return(ans):
  fi:
  expr^2:
end:

MySolve:=proc(equ, var)
  local havevar,novar,i,j,tempequ,tempvar,havevarequ,novarequ,ans:
  if nops(equ) <> nops(var) then
    return(‘error! number of eqution is not equal to number of
      variables‘):
  fi:

  if nops(var) > 1 then
    tempvar:=op(1,var):
  else
    tempvar:=op(var):
  fi:

  if nops(var) = 1 then
    tempequ:=MyExpand(op(1,equ)):
    if op(0,tempequ)=‘*‘ then return([op(var)=0]):
    fi:
    if op(0,tempequ)=‘.‘ then return([op(var)=0]):
    fi:
    havevar:=0:
    novar:=0:

    for i from 1 to nops(tempequ) do
      if HaveElement(op(i,tempequ),tempvar) then
        havevar:=havevar+MyWithoutLastTerm(op(i,tempequ)):
      else novar:=novar+op(i,tempequ):
      fi:
    od:
    return([tempvar=-(1/havevar) . novar]):
  fi:

  havevarequ:=[]:
  novarequ:=[]:

  for i from 1 to nops(equ) do
    tempequ:=MyExpand(op(i,equ)):
    havevar:=0:
    novar:=0:
    for j from 1 to nops(tempequ) do
```

```
      if HaveElement(op(j,tempequ),tempvar) then
        havevar:=havevar + MyWithoutLastTerm(op(j,tempequ)):
      else novar:=novar+op(j,tempequ):
      fi:
    od:
    if havevar=0 then novarequ:=[op(novarequ),tempequ]:
      else havevarequ:=[op(havevarequ),MyExpand((1/havevar) . novar)]:
    fi:
  od:

  for i from 2 to nops(havevarequ) do
    novarequ:=[op(novarequ), op(i,havevarequ)-op(1,havevarequ)]:
  od:
  tempequ:=op(1,havevarequ);
  ans:=MySolve(novarequ,[op(2..nops(var),var)]);
  ans:=[op(ans),subs(ans,tempvar=-tempequ)]:
  ans:
end:


MyCoeff:=proc(expr)
  if op(0,expr)  <> '*' then return(1):
  fi:
  if  type(op(1,expr),complex) then
    return (op(1,expr)):
  fi:
  return(1):
end:

MyNoCoeff:=proc(expr)
  local ans,i:
  if op(0,expr)  <> '*' then return(expr):
  fi:
  if  type(op(1,expr),complex) then
    ans:=1:
    for i from 2 to nops(expr) do
      ans:=ans * op(i,expr):
    od:
    return(ans):
  fi:
  return(expr):
end:

MyCombine:=proc(expr)
  local i,j,tempexpr,ans,withf,withoutf :
  if op(0,expr) = '+' then
    withf:=0:
    withoutf:=0;
    tempexpr:=MyFirstTermInProduct(op(1,expr)):
    for i from 1 to nops(expr) do
      if MyFirstTermInProduct(op(i,expr))=tempexpr then
        withf:= withf + MyNoFirstTermInProduct(op(i,expr)):
      else
        withoutf:=withoutf+op(i,expr):
      fi:
    od:
    return( tempexpr . MyCombine(withf) + MyCombine(withoutf)):
  fi:

  if op(0, expr)='*' then
    ans:=1:
    for i from 1 to nops(expr) do
      ans:= ans * MyCombine(op(i,expr)):
```

```
    od:
    return(ans):
  fi:
  if op(0, expr)='.' then
    ans:=1:
    for i from 1 to nops(expr) do
      ans:= ans . MyCombine(op(i,expr)):
    od:
    return(ans):
  fi:
  if op(0, expr)='^' then
    return(MyCombine(op(1,expr))^op(2,expr)):
  fi:
  return(expr):
end:

