Multiple Side Linear Equations

A New Tool For Solving Sparse Algebraic
Equations in Finite Field

by
Rigzin Wangdwei

Master Thesis

University of Bergen
Department of Informatics
February 3, 2010

Acknowledgements

The first person I would like to express my deep-felt gratitude to is my supervisor Professor Igor Semaev. Thanks for your excepting me as your student, for your encouragement, guidance and support from the initial to the final level which enabled me to develop an understanding of the subject.

I would like to thank the Network for University Co-operation Tibet-Noway and its all staff, especially Bente Elholm Bjørknes and Anak Bhandari. Thanks for your hard work, dedication, financial support and holding many fruitful seminars.

Another person to whom I owe a lot for his unconditional help from the very beginning of my life in Bergen is Professor Yngvar Gjessing. Your support has been critical for me in many occasions.

My thanks also goes to Thorsten Schilling, for your reading and giving lots of language corrections on this thesis.

I would like to thank in particular all the Tibetan students in Bergen for all the fun time we spent together, smoking, drinking or talking just about anything. I must thank the University of Tibet for allowing me leave from the work during my study in Norway.

Finally, I want to dedicate this work to my lovely wife Chungda. Thanks for your continuing and loving support without complaints, taking care of our children while I was away, giving me the chance to focus on this thesis and finish it in time.

Rigzin Wangdue
February 3, 2010

Contents

Acknowledgements **i**

1 Introduction **1**

2 Mathematical preliminaries **3**
2.1 Vector Space Over Finite Field 3
2.2 Linear Equation System 4
2.3 Matrix Notation 5
2.4 Solving Linear Equation System 6
2.5 Computational Complexity 7

3 Gluing and Agreeing **11**
3.1 The Gluing Algorithm 11
3.1.1 Gluing a Pair of Symbols 12
3.1.2 Gluing A List of Symbols 13
3.1.3 The Gluing1 Algorithm 15
3.2 Sorting Symbols 17
3.3 Agreeing Procedure and Agreeing-Gluing Algorithm 18
3.3.1 The Agreeing Procedure 19
3.3.2 Agreeing-Gluing Algorithm 21

4 Multiple Side Linear Equation **23**
4.1 Coset Covering 24
4.1.1 Coset and Maximal Coset 24
4.2 Minimal Cosets Covering 29
4.2.1 Greedy Approximation Algorithm 30
4.2.2 An Exact Algorithm 31
4.3 Procedure of Linearization 32

5 Gluing on MSLEs **37**
5.1 Gluing A Pair MSLEs 37

5.1.1 Consistency 37
5.1.2 Gluing 39
5.2 Gluing A List of MSLE 41
5.3 MSLE-Gluing 42
5.4 Implied System 42
5.4.1 Implication 43
5.4.2 Procedure of Reduction 44
5.5 Edge Removal 47

6 Experimental Results 49
6.1 Cosets Covering 50
6.2 Sorting 50
6.3 Comparison of Gluing1 and MSLE-Gluing 50
6.4 Reducing of Gluing operations 53
6.5 Edge Removing 53

7 Discussion 57
7.1 Discussion 57
7.2 Conclusion 57

A Algorithms Written in Mathematica 59
A.1 Auxiliary Routines 60
A.2 Instance Generator 62
A.3 Algorithms in Chapter 3 63
A.4 Algorithms in Chapter 4 66
A.5 Algorithms in Chapter 5 69

B Sample Experiments 73
B.1 Experimental Environment 75

C Data Corresponding to Figure 6.9 77

References 79

List of Figures

3.1 Search Tree 18

6.1 Comparison of minimal coset covering for $n = 4$. 50
6.2 Comparison of minimal coset covering for $n = 5$. 51
6.3 MSLE-Gluing operations on unsorted and sorted instance. . . 51
6.4 Comparison of the number of Gluing1 and MSLE-Gluing operations at each tree depth. $n = 16$ 52
6.5 Comparison of the number of Gluing1 and MSLE-Gluing operations at each tree depth. $n = 32$ 52
6.6 Comparison of the number of Gluing1 and MSLE-Gluing operations at each tree depth. $n = 48$ 53
6.7 Reduction of gluings on the MSLE-Gluing. 54
6.8 Edge removing procedure on MSLE-Gluing and Gluing1. . . . 54
6.9 Edge removing procedure on MSLE-Gluing. $n = 64$ 55

List of Algorithms

1 Glue(S_i,S_j) 13
2 GlueSet(S) 14
3 TreeGluing(S) 16
4 Sorting(S) 19
5 AgreePair(S_1, S_2) 20
6 Agree-Gluing(S) 21
7 MakePairs(V) 25
8 MaximalCosets(V) 27
9 MinimalCosetSetCover1(V, M) 30
10 MinimalCosetCover2(V, M) 32
11 LinearEquation 34
12 MSLEGlue(e_1, e_2) 39
13 MSLEGluePair(E_p, E_q) 41
14 MSLEGlueSet(E) 41
15 MSLE-Gluing(E) 43
16 MSLE-GluingReduction(E) 46
17 MSLE-GluingEdgeRemoval(E) 48

Chapter 1

Introduction

The design of a typical modern symmetric key cipher enables potential attackers to represent it as a system of multivariate polynomial (MP) equation system over a finite field:

$$f_1(X_1) = 0, f_2(X_2) = 0, \ldots, f_m(X_m) = 0, \tag{1.1}$$

where f_i are polynomials and X_i are subsets of the variable set X.

To solve this equation system is then equivalent to break the cipher. Therefore, any general algorithm to solve a set of multivariate polynomial equations can be used for attacking the cipher. Thus, it is very important to know how efficient algorithms are to find a solution to such a system and thus break the cryptosystem.

This kind of attack is commonly referred to as an algebraic attack, which focuses on solving multivariate polynomial equation systems that represent the ciphers. It has the advantage that only a few number of known plain-text and cipher-text pairs are needed to set up an equation system which describes the cipher. Finding solutions efficiently to the system is a hard problem in the cryptanalysis of modern ciphers. This problem belongs to the class of NP-complete problems, because any instance of the 3-Satisfiability problem can be reduced to an equation system in the form of (1.1)[1], which was shown by Cook in 1971, see [Coo71] to be in the class of NP-complete problems.

Conventional solving strategies for multivariate polynomial equation systems are often based on Gröbner bases techniques. Given a polynomial equation system in the form (1.1), one computes a basis for the ideal generated by the set of given polynomials, i.e, one tries to build a simpler, easier to solve equation system from which one could obtain the solutions to the input equation system by backward substitutions. Once a basis has been obtained,

[1]The proof of this reduction is due to Michael Garey and David Johnson[GJ79]

it is possible to investigate what kind of solutions exist and if possible, to compute them.

When equations are in $\mathbb{F}_2$, then one way to solve the system is to write the equation system in algebraic normal form (ANF), then converted into SAT-problem and use SAT-solvers, see [BCJ07].

An alternative approach to solve an equation system (1.1) in which each f_i only depends on small number of variables X_i, is the family of so called Gluing-Agreeing Algorithms [RS06]. The aim of this group of algorithms is to find the common solutions for each equations in (1.1) from the solution set in $\mathbb{F}_q$ by guessing a solution and verifying it. In practice the number of intermediate solutions grows fast.

Another recently proposed approach is the so called Multiple Side Linear Equations (MSLE) [sem09a]. The method is supposed to have a smaller number of intermediate solutions. The MSLE approach builds linear equation systems from the solution set of each f_i and then the Gluing algorithm is applied to solve the whole system.

The main matter of this thesis is implementing all algorithms described in the thesis, to compare number of intermediate solutions (gluing operations) occurring while system (1.1) is solved by the Gluing algorithm and the new Gluing algorithms on MLSE presented in this thesis.

The work on this thesis gives a summary of Gluing-Agreeing algorithms. A detailed description on constructing MSLE and the application of the Gluing algorithm on MSLE. Furthermore independent experimental results obtained by the implementation are presented and several examples are included to clarify the understanding of the algorithms.

Chapter 2

Mathematical preliminaries

This chapter will introduce some of the basic mathematical backgrounds and notations necessary to use throughout this thesis. We will summarize some theorems, definitions and notations without presenting proofs for theorems, that is beyond the scope of this thesis. Interested reader may consult [C.L06],[Fra03],[Gol96].

2.1 Vector Space Over Finite Field

Throughout this thesis $\mathbb{F}_q$ stands for a finite field with q elements, where $q = p^n$ and p is a prime. The addition operation is denoted by $(+)$, we may simply omit the$(\cdot)$ for multiplication. For instance, x_1x_2 indicates product of the two elements x_1 and x_2.
Let V be a set of all ordered n-tuples of elements of $\mathbb{F}_q$ with addition defined by $(u_1, u_2, \dots u_n) + (v_1, v_2, \dots v_n) = (u_1 + v_1, u_2 + v_2, \dots u_n + v_n)$ and scalar multiplication defined by $\gamma(u_1, u_2, \dots, u_n) = (\gamma u_1, \gamma u_2 \dots \gamma u_n)$, where $u, v \in V$ and $\gamma \in \mathbb{F}_q$. We say that V is a vector space over finite field $\mathbb{F}_q$ if it has the following three properties:

- The zero vector is in V.
- For each u and v in V, the sum $u + v$ is in V.
- For each u in V and each scalar c in $\mathbb{F}_q$ the vector cu is in V.

If vector subset U of V fulfill above three properties, then we say that U is a subspace of V. We will denote an n-tuple vector space over a finite field by $\mathbb{F}_q^n$. In this thesis $\bar{0}$ denote the column-vector whose entries are all 0. Similarly $\underline{0}$ denote the zero row-vector.
Consider a vector space V over $\mathbb{F}_q$ and $v_1, v_2, \dots, v_m \in V$, then a linear

combination of these vectors is a vector of the form:

$$\alpha_1 v_1 +, \ldots, +\alpha_m v_m$$

where $\alpha_i \in \mathbb{F}_q$. A set of vectors $v_1, \ldots, v_t$ is called linearly independent if the vector equation

$$x_1 v_1 + x_2 v_2 +, \ldots, +x_t v_t = 0$$

has only the trivial solution, i.e, all x_i are 0. In a vector space $\mathbb{F}_q^n$, there exits a set of linearly independent vectors $b_1, \ldots, b_r$, such that all vectors in the space can be represented by the linear combination of these vectors. Such vectors $b_1, \ldots, b_r$ are called basis of the space. The space is said to be generated by the basis. Let r be the number of basis, then the space generated by these basis is called r-dimensional vector space.
Throughout this thesis, all examples are illustrated in $\mathbb{F}_2$. Elements of this field are called bits. Elements of $\mathbb{F}_2^n$ are called binary vectors or Boolean vectors. A binary vector of length n may be represented by a decimal number, for instance, decimal form of binary vector (1011) is $1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 11$.
The inner product of two vectors $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$ in $\mathbb{F}_q^n$ is defined as

$$\langle x, y \rangle = \sum_{i=1}^{n} x_i y_i.$$

2.2 Linear Equation System

A linear equation over $\mathbb{F}_q$ in the variables $X = \{x_1, \ldots, x_n\}$ is an equation that can be written in the form

$$a_1 x_1 + a_2 x_2 + \ldots + a_n x_n = b, \tag{2.1}$$

where b and coefficients $a_1, \ldots, a_n$ are in $\mathbb{F}_q$, usually known in advance. A collection of one or more linear equations of variables in the same set X are called linear equation system. For example:

$$\begin{cases} x_1 + x_2 + x_4 = 0 \\ x_2 + x_3 = 1 \\ x_1 + x_4 = 1 \end{cases} \tag{2.2}$$

A solution to (2.2) is a list $(s_1, s_2, \ldots, s_n)$ of values that is a true statement when the values of $s_1, s_2, \ldots, s_n$ are substituted to $x_1, \ldots, x_n$, respectively. For instance,(1,1,0,0) is a solution to (2.2) over $\mathbb{F}_2$, because, when these values are substituted in (2.2) for x_1, x_2, x_3, x_4 respectively, the equality will hold. If such a solution exists then the equation system is called consistent.

2.3 Matrix Notation

A linear equation system is commonly represented as an equation of the form:

$$AX = b, \tag{2.3}$$

where A is an $m \times n$ matrix with entries in $\mathbb{F}_q$. Usually A is called coefficient matrix. That is, coefficients of each variable aligned in columns, for instance the coefficient matrix for (2.2) can be written in the following form:

$$A = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix}.$$

Each element in a row indicates the coefficient of corresponding variable. For instance, the second row represents $0x_1 + 1x_2 + 1x_3 + 0x_4$, while X consists of these four variables. The set of variables in X is represented as a column vector:

$$X = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}.$$

Now we can write (2.2) as:

$$\begin{pmatrix} 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix} X = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}. \tag{2.4}$$

Note, AX is defined only if the number of column of A equals the number of entries in X.

If the constant column from the right hand side is added to the right of the coefficient matrix, then it is called augmented matrix of the system. The augmented matrix for our example of (2.2) is

$$\left(\begin{array}{cccc|c} 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \end{array}\right)$$

we denote an augmented matrix as:

$$(A|b)$$

If $b = \bar{0}$(the zero vector), then the matrix equation (2.3) is said to be homogenous equation system. Such a system always has at least one solution,

namely, $X = \bar{0}$, it is called the trivial solution. The collection of all solutions to the homogenous system is called null space. We denote the basis of null space $AX = \bar{0}$ by $Null(\text{A})$.
Finally, we present following two theorems, these two theorems are the most important theories in constructing linear equation systems, cf.Chapter4.

Theorem 2.3.1 *The null space of an $m \times n$ matrix is a subspace of $\mathbb{F}_q^n$. Equivalently, the set of all solutions to a system $AX = \bar{0}$ of m homogeneous linear equations in n variables is a subspace of $\mathbb{F}_q^n$.*

Proof, See [C.L06].

Theorem 2.3.2 (Rank Theorem) *If a matrix A has n columns and rank r (cf.Section2.4), then the number of linearly independent vectors in $Null(A)$ is $n - r$.*

Proof, See[C.L06].

2.4 Solving Linear Equation System

A common approach to solve a linear equation system is using the Gaussian elimination algorithm to determine, the row reduced echelon form (RREF) of the augmented matrix. From this the solution can easily be read out.
The process of Gaussian elimination has two parts. the first part reduces a matrix to row echelon form using elementary row operations while the second transform the matrix into reduced row echelon form. The standard Gaussian elimination algorithm takes an $m \times n$ matrix A over a field $\mathbb{F}_q$ and applies successive elementary row operations:

- multiply a row by a element in field $\mathbb{F}_q$ (the inverse of the left-most non-zero);
- subtract a multiple of one from another (to create a new left-most zero);
- exchange two rows (to bring a new non-zero pivot to the top).

The resulting matrix is called row echelon form if it fulfills the following three properties:

1. All nonzero rows are above any rows of all zeros;
2. Each leading entry of a row is in a column to the right of the leading entry of the row above it;
3. All entries in a column below a leading entry are zeros.

If the matrix in echelon form satisfies the following additional condition, then it is called reduced row echelon form (RREF):

4. The leading entry in each nonzero row is 1;
5. Each leading 1 is the only nonzero entry in its column.

The leading entry of a row is defined as the leftmost nonzero element. From the RREF of a matrix, it is easy to determine the rank of the matrix. The number of non-zero rows in RREF is the rank of the matrix. We will denote the rank of a matrix A by $Rank(A)$.

Example 2.4.1

We use Gaussian elimination algorithm to solve (2.2) in $\mathbb{F}_2$. First, write the system in augmented matrix form and then apply the algorithm:

$$\left(\begin{array}{cccc|c} 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \end{array}\right) \widetilde{(1)} \left(\begin{array}{cccc|c} 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \end{array}\right) \widetilde{(2)} \left(\begin{array}{cccc|c} 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 \end{array}\right)$$

$$\widetilde{(3)} \left(\begin{array}{cccc|c} 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{array}\right)$$

The first step is moving the last row to the top of the matrix. The second step subtract the second row from the first (bitwise XOR the two rows). The third step subtract the third row from the second obtained by step two.

From the result we can easily read the solution for the system, which is

$$\begin{cases} x_1 + x_4 = 1 \\ \qquad x_2 = 1 \\ \qquad x_3 = 0 \end{cases} \rightsquigarrow \begin{cases} x_1 = 1 + x_4 \\ x_2 = 1 \\ x_3 = 0 \end{cases} .$$

This implies that x_4 is a free variable, it's value could be any element in $\mathbb{F}_2$. so, the solutions to the the equation system (2.2) are $(1, 1, 0, 0), (0, 1, 0, 1)$.

2.5 Computational Complexity

This section will introduce some of the basic notions and ideas of complexity theory, which we need for estimating the running time and memory consumption of some of our algorithms described in the next chapters. The definitions follow from [DCP06, Lev07].

Definition 2.5.1 *[Lev07] A function $t(n)$ is said to be in $O(g(n))$, denoted $t(n) \in O(g(n))$, if $t(n)$ is bounded above by some constant multiplication of $g(n)$ for all large n,i.e., if there exists some positive constant c and some nonnegative integer n_0 such that*

$$t(n) \leq cg(n), \forall n \geq n_0$$

A function $t(n)$ is said to be in $o(g(n))$, if $\frac{t(n)}{g(n)} \to 0$ while n tends to infinity.

The $O(g(n))$ representation is always simplified by only taking the most dominant terms of a function. When faced with a complicated function like $8n^3 + 7n + 5$, the common approach to simplify is just to replace it with $O(g(n))$, where $g(n)$ is as simple as possible. In this example n^3 dominates (grows faster then the rests as n goes to infinity) the rest of the terms, so we use $O(n^3)$.
Some commonsense rules that help simplify function by omitting dominated terms:

1. Multiplicative constants can be omitted;
2. n^a dominates n^b if $a > b$;
3. Any exponential dominates any polynomial;
4. Any polynomial dominates any logarithm.

An algorithm that has time complexity $O(n^c)$ for some constant $c > 1$ is said to be polynomial and time complexity $O(c^n)$ for some constant $c > 1$ is said to be exponential. In practice, any algorithm that has a time complexity that is polynomial or less is considered efficient.

Definition 2.5.2 *[DCP06] Class P is a class of decision problems which are problems can be answered by **yes** or **no** in polynomial time by a deterministic Turing machine.*

Definition 2.5.3 *[DCP06] The class NP is defined as the set of all decision problems to which a solution can be found and verified in polynomial time by a non-deterministic Turing machine.*

A non-deterministic algorithm is a two-stage procedure that takes as its input an instance I of a decision problem and does the following:

Nondeterministic (guessing) stage: An arbitrary string S is generated that can be thought of as a candidate solution to the given instance I, but it is also possible that it is not the right solution at all.
Deterministic (verification) stage: The algorithm takes both I and S as its

input and outputs **Yes** if S is a solution to the instance, **No**, otherwise. Almost all problems in class NP have in common an exponential growth of choices, as a function of input size, from which a solution can be found.

Example 2.5.1

Let V be a finite set of size n, and let $C = \{c_1, c_2, \ldots, c_k\}$ be a family of subsets of V, that is $c_i \subseteq V$ such that the union of all c_i equals V. Given V and C as input to a algorithm, the algorithm should output the minimum number of c_i, which contain all elements in V. This problem is called *minimal set cover problem*. It was one of Richard Karp's 21 NP-complete problems shown to be in the class of NP-complete in 1972, see [Kar72]. Roughly spoken, NP-complete problem is a problem in NP that is as difficult as any other problems in this class. We will encounter a similar problem in Chapter 4.

The algorithm first proposes a list of candidate solutions (nondeterministic stage). Secondly, for each element it is determined whether it is a solution or not. The worst case for this algorithm is checking all subsets of C. there are 2^k subsets, which is exponential.