MyCombineCoeff:=proc(expr)
  local i,j,tempexpr,ans,withf,withoutf :
  if op(0,expr) = '+' then
    tempexpr:=1:
    for i from 1 to nops(expr) do
      if nops(op(i,expr))=1 and type(op(i,expr),complex) and
        op(i,expr)<0
        then tempexpr:=-1:
      fi:
    od:
    withf:=0:
    for i from 1 to nops(expr) do
      withf:= withf + MyCombineCoeff(tempexpr* (op(i,expr))):
    od:
    return( tempexpr * withf ):
  fi:
  if op(0, expr)='*' then
    ans:=1:
    for i from 1 to nops(expr) do
      ans:= ans * MyCombineCoeff(op(i,expr)):
    od:
    return(ans):
  fi:
  if op(0, expr)='.' then
    ans:=1:
    for i from 1 to nops(expr) do
      tempexpr:=1:
      if  op(0,(op(i,expr))) = '+' then
        for j from 1 to nops(op(i,expr)) do
          if type(op(j,op(i,expr)),complex) and op(j,op(i,expr))<0
    then tempexpr:=-1:
          fi:
        od:
      fi:
      ans:= tempexpr*ans . MyCombineCoeff(tempexpr*op(i,expr)):
    od:
    return(ans):
  fi:

  if op(0, expr)='^' then
    return(MyCombineCoeff(op(1,expr))^op(2,expr)):
  fi:
  return(expr):
end:


# MyExplicitForm returns the explicit form of expr
```

```
MyExplicitForm:=proc(expr)
  local i,j,ans,tempbase,goodform,locateplace,temp,badform,
    leftpart,rightpart,left,right:
  if nops(expr) =1 then return(expr): fi:

  if op(0,expr)='+' then
    ans:=0;
    for i from 1 to nops(expr) do
      ans:=ans + MyExplicitForm(op(i,expr)):
    od:
    return(ans):
  fi:

  if op(0,expr)='*' then
    ans:=1;
    for i from 1 to nops(expr) do
      ans:=ans * MyExplicitForm(op(i,expr)):
    od:
    return(ans):
  fi:

  if op(0,expr)='.' then
    ans:=1;
    for i from 1 to nops(expr) do
      ans:=ans . MyExplicitForm(op(i,expr)):
    od:
    return(ans):
  fi:

  if op(0,expr)='^' then
    if op(2,expr) <> -1 then
      return(MyExplicitForm(op(1,expr))^op(2,expr)):
    fi:
    tempbase:=op(1,expr):
    goodform:=MyIsGoodExpr(tempbase):

    if goodform then
      return(MyExplicitForm(op(1,expr))^op(2,expr)):
    fi:

    badform:=true:
    leftpart:=1:
    rightpart:=1:
    temp:=tempbase:
    while badform do
      badform:=false:
      left:=1:
      right:=1:
      left:=MyLeftReciprocalTerm(temp):

      if left <> 1 then rightpart:= MyCoeff(left)
          * MyCoeff(rightpart) * MyNoCoeff(left) . MyNoCoeff(rightpart):

        if op(0,temp)='+' then
          ans:=0;
          for i from 1 to nops(temp) do:
             ans:= ans +((MyCoeff(left) * MyCoeff (op(i,temp)) *
             MyExplicitForm(MyExpand( MyNoCoeff(left)
       .MyNoCoeff(op(i,temp))))))):
          od:
          temp:=ans:
          badform:= true:
          next:
```

```
        fi:
        temp:=(MyCoeff(left) * MyCoeff(temp)
  * MyExplicitForm(MyExpand(MyNoCoeff(left)
    . MyNoCoeff(temp)))):
        badform:=true:
        next:
      fi:

      #right:=MyRightReciprocalTerm(temp):
      if right <> 1 then leftpart:= leftpart . right:
        temp:=MyExpand(temp . right):
        badform:= true:
        next:
      fi:
    od:
    return (leftpart . (MyExplicitForm( temp))^(-1) . rightpart):
  fi:
  expr:
end:
```

A.4 Two Dimensional Goulden-Jackson Method

```
# AllPermute, gives all possible letters with length n with
# letters 1..d:
AllPermute:=proc(d,n)
  local i,A,j,answer:
  answer:=[]:
  if n<=0 then return ([[]]):fi:
  A:=AllPermute(d,n-1):
  for i from 1 to d do
    for j from 1 to nops(A) do
      answer:= [op(answer), [op(op(j,A)),i]]:
    od:
  od:
  answer:
end:
# CollectMistakes_0,compute vertical line mistakes
# Example: CollectMistakes_0(2,[2,2],2);
CollectMistakes_0:=proc(m,pattern,d)
  local i,j,k,l,answer,A,B;
  answer:={}:
  l:=nops(pattern):
  A:=AllPermute(d,m-l):
  if l>m then return({}): fi:
  if l=m then return({pattern}): fi:
  for i from 0 to m-l do
    k:=i+1;
    for j from 1 to nops(A) do
      B:=op(j,A):
      answer:=answer union {[op(1..i,B),op(pattern),
        op(k..nops(B),B)]}: #,
    od:
  od:
  answer:
end:
# CollectMistakes_1: compute horizental line mistakes
# Example: CollectMistakes_1(2,[2,2],2);
CollectMistakes_1:=proc(m,pattern,d)
  local i,j,k,l,answer,A,B,C,t,begin1,end1,begin2,end2;
```