Chapter 3

Gluing and Agreeing

This chapter will introduce an approach to solve system (1.1), called Gluing. The algorithm belongs to the family of Gluing/Agreeing algorithms and works in case of sparse equation system (1.1), where the size of each X_i is bounded. We will also briefly introduce a related algorithm called Agreeing algorithm. Its aim is to delete irrelevant partial solutions from the solution candidates. The Agreeing algorithm was firstly discovered by Arkadij Zakrevskij and Irina Vasilkova [ZV00], afterwards Igor Semaev and Håvard Raddum developed it along with Gluing algorithm, see [Rad04, RS06]. Asymptotical complexity estimates are in [Sem08, Sem09b]. Hardware implementation of the Agreeing method is described in [Sem09c].

3.1 The Gluing Algorithm

Let (1.1) be over a finite field $\mathbb{F}_q$ with q elements and all variables X_i,$1 \leq i \leq m$ are in $X = X_1 \cup X_2 \cup \ldots \cup X_m$. Each f_i depends only on variable X_i. If for all $1 \leq i \leq m : |X_i| \leq l$, for some integer l, then we call the system l-sparse.

We can see that each f_i defines a mapping: $f_i : \mathbb{F}_q^l \to \mathbb{F}_q$. The set of solutions to a particular $f_i(X_i) = 0$ is a set of vectors $V \subseteq \mathbb{F}_q^l$ for which $f_i(v) = 0$, where $v \in V$. To find all assignments to $f_i(X_i) = 0$, one tries all q^l vectors in $\mathbb{F}_q^l$, that is an exponential number of trials in l. This explains why we emphasize that the system should be sparse for our approach, as mentioned in the introduction.

Obviously, the equation $f_i(X_i) = 0$ is determined by the set of variables X_i and the set of l-tuple vectors in $\mathbb{F}_q$, where f_i is zero. This leads to the following definition.

Definition 3.1.1 (Symbol) *A symbol $S = (X, V)$ corresponding to an equation f consists of an ordered set of variables X and a list of vectors $V = \{v_1, \ldots, v_k | v_i \in \mathbb{F}_q^{|X|}\}$ which contains all satisfying assignments to $f(X) = 0$ in variables X.*

With this definition we can represent system (1.1) by

$$E = \{S_1, S_2, \ldots, S_m\} = \{(X_1, V_1), (X_2, V_2), \ldots, (X_m, V_m)\}. \tag{3.1}$$

From now on when we write an equation in symbol form, e.g. (X_i, V_i) is equivalent to the equation $f_i(X_i) = 0$. The solutions to the equation system (1.1) can be represented by (X, V), where V is a set of vectors in $\mathbb{F}_q^n$ to all variables in the set X which satisfy all equations in (1.1).

3.1.1 Gluing a Pair of Symbols

First, we describe the Gluing procedure and how to get common solutions to two given symbols. Consider two symbols as the input to our algorithm:

$$(X_1, V_1), (X_2, V_2). \tag{3.2}$$

One defines two sets of variables $Z = X_1 \cup X_1$ and $Y = X_i \cap X_j$. One can sort variables X_1 and X_2 in Y. Note that the entries in V_i under the projection of variables Y are also permutated after the sorting procedure. The common vectors in Z are defined by $U = \{a_1, b, a_2\}$, where $(a_1, b) \in V_1$, $(b, a_2) \in V_2$ and a_i are all entries under the projection of variables $(X_i \setminus Y)$. We call it a $(X_i \setminus Y)$-vector. b is a vector under projection of variables Y, the common entries in X_1 and X_2. We call it Y-vector.

We denote $(a_1, b, a_2) = (a_1, b) \circ (b, a_2)$, and say that (a_1, b, a_2) is the gluing of (a_1, b) and (b, a_2). The new symbol is then denoted by $(Z, U) = (X_1, V_1) \circ (X_2, V_2)$.

We summarize the pairwise gluing procedure into following pseudo code, Algorithm 1. The variables in symbol S are denoted by $X(S)$.

Note that when there are no intersection variables in X_1 and X_2, i.e., $Y = \emptyset$, the Gluing algorithm only returns trivial solutions. I.e., all Z-vectors U in variables $Z = X_1 \cup X_2$. The overall complexity of this algorithm is bounded by

$$O(|U| + |V_1| + |V_2|),$$

operations, with rewriting and comparing vectors in $\mathbb{F}_q^{|Z|}$ and $|V_1| + |V_2|$ sorting steps.

Algorithm 1: Glue(S_i,S_j)

Input : Two symbols $S_i : (X_i, V_i)$ and $S_j : (X_j, V_j)$
Output: A new glued symbol, if the two symbols are gluable

$Y \leftarrow X(S_i) \cap X(S_j)$
$Z \leftarrow X(S_i) \cup X(S_j)$
if $Y \neq \emptyset$ **then**
  $U \leftarrow \{(a_i, b, a_j) | (a_i, b) \in V_i, (b, a_j) \in V_j\}$
else
  $U \leftarrow \{(a_i, a_j) | a_i \in V_i, a_j \in V_j\}$
Return (Z, U)

Example 3.1.1

Consider the symbols S_1, S_2 in variables $X = \{x_1, x_2, x_3, x_5\}$

$$S_1 = \begin{array}{c|c|c|c} & x_2 & x_3 & x_5 \\ \hline a_1 & 1 & 1 & 1 \\ a_2 & 1 & 0 & 1 \\ a_3 & 0 & 1 & 1 \end{array}, \quad S_2 = \begin{array}{c|c|c} & x_1 & x_2 \\ \hline b_1 & 0 & 1 \\ b_2 & 0 & 0 \\ b_3 & 1 & 1 \end{array}.$$

After gluing the two symbols above we become:

$$S_1 \circ S_2 = \begin{array}{c|c|c|c|c} & x_1 & x_2 & x_3 & x_5 \\ \hline c_1 & 0 & 0 & 1 & 1 \\ c_2 & 0 & 1 & 0 & 1 \\ c_3 & 1 & 1 & 0 & 1 \\ c_4 & 0 & 1 & 1 & 1 \\ c_5 & 1 & 1 & 1 & 1 \end{array}.$$

The resulting vectors c_1, c_2, c_3, c_4, c_5 are then all satisfying assignments to S_1 and S_2.

3.1.2 Gluing A List of Symbols

To solve system (1.1) one can apply Algorithm 1 successively to the symbols. Consider as input to the algorithm the set of symbols (3.1). First, we glue S_1 and S_2. This gives a new symbol $(Z, U) = (X_1, V_1) \circ (X_2, V_2)$. Now let (Z, U) be the first argument of the algorithm and the next symbol from the set be the second argument. After running the algorithm we will get another new symbol. Apply this procedure for the rest of the symbols in the set

Algorithm 2: GlueSet(S)

Input : A list of symbols: $S_1, S_2, \ldots, S_m$
Output: All system solutions U.

$m \leftarrow length(S)$
T$\leftarrow S_1$
$k \leftarrow 2$
while $k \leq m$ **do**
  T$\leftarrow$ **Glue** (T,S_k)
  $k \leftarrow k+1$
Return $T = (X, U)$

continuously. A suggested implementation of this procedure is depicted in Algorithm 2.

The running time of this algorithm is

$$O(\sum_{k=1}^{m-1} |U_k||V_{k+1}| + |U_k| + |V_{k+1}|) = O(\sum_{k=1}^{m-1} |U_k| + m), \tag{3.3}$$

operations with vectors in $\mathbb{F}_q^n$, where n is the number of variables in the system and m is the number of equations. These two numbers may grow with respect to the size of the input instance. The size of the ground field q and sparsity l are fixed. The memory consumption of this algorithm is of the same magnitude as the running time (3.3). Here $(X(k), U_k) = (X_1, V_1) \circ \ldots \circ (X_k, V_k)$ and (3.3) is the cost of $m-1$ gluings. The set U_k consists of all solutions to the first k equations in variables $X(k) = X_1 \cup \ldots \cup X_k$. The sequence of $|U_k|$ fully characterized the algorithm's running time. For the asymptotical analysis of $|U_k|$ see [Sem08, Sem09b].

Example 3.1.2

Let us add another symbol S_3 to the symbols in Example 3.1.1:

$$S_1 = \begin{array}{c|c|c|c} & x_2 & x_3 & x_5 \\ \hline a_1 & 1 & 1 & 1 \\ a_2 & 1 & 0 & 1 \\ a_3 & 0 & 1 & 1 \end{array}, \quad S_2 = \begin{array}{c|c|c} & x_1 & x_2 \\ \hline b_1 & 0 & 1 \\ b_2 & 0 & 0 \\ b_3 & 1 & 1 \end{array}, \quad S_3 = \begin{array}{c|c|c|c} & x_2 & x_4 & x_5 \\ \hline d_1 & 0 & 1 & 0 \\ d_2 & 0 & 1 & 1 \\ d_3 & 1 & 0 & 0 \end{array}.$$

The algorithm first glues S_1 and S_2, which we have already done in our previous Example 3.1.1. The glued symbol is now T. Now we just need to

glue T and S_3

$$T \circ S_3 = \begin{array}{c|c|c|c|c} & x_1 & x_2 & x_3 & x_5 \\ \hline c_1 & 0 & 0 & 1 & 1 \\ c_2 & 0 & 1 & 0 & 1 \\ c_3 & 1 & 1 & 0 & 1 \\ c_4 & 0 & 1 & 1 & 1 \\ c_5 & 1 & 1 & 1 & 1 \end{array} \circ \begin{array}{c|c|c|c} & x_2 & x_4 & x_5 \\ \hline d_1 & 0 & 1 & 0 \\ d_2 & 0 & 1 & 1 \\ d_3 & 1 & 0 & 0 \end{array} .$$

$$= \begin{array}{c|c|c|c|c|c} & x_1 & x_2 & x_3 & x_4 & x_5 \\ \hline e_1 & 0 & 0 & 1 & 1 & 1 \end{array} .$$

After the second gluing step we find that $(x_1, x_2, x_3, x_4, x_5) = (0, 0, 1, 1, 1)$ is the only solution to this particular example. It is also possible that the output of the algorithm contains more than one solution.

3.1.3 The Gluing1 Algorithm

In this section we introduce a variant of the Gluing algorithm called Gluing1 algorithm. It has the same time complexity as the algorithm we introduced in Section 3.1.2, but requires only `poly(n)` bits memory. The algorithm walks through a search tree with backtracking[1] in order to find a path in which partial solutions to the equation could be revealed. To apply the algorithm one defines a search tree rooted at $\emptyset$. At each tree level a symbol is defined and the tree levels are labeled by d, $1 \leq d \leq m$. For instance at $d = 1$ symbol S_1 is located and all vectors $v_i \in V_1(S_1)$ can be referred to as nodes at this tree level. Nodes at level $d = 2$ are labeled by some $b \in V_2(S_2)$. A node at level 1 which is labeled a is connected to a node at level 2, labeled by b, whenever $a \circ b$ is possible. Keep the value of $a \circ b$ and try to find a node c at the next tree level and attempt to $a \circ b \circ c$. If this is possible keep this value and go further. If not, the algorithm goes one step back and tries gluing a with another node at level 2, and so on. If the algorithm finds a path which goes from the root to the bottom, namely, $a \circ b \circ c \circ \ldots \circ d$, then this is a solution to the system. A suggested implementation of the above procedure is Algorithm 3.

A list of variables $M_1, M_2, \ldots, M_m$ is defined. Initially, all M_i are empty. Each M_i is designed to contain the partial solution generated by the algorithm at each tree level and M_m contains the final solution. The current tree level, where the algorithm is running is denoted by d.

[1] Roughly, the technique tries to find a final solution to a problem by repeatedly extending a partial solution. The partial solution shrinks(backtracking) whenever it can not be further extended. For details see[Val98].

Algorithm 3: TreeGluing(S)

```
Input  : A list of symbols S_1, S_2, ..., S_m
Output: if the equation system has solutions, return the solution.

d ← 1
s_1 = 1, s_2 = 1, ..., s_m = 1 // Initialize all s_i
M_1, M_2, ..., M_m ←Initialize all M_i as empty variables
while 0 < d ≤ m do
    T ← M_d ∘ v_{s_d} ∈ S_d
    if T ≠ ∅ //the Glue operation returns a solution then
        M_{d+1} ← T
        s_d ← s_d + 1
        d ← d + 1
    else
        s_d ← s_d + 1
        if s_d > length(V_d) then
            s_d ← 1
            d ← d − 1
if d = m then
    Return M_d
else
    Return No solution
```

A set of integer variables $s_1, s_2, \ldots, s_m$ are used for keeping track of which vector is already tried at each tree level. Initially, the value of each s_i is 1. Every s_i is specific for a symbol S_i and points to the next not yet tested vector in the set $V_i(S_i)$. The algorithm returns a solution if it finds a path from the root to $d = m$ and it halts if $d < 1$. In that case all vectors in S_1 tried and no gluing is possible. This indicates that the system dose not have any solutions.

Note that Algorithm 3 only gives a single solution, while the input system might have more.

It is not difficult to output all solutions by a slightly modified algorithm. One can add an “ungluable” symbol at the end of the input instance. This forces the algorithm find all possible pathes from the root to level $d = m$. The algorithm aborts after exhausting all vectors in $V_1(S_1)$ whenever the system has one or more solutions. At each iteration the solution can be read at tree level $d = m$, namely M_m is a solution.

Example 3.1.3

In this example we use the symbols from Example 3.1.2. Assume they are

in the same order. We define the search tree by S_1, S_2, S_3 located at tree level $d = 1, 2, 3$ respectively. Initially the value of all variables M_1, M_2, M_3 are empty.

The initial value of s_1 is 1, therefore it points to a_1 at level 1. The value of M_1 becomes:

$$M_1 = a_1 = \left(\begin{array}{c|c|c} x_2 & x_3 & x_5 \\ \hline 1 & 1 & 1 \end{array} \right).$$

Now, we try to glue M_1 with vectors b at the next tree level. The initial value of pointer $s_2 = 1$, so it points to b_1 now. The algorithm tries to glue M_1 and b_1. This yields

$$M_2 = M_1 \circ b_1 = \left(\begin{array}{c|c|c|c} x_1 & x_2 & x_3 & x_5 \\ \hline 0 & 1 & 1 & 1 \end{array} \right)$$

but this time we can not find a vector in the third symbol which can be glued to M_2. Therefore, the algorithm backtracks and tries to glue M_1 and b_2. This is also a contradiction to the values of variable x_2. In this case, the next possible gluing is $M_2 = M_1 \circ b_3$, but there are no possible vectors at tree depth 3 which can extend the partial solution. At this point the algorithm tried all vectors in S_2, the pointer s_2 resets to 1 and goes back to the first symbol in the search tree. Recall that s_1 is already set to 2. It points to a_2 now. So, $M_1 = a_2$ and we see that after trying all vectors at tree level 2 the algorithm can not find a path to reach the bottom. This implies that no solutions can be found along this branch. The pointer s_2 is set to 1 and chooses a_3 from the first symbol. Now, $M_1 = a_3$ and the possible gluable element in the next tree level is b_2. This extends the partial solution to

$$M_2 = M_1 \circ b_2 \left(\begin{array}{c|c|c|c} x_1 & x_2 & x_3 & x_5 \\ \hline 0 & 0 & 1 & 1 \end{array} \right).$$

From tree level 3 we find that d_2 is the only possible gluing with M_2. The partial solution can be extended by gluing M_2 with d_2. This yields

$$M_3 = M_2 \circ d_2 = \left(\begin{array}{c|c|c|c|c} x_1 & x_2 & x_3 & x_4 & x_5 \\ \hline 0 & 0 & 1 & 1 & 1 \end{array} \right).$$

That is the final solution of the computation. In the search tree the algorithm walks through the path $a_3 \circ b_2 \circ d_2$. We depict the walks of the algorithm on the search tree in Figure 3.1.

3.2 Sorting Symbols

Consider in Example 3.1.2 that if we take another ordering of the symbols, say the symbols are in the order S_3, S_2, S_1, the number of intermediate solu-

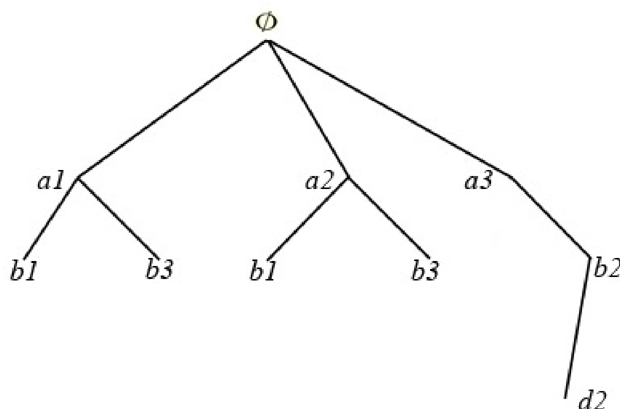

Figure 3.1: Search Tree

tions are 4 instead of 5 after the first gluing step. If we sort the symbols in the order S_1, S_3, S_2 much less intermediate solutions are yield, namely 1. The observation is that when two symbols have a bigger number of intersecting variables, the more restrictions occur on gluing operations. Therefore the number of intermediate solutions is lower. Considering this in terms of running time and memory sorting is recommended before applying the gluing procedure.

Here we present a simple sorting algorithm in Algorithm 4 which is developed by T. Schilling in his master's project, see [Sch08].

We start by scanning the given list to find two symbols T_1 and T_2 which have the minimal $|X_i \cup X_j|$. Then we remove these two symbols from the given list and store T_1 and T_2 in a new list. Then we scan the remaining list again, starting from the first symbol and compare each S_j. The symbol which has the maximal number of common variables with $X(T_1) \cup X(T_2)$ is then chosen and removed from the original list to append it to the newly constructed list. Repeatedly find and add to the new list each symbol which has the maximal number of intersecting variables with $X(T_1) \cup X(T_2)$ from the rest. The running time of the algorithm is bounded by $O(n^2)$.

3.3 Agreeing Procedure and Agreeing-Gluing Algorithm

In this section the Agreeing procedure is briefly described. The aim of this procedure is to eliminate all vectors (solution candidates) which can not be part of any solution to the whole system.

Algorithm 4: Sorting(S)

Input : A list of symbols $S = \{S_1, S_2, \ldots, S_m\}$
Output: Sorted symbol $S^{'}$

```
n ← |X(S)|          // Store the number of varables
T_1 ← S_1
T_2 ← S_2
for each S_i, S_j ∈ S do
    if |X(S_i) ∪ X(S_j)| < |X(T_1) ∪ X(T_2)| then
        T_1 ← S_i
        T_2 ← S_j  //Find pair S_i, S_j with smallest |X(S_i) ∪ X(S_j)|
S ← S \ {T_1, T_2}
S'[1] ← T_1
S'[2] ← T_2    // Store T_1 and T_2 in a new list
while |S| > 0 do
    c ← n
    for each S_i ∈ S do
        if |X(S') ∪ X(S_i)| < c then
            c ← |X(S') ∪ X(S_i)|
            e ← S_i
    S ← S\e
    Append e to S'
Return S'
```

3.3.1 The Agreeing Procedure

Given two equations in symbols form (3.2), one defines the set of variables $Y = X_1 \cap X_2$. The set of projections of V_1 on variables Y is defined by $V_{1,2}$, i.e, the set of Y-subvectors of V_1. Similarly, the set of projections of V_2 on variables Y is defined by $V_{2,1}$.

Definition 3.3.1 (Agree) *Two symbols S_i and S_j agree if and only if the two sets $V_{1,2}$ and $V_{2,1}$ are equal. Otherwise the two symbols are said to not agree.*

When the given two symbols not agree, we apply the agreeing procedure. Remove all v_i from $V_1(S_1)$ and all v_j from $V_2(S_2)$ whose Y-subvectors are not in $V_{1,2} \cap V_{2,1}$. Note, since $V_{1,2} \cap V_{2,1}$ is a intersection of two sets, there are no repeated elements occurring in this set. We can see that this procedure actually is deleting vectors in $V_i(S_i)$ that can not be part of any common solution to the two equations.

The new symbols after agreeing are defined by (X_1, V_1') and (X_2, V_2'), where $V_1' \subseteq V_1$ and $V_2' \subseteq V_2$. We summarize the Agreeing procedure in Algorithm 5.