```
  answer:={}:
  l:=nops(pattern):
  A:=AllPermute(d,(m-1)*(l)):
  for i from 0 to m-1 do
    for j from 1 to nops(A) do
      B:=[]:
      C:=op(j,A);
      for t from 1 to l do
        begin1:=1+(t-1)*(m-1);
        end1:=i+(t-1)*(m-1);
        begin2:=end1+1;
        end2:=(t)*(m-1);
        B:=[op(B),[op(begin1..end1,C),op(t,pattern),
  op(begin2..end2,C)]]
      od:
      answer:=answer union {B}:
    od:
  od:
  answer:
end:
# CollectMistakes_2: compute main diagonal mistakes
# Example: CollectMistakes_2(2,[2,2],2);

CollectMistakes_2:=proc(m,pattern,d)
  local i,j,k,l,answer,A,B,C,t,begin1,end1,begin2,end2;
  answer:={}:
  l:=nops(pattern):
  A:=AllPermute(d,(m-1)*(l)):
  for i from 0 to m-l do
    for j from 1 to nops(A) do
      B:=[]:
      C:=op(j,A);
      for t from 1 to l do
        begin1:=1+(t-1)*(m-1);
        end1:=i+(t-1)*(m-1)+t-1;
        begin2:=end1+1;
        end2:=(t)*(m-1);
        B:=[op(B),[op(begin1..end1,C),op(t,pattern),
  op(begin2..end2,C)]]
      od:
      answer:=answer union {B}:
    od:
  od:
  answer:
end:

# CollectMistakes_3: compute second diagonal mistakes
# Example: CollectMistakes_3(2,[2,2],2);
CollectMistakes_3:=proc(m,pattern,d)
  local i,j,k,l,answer,A,B,C,t,begin1,end1,begin2,end2;
  answer:={}:
  l:=nops(pattern):
  A:=AllPermute(d,(m-1)*(l)):
  for i from 0 to m-l do
    for j from 1 to nops(A) do
      B:=[]:
      C:=op(j,A);
      for t from 1 to l d
        begin1:=1+(t-1)*(m-1);
        end1:=i+(t-1)*(m-1)+l-(t);
        begin2:=end1+1;
        end2:=(t)*(m-1);
        B:=[op(B),[op(begin1..end1,C),op(t,pattern),
```

```
  op(begin2..end2,C)]]
      od:
      answer:=answer union {B}:
    od:
  od:
  answer:
end:

# IsFactor, judge whether a small letter is a factor of big letter
IsFactor:=proc(small,big)
  local i,j,k,s:
  s:=op(small):
  for i from 1 to nops(big) do
    if s=op(i,big) then return(true): fi:
  od:
  false:
end:

#collect all the mistakes
AllMistakes:=proc(m,pattern,d)
  local i,j,k,M0,M1,M2,M3,All,isfactor;
  M1:=CollectMistakes_1(m,pattern,d):
  M2:=CollectMistakes_2(m,pattern,d):
  M3:=CollectMistakes_3(m,pattern,d):
  All:={}:
  for i from 1 to nops(M1) do
    isfactor:=false:
    for j from 1 to nops(M0) do
      if IsFactor(op(j,M0),op(i,M1)) then isfactor:=true: break:fi:
    od:
    if not isfactor then All:=All union {op(i,M1)}: fi:
  od:
  return (All):
end:

#gives d^m - 1-dimensional mistakes that is the legal vector letters.
AllAlphaBeta:=proc(m,pattern,d)
  local A:
  A:={op(AllPermute(d,m))}:
  A:=A minus CollectMistakes_0(m,pattern,d):
  A:
end:
```

A.5 The Sprague-Grundy Function

```
# mex, compute the mex of a set
mex:=proc(s)
  local i,j,k;
  for i from 0 to max(op(s)) do
    if member(i,{op(s)}) then:else
      return i;
    fi:
  od:
  return max(op(s))+1;
end:

# f(a,b): compute the Sprague-Grundy function
f:=proc(a,b) option remember:
  local i,j,k,children;
  if a=0 then return b;fi:
  if b=0 then return a;fi:
```

```
  children:=[];
  for i from 0 to a-1 do
    children:=[op(children),f(i,b)];
  od:
  for i from 0 to b-1 do
    children:=[op(children),f(a,i)];
  od:
  for i from 1 to min(a,b) do
    children:=[op(children),f(a-i,b-i)];
  od:
  return mex(children):
end:

# optmex,optimized mex function
optmex:=proc(s,m)
  local i,j,k;
  if nops(s)=0 then return m; fi:
  for i from 0 to max(op(s))-m do
    if member(i+m,{op(s)}) then:else
      return i+m;
    fi:
  od:
  return max(op(s))+1;
end:

# optf, optimized Sprague-Grundy function
optf:=proc(a,b) option remember:
  local i,j,k,children,begin,begin2;
  if a=0 then return b;fi:
  if b=0 then return a;fi:
  children:=[];
  begin:=0;
  if a-2*b-1 >0 then begin:=a-2*b-1;fi:
  begin2:=0;
  if a-3*b-1 >0 then begin2:=a-3*b-1;fi:
  for i from begin2 to a-1 do
    children:=[op(children),optf(i,b)];
  od:
  for i from 0 to b-1 do
    children:=[op(children),optf(a,i)];
  od:
  for i from 1 to min(a,b) do
    children:=[op(children),optf(a-i,b-i)];
  od:
  return optmex(children,begin):
end:

#expandchain, get the arry of the chain ch.
expandchain:=proc(ch,n)
  local i,j,k,chain,p;
  p:=nops(ch);
  chain:=[]:
  for i from 0 to n-1 do
    chain:=[op(chain),ch[i-p*floor(i/p)+1]+p*floor(i/p)    ]:
  od:
  chain:
end:

#guess the guessing period part
guess:=proc(chain,symbol)
  local i,j,k,num,s,half,form,t;
  num:=nops(chain);
  half:=floor(num/2):
```

```
  for i from half+1 to num do
    k:=i-half;
    if(chain[half]+k=chain[i]) then
      s:=0;
      for j from half to num-k do
        if chain[j]+k <> chain[j+k] then
          s:=1;
        fi:
        if s=1 then break; fi:
      od:
      if s=0 then
        t:=floor(half/k)*k+1:
        form:=[]:
        for j from t to t+k-1 do
          form:=[op(form),symbol+chain[j]-t]:
        od:
      fi:
      if s=0 then
        for j from half to 1 by -1 do
          if chain[j]+k <>chain[j+k] then
    return([k,j+1,form]);
          fi:
        od:
        return([k,1,form]);
      fi:
    fi:
  od:
  return([-1]):
end:

symbolchainf:=proc(a,n,chain,symbol) option remember:
  local i,j,k,ch;
  ch:=guess(chain,symbol)[3]:
  expandchain(ch,n);
end:

symbolf:=proc(a,n,symbol) option remember:
  local i,j,k,ch;
  ch:=[seq(optf(i,a),i=0..1000)];
  symbolchainf(a,n,ch,symbol):
end:

symbolelemf:=proc(a,b,symbol) option remember:
  local i,j,k,ch;
  ch:=[seq(optf(i,a),i=0..1000)];
  symbolchainf(a,b,ch,symbol)[b]:
end:

# proof, the proving part
proof:=proc(b)
  local i,j,k,gue,p,prf,m,ch,guessedchain;
  #p:=nops(chain);
  ch:=guess([seq(optf(i,b),i=0..1000)],m):
  p:=ch[1]:
  guessedchain:=expandchain(ch[3],2*p+3*b);
  prf:=true:
  for i from 1 to p do
    if symbolmex(b,3*b+i,m)+p<>symbolmex(b,3*b+i+p,m)
      then prf:=false;
    fi:
  od:
  ch,prf:
end:
```

```
symbolmex:=proc(b,a,sym)
  option remember: local i,j,k,elem;
  elem:=[];
  for i from 1 to a-1 do
    elem:=[op(elem),symbolelemf(b,i,sym)]:
  od:
  for i from 0 to b-1 do
    elem:=[op(elem),symbolelemf(i,a,sym)]:
  od:
  for i from 1 to min(a-1,b) do
    elem:=[op(elem),symbolelemf(b-i,a-i,sym)]:
  od:
  optmex(elem,sym+a-1-2*b-1):
end:
```