Algorithm 5: AgreePair(S_1, S_2)

input : A pair of symbols $(X_1, V_1), (X_2, V_2)$
output: Agreed symbol $(X_1, V_1'), (X_2, V_2')$

$Y \leftarrow X_1 \cap X_2$
$V_{1,2} \leftarrow V_1(Y)$
$V_{2,1} \leftarrow V_2(Y)$ //$Y-$subvectors
if $V_{1,2} = V_{2,1}$ **then**
 Return $(X_1, V_1), (X_2, V_2)$
else
 $V_1' \leftarrow$remove all $v_i \in V_1$ whose $Y-$subvectors are not in $V_{1,2} \cap V_{2,1}$
 $V_2' \leftarrow$remove all $v_j \in V_2$ whose $Y-$subvectors are not in $V_{1,2} \cap V_{2,1}$
Return $(X_1, V_1'), (X_2, V_2')$

The complexity of this Agreeing algorithm is bounded by

$$O(|V_1| + |V_2|)$$

operations, as rewriting and comparisons, with vectors in $\mathbb{F}_q^n$.

Example 3.3.1

We agree the first and the third symbol from Example 3.1.2. First define the intersection variables Y, $Y = \{x_2, x_5\}$. From each symbol the Y-subvectors are defined by $V_{1,2} = \{(1\ 1), (0\ 1)\}$, and $V_{2,1} = \{(0\ 0), (0\ 1), (1\ 0)\}$.

Now look for the common elements in $V_{1,2}$ and $V_{2,1}$. We can see that the subset $\{(0\ 1)\}$ occurs in both sets. Those vectors whose projection on Y are not in this subset are irrelevant to the common solutions to the equations. Therefore, we remove a_1, a_2 from S_1 and d_1, d_3 from S_3. The remaining vectors in $V_1(S_1)$ and $V_2(S_2)$ build up the newly agreed symbols

$$S_1' = \begin{array}{c|c|c|c} & x_2 & x_3 & x_5 \\ \hline a_3 & 0 & 1 & 1 \end{array}, \quad S_3' = \begin{array}{c|c|c|c} & x_2 & x_4 & x_5 \\ \hline d_2 & 0 & 1 & 1 \end{array}.$$

When more than two symbols are considered vectors in V_i can be deleted by agreeing each pair of symbols until all pairs agree. This procedure is similar to the procedure of gluing a list of symbols. See Algorithm2 described in Section 3.1.2.

3.3.2 Agreeing-Gluing Algorithm

To solve equation system (1.1) one applies the Gluing algorithm to the new symbols generated by the Agreeing algorithm. The combined algorithm is called Agreeing-Gluing algorithm [Sem09b]. We summarize the algorithm in the following code.

Algorithm 6: Agree-Gluing(S)

```
input  : A list symbols S_1, S_2, ..., S_m
output: All system solutions
(Z, U) ← (X_1, V_1)
k ← 2
while k ≤ m do
    s ← k
    while s ≤ m do
        AgreePair((Z, U), (X_s, V_s))
        s ← s + 1
    (Z, U) ← Glue((Z, U), (X_k, V_k))
    k ← k + 1
Return (Z, U)
```

The complexity of this algorithm is

$$O(m(\sum_{k=1}^{m-1} |U_k^{'}| + 1)),$$

operations with vectors in $\mathbb{F}_q^n$ as showed in [Sem09b], where the symbol $(X(k), U^{'})$ is $(X(k), U_k) = (X_1, V_1) \circ \ldots \circ (X_k, V_k)$ after agreeing with $(m-k)$ symbols (X_i, V_i) for $k < i \leq m$.

More efficient and low memory cost Agreeing-Gluing algorithms are studied by Igor Semaev and Håvard Raddum in their papers, interested readers can refer to [RS06, Sem09b].

Chapter 4

Multiple Side Linear Equation

The general way of linearizing multivariate polynomial equations is to replace any monomials (product of several variables) by a new single variable. For instance monomial $x_1x_2x_3$ is replaced by y_{123}. Therefore, the original equation in variables X can be rewritten as a linear equation in the new variables Y. Kipnis and Shanir developed the so called Relinearization Technique[KS99] based on this idea. They expected that the relinearized system is uniquely solvable if the number of equations is bigger then the number of new variables. If the number of the equations is strictly less then the number of variables then the linear system has many solutions to y which do not correspond to any real x.

Inspired by [KS99], Courtois, Klimov, Patarin and Shamir proposed a variant algorithm called eXtended Linearization(XL)[CKPS00]. It was proposed as a technique which can be viewed as a combination of bounded degree Gröbner bases[BW93] and linearization. The basic idea of this approach is that the multivariate polynomial is expanded by multiplying each of the equation with all possible monomials of certain bounded degree and the number of expanded equations depend on the degree. The higher the degree, the higher the number of the equations. To linearize the expanded system one constructs a coefficient matrix of this equation system in which rows represent equations and columns represent coefficients. In order to solve it one applies gaussian elimination. One of the main drawback of the XL algorithm is that the number of equations grows fast, while equations are multiplied by monomials of a certain degree.

Several modifications to the XL algorithm have been proposed, one among the promising variants is eXtended Spars Linearization(XSL) [CP02]. This algorithm keeps the number of equations within some certain bounds. The authors claim that it might be possible to break AES(Advanced Encryption Standard)[DR99] using the XSL algorithm, but later Diem showed that some

different estimates of the algorithm complexity[Die04]. His analysis showed that the authors are rather optimistic in their original estimate.

In this chapter we will introduce a different approach of linearization in which linear equations are constructed from solution sets of multivariate polynomials instead of using the equations itself. Each multivariate polynomial equation in system (1.1) can be represented by a set of correspondent linear equation systems, we call them Multiple Side Linear Equation System (MSLE), such that the union of their solutions exactly coincides with the set of solutions of the initial multivariate polynomial equation. The concept of Multiple Side Linear Equation System and their possible use in the field of cryptanalysis is a rather new and unconventional subject.

4.1 Coset Covering

The main idea of coset covering is to cover solutions to $f_i(X_i) = 0$ by cosets.

In this section we will introduce algorithms to compute cosets from a given solution set of multivariate polynomials $f_i(X_i) = 0$. Then we apply set cover algorithms to determine the minimal number of cosets that cover the given solution set. Afterwards we will introduce a method to construct linear equation systems from cosets. Thus the solution set of the linear equation systems coincide the solution set of equation $f_i(X_i) = 0$.

4.1.1 Coset and Maximal Coset

Computing cosets is one of the main building block of our linearization approach. We begin this section by a definition adopted from[AP01].

Definition 4.1.1 (Coset) *If U is a vector subspace in $\mathbb{F}_q^n$ the set $a + U = \{a + u | u \in U, a \in \mathbb{F}_q^n\}$ is a coset of the subspace U, where a and u are vectors of length n.*

Recall that in our notation $\mathbb{F}_q^n$ stands for a n-dimensional vector space over a finite field $\mathbb{F}_q$. Binary operations in a vector space are commutative, therefore we only use the term *coset* . There should be no confusion when using this term instead of using *right coset* or *left coset*.

Since U is a subspace of $\mathbb{F}_q^n$ there exist a set of linearly independent vectors $B = \{b_1, b_2, \ldots, b_r\}$ in $\mathbb{F}_q^n$, such that each element in U can be written as a linear combination of these vectors. The set of vectors B are called basis of the subspace U.
With this property of a vector space, we can represent cosets in the form:

$$a + \langle b_1, b_2, \ldots, b_r \rangle. \tag{4.1}$$

We say that the coset is generated by the basis $b_1, b_2, \ldots, b_r$. The number of vectors in the basis B is the dimension of the coset.

Let $V = \{a_1, a_2, \ldots, a_s\}$ be a set of vectors in $\mathbb{F}_q^n$ and C be a coset in V. We say that C is maximal in V if there is no another coset that contains C. We now describe a method for constructing maximal cosets in V.

For a multivariate polynomial equation in $\mathbb{F}_q$, the solution set can be either a subspace or just a set of vectors. Given an arbitrary set of vectors $V = \{a_1, a_2, \ldots, a_s\} \subseteq \mathbb{F}q^n$. One takes any non-zero vector h from $\mathbb{F}_q^n$, a vector a from the given set V and computes pairs

$$P_h = \{a, a+h \mid a \in V, a+h \in V\}, \tag{4.2}$$

such that $a_j = a_i + h,\ i \neq j$. In this way, one constructs pairs for each non-zero $h \in \mathbb{F}_q^n$. It is possible for any h that there exists no pair or there exists one or more pairs. The suggested algorithm for computing pairs for V is Algorithm 7.

Algorithm 7: MakePairs(V)

```
input : A set of vectors V ⊆ F_q^n of size s
output: A list of pairs for V

for each non-zero h ∈ F_q^n do
    for i = 1, i < s do
        for j = i + 1, j ≤ s do
            if a_i = a_j + h then
                P_h ← Append (a_i, a_j) to P_h
    P ← Append P_h to P
Return P
```

Since the algorithm computes pairs for each non-zero h in a $n-$dimensional vector space. There are $2^n - 1$ choices for h. For each h it computes pairs for each element in V. Therefore, the overall running time for computing pairs is $O(2^n|V|)$.

Theorem 4.1.1 *If P_h contains only one pair $a, a+h$, then P_h is a coset itself.*

Proof. We can represent the coset as $a + \langle h \rangle$, where $a \in V,\ h \in \mathbb{F}_q^n$. It is easy to see that h generates a subspace with only two elements, i.e, the zero vector and itself. Hence the elements $a + \bar{0} = a$ and $a + h = a + h$ are a coset.This proves Theorem 4.1.1.

Remark 4.1.1 *One vector in $\mathbb{F}_q^n$ is a coset.*

Proof: We see that the zero vector in $\mathbb{F}^n$ is a subspace of $\mathbb{F}^n$. For any vector $a \in \mathbb{F}^n$, such that $a = a + \bar{0}$. This proves the Remark.
It is also true if P_h contains just two pairs $a, a+h$ and $b, b+h$, then P_h is a coset. Because, let $a + b = g$, then $P_h = a + \langle g, h \rangle$.

When P_h contains more than two pair one takes the row-vector h and constructs a matrix G by

$$hG = \underline{0}. \tag{4.3}$$

That is equivalent to asking for solutions of

$$\langle Y, h \rangle = 0. \tag{4.4}$$

Recall that $\langle Y, h \rangle$ is the inner product of $Y = (y_1, \ldots, y_n)$ and the nonzero vector $h \in \mathbb{F}_q^n$. Solutions to the equation above can be viewed as $Null(h)$. By Theorem 2.3.2, since h consists of only one vector, i.e, it is of rank 1, one obtains $n-1$ linearly independent solutions to the linear equation. Let these solutions be the columns of matrix G. For each $a \in P_h$, compute a set V'.

$$V' = \{a' = aG | a \in P_h\}, \tag{4.5}$$

where a is a row-vector. The equation above can be viewed as a mapping $\mathbb{F}_q^n \to \mathbb{F}_q^{n-1}$. Vectors in V' have the length $n-1$. Both elements in the pair $a, a+h \in P_h$ will map to the same element in $\mathbb{F}_q^{n-1}$ by the distribution low of matrix multiplication $(a+h)G = aG + hG$. Since $hG = \underline{0}$, we get $(a+h)G = aG$.

If V' contains a coset $a^* + \langle b_1^*, \ldots, b_r^* \rangle$, then P_h contains the coset $a + \langle b_1, \ldots b_r, h \rangle$, where a is a solution to $YG = a^*$ and b_i is a solution to $YG = b_i^*$.

One can observe that we are doing the same computation, but the size of the problem is getting smaller and smaller until the size of the problem reaches a *base case* (P_h consists of one pair). This is the main property of the divide-and-conquer strategy[KET06] in which the problem is reduced into subproblems that are themselves smaller instances of the same type of problem, then solve them recursively. We summarize our approach of computing cosets in a divide-and-conquer based algorithm in Algorithm 8.

The input to the algorithm P_h is a set of pairs which correspondent to a particular h. In the last For loop for each k a coset is computed appended to the variable *coset*. The recursion returns solutions while the input of the function is reduced into only one pair.

Now we discuss the running time of Algorithm 8. It is obvious that the most time consuming computation is computing pairs, which is upper bounded by $O(2^n|V|)$. The other operations take linear time. Let $T(n)$ be

Algorithm 8: MaximalCosets(V)

input : A set of vectors $V \subseteq \mathbb{F}_q^n$
output: A list of maximal cosets in V

```
P_h ← MakePair(V)
if P_h contains one pair then
    Return a+ < h >
if P_h contains more then one pair then
    G ← Solve(⟨X, h⟩ = 0̄)
    for ∀a ∈ P_h do
        V′ ← {aG}
    MaximalCosets(V′)      // Call recursion
    for each maximal cosets in V′ do
        compute maximal cosets in V
Return maximal cosets in V
```

the running time of routine MaximalCosets, then after recursion we have the following:

$$\begin{aligned} T(n) \leq & \quad 2^n(T(n-1) + n^2 2^n) \\ \approx & \quad 2^{n+(n-1)+,\ldots,+1} \\ \approx & \quad 2^{\frac{n(n-1)}{2}} \\ \approx & \quad 2^{\frac{n^2}{2}+\frac{n}{2}} \\ \approx & \quad 2^{\frac{n^2}{2}(1+(\frac{1}{n}))} \end{aligned}$$

In practice n is a small number like 3,4,5 and 6. The overall running time of Algorithm 8 is

$$2^{\frac{n^2}{2}(1+o(1))}, \tag{4.6}$$

where $o(1)$ is a constant.

Example 4.1.1

To get a better understanding of all algorithms we mentioned above we give an example. Vectors are represented in decimal form, cf. Section 2.1. Let a subset $V \subseteq \mathbb{F}_2^4$ be given as following:

$$\{(0001), (0011), (0100), (0101), (1001), (1011), (1101)\}. \tag{4.7}$$

Written in decimal form the vectors are $\{1, 3, 4, 5, 9, 11, 13\}$. Now, for each non-zero $h \in \mathbb{F}_2^4$ we compute pairs. For $h_1 = (0001) = 1$, the pairs are

$P_{h_1} = \{(0100), (0101)\} = \{(4, 5)\}$. The same computation is applied for the rest of h, we obtain

$$\begin{array}{rl}
P_{h_2}: & \{(1,3),(9,11)\} \\
P_{h_4}: & \{(1,5),(9,13)\} \\
P_{h_5}: & \{(1,4)\} \\
P_{h_6}: & \{(3,5),(11,13)\} \\
P_{h_7}: & \{(3,4)\} \\
P_{h_8}: & \{(1,9),(3,11),(5,13)\} \\
P_{h_9}: & \{(4,13)\} \\
P_{h_{10}}: & \{(1,11),(3,9)\} \\
P_{h_{12}}: & \{(1,13),(5,9)\} \\
P_{h_{14}}: & \{(3,13),(5,11)\} \\
P_{h_{15}}: & \{(4,11)\}
\end{array}$$

for this particular example. We could not find pairs for $h = 3, 11, 13$ which means that $h = 3, 11, 13$ do not produce pairs.

Next, we compute maximal cosets from each P_h. According to Theorem4.1.1 the algorithm returns maximal cosets for those P_h containing only one pair. Cosets from pairs $P_{h_1}, P_{h_5}, P_{h_7}, P_{h_9}$ and $P_{h_{15}}$ are

$$\begin{array}{rl}
C_1: & 4 + \langle 1 \rangle \\
C_5: & 1 + \langle 5 \rangle \\
C_7: & 3 + \langle 7 \rangle \\
C_9: & 4 + \langle 9 \rangle \\
C_{15}: & 4 + \langle 15 \rangle
\end{array}$$

respectively. The algorithm needs to run the recursion step to compute maximal cosets for the rest of P_h which contain more then one pair. We choose one among them to illustrate the procedure. Maximal cosets from the others are computed in the same fashion.

Assume the given pair is P_{h_8}, fist, we compute matrix G. The solutions for $\langle X, h_8 \rangle = 0$ are $(0100), (0010), (0001)$. There may exit other solutions, but we only need the set of linearly independent vectors.

$$G = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Take the first element from each pair, i.e, (0001),(0011),(0101). Then multiply each of them by matrix G from the right hand side, we get a new set of vectors,

$$V' = \{1, 3, 5\} = \{(001), (011), (101)\}.$$

We see that the length of vectors in V' is indeed less then the length of vectors in V. Now, we apply the MakePairs subroutine for V'. This gives following pairs.

$$\begin{array}{rl} P'_{h'_2}: & \{(1,3)\} \\ P'_{h'_4}: & \{(1,5)\} \\ P'_{h'_6}: & \{(3,5)\} \end{array}$$

The maximal cosets can easily be read from the pairs above. They are $1+\langle 2\rangle, 1+\langle 4\rangle, 3+\langle 6\rangle$. From these maximal cosets we obtain maximal cosets in V:

$$\begin{array}{rl} C_{8,1}: & 1+\langle 2,8\rangle \\ C_{8,2}: & 1+\langle 4,8\rangle \\ C_{8,3}: & 3+\langle 6,8\rangle \end{array}.$$

We use the same approach to compute cosets from pairs $P_{h_2}, P_{h_4}, P_{h_6}, P_{h_{10}}, P_{h_{12}}$ and $P_{h_{14}}$:

$$\begin{array}{rl} C_2: & 1+\langle 8,2\rangle \\ C_4: & 1+\langle 8,4\rangle \\ C_6: & 3+\langle 8,6\rangle \\ C_{10}: & 1+\langle 2,10\rangle \\ C_{12}: & 1+\langle 4,12\rangle \\ C_{14}: & 3+\langle 6,14\rangle \end{array}.$$

We see that $\{C_2, C_{10}, C_{8,1}\}$ are equal, so two of them should be removed. Other repeating cosets are $\{C_4, C_{12}, C_{8,2}\}$ and $\{C_6, C_{14}, C_{8,3}\}$.

After removing the repetition cosets, we get the maximal costes for our instance V, the list of maximal cosets are:

$$M = \{C_1, C_2, C_4, C_5, C_6, C_7, C_9, C_{15}\}. \tag{4.8}$$

4.2 Minimal Cosets Covering

From the previous section we see that all maximal cosets C_i computed from a given set V are actually subsets of V. The goal of computing a minimal coset covering is to minimize the number of maximal cosets C_i whose union covers the given set V. In this section we will introduce two different algorithms for computing minimal cosets covering. A more formal definition for the problem may be stated as follows.

Definition 4.2.1 (Minimal cosets cover) . *Let V be a set of vectors in $\mathbb{F}_q^n$ and let $M = \{C_1, C_2, \dots, C_t\}$ all maximal cosets of V. A collection of cosets $M' \subseteq M$ is a coset cover of V, if $V = \bigcup_{C\in M'} C$. Given V and M as*

input instance, the algorithm should output a set M' containing a minimal number of cosets C.

Note,there is no importance to the size of $C_i \in M'$. It is unknown whether *minimal cosets cover* problem belongs to the class of $NP-$complete problem. As we mentioned in Chapter 2 this problem is similar to the problem *minimal set cover* problem which belongs to the class of NP-complete. Therefore there is not much hope in finding a polynomial time algorithm which solves this particular problem.

4.2.1 Greedy Approximation Algorithm

In general the most natural first attempt to solve such a problem is to design a greedy algorithm. For a detailed description and analysis of the complexity of greedy algorithms see [ACGK99]. The underlying idea is simple: We make locally optimal choices, hoping that this will lead us to a optimal global solution.

First we choose the largest cosets which covers the largest amount of elements in V and remove the already covered elements from V. Then we compute maximal cosets from the remaining elements in V. This step is repeated until all elements in V are covered. It iteratively makes one greedy choice after another, reducing each given problem into a smaller one until the given set V gets empty. The procedure is summarized in Algorithm 9.

Algorithm 9: MinimalCosetSetCover1(V, M)

```
input  : A vecotr set V ⊆ F_q^n, a set of maximal cosets
         M = {C_1, C_2, ..., C_t}
output: A coset cover of V
C ← M
V' ← V
while V' ≠ ∅ do
    c ← largest coset in C
    V' ← V' \ c              // uncovered elements.
    C ←MaximalCosets(V')// cosets from the survived elements.
    M' ← Append c to M'
Return M'
```

Note, it is possible that V' is reduced to one element only. In this case the coset is the element itself (Remark 4.1.1). The cosets produced by this algorithm are disjoined, in other words, any two cosets have no elements in

common. The disjoint cosets and cosets with common elements. (cf.Section 4.2.2) have a quiet different behavior solving equation systems cf.Chapter 6.

A greedy algorithm never reconsiders its choices. Therefore, such an algorithm may not always give a optimal solution.

The algorithm first chooses a maximal sized coset C_i. Then computes maximal cosets for $V \setminus C_i$ and takes the maximal sized coset from the list, etc. The running time of the algorithm is then

$$O(2^n(T(n-1)+n^2 2^n)) = 2^{\frac{n^2}{2}(1+o(1))}.$$

4.2.2 An Exact Algorithm

As mentioned above designing an efficient algorithm for a NP-problem is considered to be a difficult job. In this section we present a not so efficient algorithm for our problem(Def.4.2.1).

Consider a set V and a set of cosets M of V. One takes any $C_i \in M$ and checks if it covers V. If true this is the minimal cover. Otherwise one takes another C_j and checks if $C_i \cup C_j$ covers the main set V. In this way one gradually constructs subsets $C_1 \cup C_2 \cup \ldots \cup C_k$ and checks the covering. The idea behind is checking the covering by k-subsets of M. Here $k-$subsets means all combinations of k elements in M.

The algorithm works as follows. For k from 1 to $|M|$ generate k-subsets and check the covering for each k. The algorithm aborts at any iteration if it finds a covering. The size of the minimal covering is then k. Assume there are t elements in M, then the number of subsets of M is

$$\sum_{k=1}^{t} \binom{t}{k},$$

This is of course the upper bound of the algorithm, i.e., $2^t - 1$, therefore exponential.

In our procedure each iteration of the algorithm only needs to process $\binom{t}{k} \approx \frac{t^k}{k!}$ subsets. In practice the minimal coset covering is found for small k, cf.Chapter 6. Therefore the algorithm is feasible for small l, i.e. small dimensional vector spaces. The suggested algorithm is depicted in Algorithm 10.

Example 4.2.1

We take the main set and the cosets from Example 4.1.1. Assume the given set V is (4.7), then the associated maximal cosets are (4.8). If k is 1 there is no 1-subset which can cover V. We also find that there is no 2-subset

Algorithm 10: MinimalCosetCover2(V, M)

input : A set V, set of maximal cosets $M = \{C_1, C_2, \ldots, C_t\}$
output: A minimum number of elements in M that covers V

```
check ← True
k ← 1
while check do
    f ← all subsets of M containing k elements
    for i = 1, i ≤ length(f) do
        if f_i ∈ f covers V then
            M' ← f_i
            check ← False
    k ← k + 1
Return M'
```

that covers V. When $k = 3$ we find that $\{C_2, C_4, C_5\}$ covers V. There may exist some other 3-subsets which cover V. The algorithm aborts and returns the solution immediately whenever it finds a solution. The cosets $\{C_2, C_4, C_5\}$ cover $\{1, 3, 9, 11\}$,$\{1, 5, 9, 13\}$ and $\{1, 4\}$ in V respectively. We see that minimal cosets covering produced by Algorithm 10 are overlapping, each of them may cover some of the same elements in the main set.

4.3 Procedure of Linearization

In this section we introduce a new linearization approach (MSLE) for multivariate polynomial equations. For a given multivariate polynomial equation, a set of linear equation systems is constructed such that union of their solutions is the the set of solutions to the multivariate polynomial. Since any set of solutions to a multivariate polynomial equation can be represented by corresponding minimal cosets covering. In the following we will describe how to construct linear equation systems from minimal cosets covering.

Consider an equation from system (1.1):

$$(X_i, V_i),$$

where $|X_i| = l$. Assume the cosets in V_i are $M = \{a_1 + B_1, a_2 + B_2, \ldots, a_t + B_t\}$, s.t.

$$V_i = \{a_1 + B_1\} \cup \{a_2 + B_2\} \cup \ldots \cup \{a_t + B_t\}.$$

We know from Section 4.2 that a minimum number of cosets can cover the main set V_i, i.e.

$$V_i = \{a_1 + B_1\} \cup \{a_2 + B_2\} \cup \ldots \cup \{a_k + B_k\}, \tag{4.9}$$

where $k \leq t$. For any $r-$dimensional coset

$$a + B = a + \langle b_1, b_2, \ldots, b_r \rangle \tag{4.10}$$

in (4.9). One solves the following equations,

$$\begin{cases} \langle b_1, y_1 \rangle = 0 \\ \langle b_2, y_2 \rangle = 0 \\ \quad\vdots \\ \langle b_r, y_r \rangle = 0. \end{cases} \tag{4.11}$$

There are $l-r$ linearly independent solutions $\{s_1, s_2, \ldots, s_{l-r} | s_i \in \mathbb{F}_q^l\}$ to the equation (4.11)(Theorem 2.3.2). These vectors can be viewed as transpose of each vector in $Null(B')$, where B' is a matrix consisted by the basis $b_1, b_2, \ldots, b_r$ as rows. A matrix C is constructed with these vectors as rows,

$$C = \begin{pmatrix} s_1 \\ s_2 \\ \vdots \\ s_{l-r} \end{pmatrix}.$$

This matrix has $l-r$ rows and l columns. By this matrix one computes a vector c,

$$Ca = c,$$

where a is a column vector.

Finally we are ready to define the linear equation system for the cosets (4.10). The linear equation system constructed from coset (4.10) is,

$$CX_i = c \tag{4.12}$$

where X_i is a column vector. We summarize the procedure of constructing linear equation systems in Algorithm 11.

Theorem 4.3.1 *Let $U = a + \langle B \rangle$ be a coset, then U are all solutions to system $CX = c$ if and only if the system is constructed from the coset by Algorithm 11.*

Proof. Assume a is a solution to $CX_i = c$, such that $Ca = c$, then $CY = \bar{0}$ if and only if

$$CY + Ca = C(Y + a) = c$$

Algorithm 11: LinearEquation

input : A coset $a + \langle b_1, b_2, \dots, b_r \rangle$ which is a subset of solutions to $f_i(X_i) = 0$

output: A linear equation system whose solutions is the coset

$B^{'} \leftarrow \begin{pmatrix} b_1 \\ \vdots \\ b_r \end{pmatrix}$ // matrix $B^{'}$ consists of b_i as rows

$C \leftarrow$ Transpose($Null(B^{'})$)

$c \leftarrow Ca$ // a is in column form

Return Linear equation system $CX_i = c$

Solutions to $CY = \bar{0}$ are a subspace of $\mathbb{F}_q^l$, i.e. the null space of C. This subspace is generated by some basis $B = \{b_1, b_2, \dots, b_r\}$. Therefore, the coset $a + B$ are all solutions to the equation $CX = c$.

Let's prove it in the opposite direction, given a r-dimensional coset $a + \langle b_1, b_2, \dots, b_r \rangle$. The basis $b_1, b_2, \dots, b_r$ are linearly independent vectors in the space $\mathbb{F}_q^l$. The equation system

$$\begin{cases} \langle b_1, y_1 \rangle = 0 \\ \langle b_2, y_2 \rangle = 0 \\ \qquad \vdots \\ \langle b_r, y_r \rangle = 0 \end{cases}$$

has $l - r$ linearly independent solutions in space $\mathbb{F}_q^l$. The solutions are rows of the matrix C. Since $Ca = c$ by the property of matrix multiplication:

$$C(a + \langle b_1, b_2, \dots, b_r \rangle) = Ca + C(\langle b_1, b_2, \dots, b_r \rangle) = c$$

We see that $C(\langle b_1, b_2, \dots, b_r \rangle) = 0$, thus the equation $CX_i = c$ holds when X_i is substituted by the coset $a + \langle b_1, b_2, \dots, b_r \rangle$. This proves the Theorem 4.3.1.

One applies Algorithm 11 to all elements in the minimal coset cover of vectors V_i from (X_i, V_i) which produces a sequence of linear equation systems

$$C_1 X_i = c_1, \qquad C_2 X_i = c_2, \qquad \dots \qquad , C_s X_i = c_s. \tag{4.13}$$

The union of solutions to these linear equation systems cover the vectors V_i. Therefore are the solutions to the equation $f_i(X_i) = 0$.

Finally we give the definition for Multiple Side Linear Equation (MSLE) systems.

Definition 4.3.1 (MSLE) [1] *MSLE is the set of linear equations:*

$$E = \{A_1X = a_1, A_2X = a_2, \ldots, A_sX = a_s\}, \tag{4.14}$$

where $|X| = l$. *Its solutions set is given by*

$$V_E = \bigcup_{i=1}^{s} \{x \in \mathbb{F}_q^l | A_i x = a_i\}. \tag{4.15}$$

With this definition, we can represent symbols (3.1) as MSLEs.

$$E = \{E_1, E_2, \ldots, E_m\} =$$

$$\{\{A_1X_1 = a_1, \ldots, A_rX_1 = a_r\}, \ldots, \{B_1X_m = a_1, \ldots, B_tX_m = b_t\}\} \tag{4.16}$$

Example 4.3.1

We will illustrate the procedure of constructing linear equation system from a given coset, such that the coset is all solutions to the linear equation system. Assume a given coset C_6 from Example 4.1.1 and a subset of variables $X = \{x_1, x_2, x_3, x_4\}$. Vectors in the coset are written in column form.

$$a + \langle b_1, b_2 \rangle \ : \ 3 + \langle 8, 6 \rangle \ = \begin{matrix} 0 \\ 0 \\ 1 \\ 1 \end{matrix} + \left\langle \begin{matrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 0 \end{matrix} \right\rangle = \begin{matrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{matrix}. \tag{4.17}$$

One solves the following equation system:

$$\begin{cases} y_1 = 0 \\ y_2 + y_3 = 0 \end{cases}.$$

The solutions to the system are $(0001), (0110)$. These are all linearly independent vectors. We obtain the matrix C by these two row-vectors:

$$C = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}.$$

[1] This term is defined relative to the term Multiple Right Hand Side (MRHS) linear equation, see [RS07]

By this matrix C and the vector a from the coset one computes a vector $c = Ca$. That yields

$$c = \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

The linear equation $CX = c$ from the coset is

$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix} X = \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

One finds that the coset (4.17) contains all solutions to the linear equation system above.

Chapter 5

Gluing on MSLEs

After the linearization techniques from the previous chapter, we are now able to represent system (1.1) as a sequence of MSLEs in the form (4.3.1). MSLEs can now be glued, i.e, by checking whether any two linear systems have common solutions. If they have common solutions one can construct a new system which has the same set of solutions. This chapter will introduce the gluing techniques on MSLE systems, thereby, solve the system (1.1). We will apply all gluing techniques described in Chapter 4 to MSLE.

5.1 Gluing A Pair MSLEs

As we see from Chapter 4 the main idea behind the Gluing Algorithms is to obtain a new symbol S_{ij} by joining two symbols S_i and S_j together, such that S_{ij} contains all common information of S_i and S_j. In this section we will apply the same idea to two MSLEs. We introduce at first how to get the common solutions to two linear equation systems.

5.1.1 Consistency

Let

$$e_1 : AX_1 = a, \qquad e_2 : BX_2 = b, \tag{5.1}$$

be two linear equation systems given. One defines a new set of variables $X = X_1 \cup X_2$ and constructs a new linear equation system as follows

$$\begin{pmatrix} A' \\ B' \end{pmatrix} X = \begin{pmatrix} a \\ b \end{pmatrix}, \tag{5.2}$$

where the part A' is the coefficient matrix of the linear equation system $AX_1 = a$ in variables X. The rows in A' represent the equations and the

columns represent the coefficients of the system $AX_1 = a$ in variables X. Analogous for $B^{'}$. $\binom{a}{b}$ is the concatenation of the two right hand sides of the above two equation systems. Note that it is important to keep the relationship between $A^{'}$ and a, $B^{'}$ and b. I.e, the right hand side can not be written $\binom{b}{a}$, if the coefficient matrix is constructed in the order $\binom{A^{'}}{B^{'}}$.
Then one transforms system (5.2) into the augmented matrix form

$$\left(\; L \mid R \; \right), \tag{5.3}$$

where L is the coefficient matrix $\binom{A^{'}}{B^{'}}$ and R represents the right hand side constant vector $\binom{a}{b}$. Then (5.3) is reduced into row echelon form (REF)[1].

$$\left(\; L^{'} \mid R^{'} \; \right), \tag{5.4}$$

Definition 5.1.1 *Let $(L^{'}|R^{'})$ be the REF of augmented matrix of two linear equation systems*

$$AX_1 = a, \qquad\qquad BX_2 = b.$$

The two equation systems are said to be ***consistent*** *if the rank of $L^{'}$ equals to the rank of $(L^{'}|R^{'})$. I.e, matrix $(L^{'}|R^{'})$ has no rows in the form*

$$[0 \qquad \dots \qquad 0 \quad | \, b],$$

where b is a non-zero entry in part $R^{'}$.

The systems of two linear equations (5.1) is consistent if they have a common solution, otherwise it is inconsistent. When the system is inconsistent, it is possible to derive a contradiction from (5.4). I.e. the statement that $0 = 1$ which implies that all coefficients of this equation are zeros. In this case, the equation is called zero equation. Every non-zero equation in (5.4) contains a basic variable with a non-zero coefficient. Either the basic variables are completely determined (with no zero-rows at the bottom of (5.4)) or at least one of the basic variables may be expressed in terms of one or more free variables (contains one or more zero rows in (5.4)). In the former case there is a unique solution to the system; In the latter case there are many solutions (one for each choice of values for the free variables).
Note that while $X_1 \cap X_2 = \emptyset$, the two systems are consistent. In this case the two systems have trivial solutions, namely all solutions to the first system and all solutions to the second system are combined together.

[1]This is elementary row operation part of Gaussian elimination algorithm, cf.Chapter 3. This operation does not exchange columns in a matrix

5.1.2 Gluing

Given two linear equation systems in the form (5.1) one checks for consistency. If they are consistent one constructs a new linear equation system as follows:

$$CX = c \leftarrow e_1 \circ e_2$$

where C is a matrix consisting of non-zero rows in $L^{'}$ and the right hand side vector c is the corresponding elements in $R^{'}$. We say that system $CX = c$ is the gluing of $AX_1 = a$ and $BX_2 = b$. The suggested algorithm for the gluing of two linear equation systems is shown in Algorithm 12.

Algorithm 12: MSLEGlue(e_1, e_2)

input : Two linear equation systmes $e_1 : AX_1 = a$, $e_2 : BX_2 = b$
output: Glued linear equation system

$X \leftarrow X_1 \cup X_j$
$D_1 \leftarrow$ AugmentedMatrix(e_1) in variables X
$D_2 \leftarrow$ AugmentedMatrix(e_2) in variables X
REF$\begin{pmatrix} D_1 \\ D_2 \end{pmatrix}$ // Row Echelon Form
if $e_1 \circ e_2$ **then**
  //Check consistency of $e_1 \circ e_2$
  Return new linear equation system $CX = c$
else
  Return $\emptyset$

One sees that the most time consuming steps of above algorithm are computing REF. This step is actually applying the elementary row operations part in Gaussian elimination algorithm. Gaussian elimination for a system of n equations in n variables requires $O(n^3)$ operations [R.W88]. Let n be the length of variables set X. The complexity of this algorithm is bounded by $O(n^3)$.

Example 5.1.1

Consider two linear equation systems in $\mathbb{F}_2$,

$$\begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix} X_1 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \qquad \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} X_2 = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \tag{5.5}$$

where $X_1 = (x_1, x_2, x_5, x_6)$, and $X_2 = (x_2, x_4, x_5, x_6)$ and the union of these two variables is $X = (x_1, x_2, x_4, x_5, x_6)$. The augmented matrix in variable

X for the above two systems is

$$\left(\begin{array}{ccccc|c} 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{array}\right). \tag{5.6}$$

Now triangulate the matrix without column exchanging. We obtain a matrix in REF as follows

$$\left(\begin{array}{ccccc|c} 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \end{array}\right). \tag{5.7}$$

This implies that the above two systems are consistent. With all non-zero rows in (5.7) and variables X one constructs the new glued system

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} X = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \end{pmatrix}. \tag{5.8}$$

Then one can solve system (5.8) which will give all common solutions to the two equation systems (5.5).
Note that in most cases it is in practice unnecessary to solve the glued equations while gluing a chain of MSLEs. The algorithm usually gives the coefficient matrix with only 1s in the diagonal entries. The solutions can easily be read from the output of the algorithm, cf. Appendix B.

Now we apply the gluing procedure on two MSLEs. Consider two MSLEs E_p and E_q in variables X_1 and X_2:

$$E_p = \begin{array}{c} e_{p_1} : A_1X_1 = a_1 \\ e_{p_2} : A_2X_1 = a_2 \\ \vdots \\ e_{p_s} : A_sX_1 = a_s \end{array}, \qquad E_q = \begin{array}{c} e_{q_1} : B_1X_2 = b_1 \\ e_{q_2} : B_2X_2 = b_2 \\ \vdots \\ e_{q_t} : B_tX_2 = b_t \end{array}. \tag{5.9}$$

One defines a set of variables $X = X_1 \cup X_2$. For each $i : 1 \leq i \leq s$ and each $j : 1 \leq j \leq t$ the consistency is checked for e_{p_i} and e_{q_j}. For any consistent i, j one constructs a new system $C_{ij}X = c_{ij}$ by gluing e_{p_i} and e_{q_j}. If they are inconsistent discards the gluing step and try to glue e_{p_i} and $e_{q_{j+1}}$. We summarize the pairwise gluing procedure in Algorithm 13.

Algorithm 13: MSLEGluePair(E_p, E_q)

input : Two MSLEs $E_p = \{e_{p1}, e_{p2}, \ldots, e_{ps}\}$, $E_q = \{e_{q1}, e_{q2}, \ldots, e_{qt}\}$
output: A Glued MSLE

$E^{'} \leftarrow \emptyset$
for *i from* 1 *to s* **do**
    for *j form* 1 *to t* **do**
        if e_{pi}, e_{qj} *are consistent* **then**
            $E^{'}$ ←Append $e_{pi} \circ e_{qj}$ to $E^{'}$

Return $E^{'}$

Let $|E_i|$ be the number of linear equation systems in each MSLE, then the running time of above procedure is:

$$O(|E_q||E_p|) \tag{5.10}$$

REF computations on linear equation systems in variables of length $|X|$ over $\mathbb{F}_q$.

5.2 Gluing A List of MSLE

The procedure of gluing a list of MSLEs is similar to Algorithm 2 which we described in Chapter 3. In order to solve a sequence of MSLEs one applies Algorithm 13 to the first pair of MSLEs. In the next step the algorithm proceeds with the result of the previous step as the first argument and the next MSLE as the second argument, etc. The procedure is shown in Algorithm 14.

Algorithm 14: MSLEGlueSet(E)

input : A list of MSLE $E_1, E_2, \ldots, E_m$
output: A Glued MSLE

$m \leftarrow length(\mathrm{E})$
$Z \leftarrow E_1$
$k \leftarrow 2$
while $k \leq m$ **do**
    $Z \leftarrow$ MSLEGluePair(Z, E_k)
    $k \leftarrow k + 1$
Return Z

The running time of this algorithm is roughly:

$$O(\sum_{k=1}^{m-1} |Z_k| + 1) \tag{5.11}$$

REF computations on linear systems in variable $X(k) = X_1 \cup \ldots \cup X_k$ over $\mathbb{F}_q$, where Z_k is the set of linear equation systems in the MSLE after gluing the first k MSLEs. The mathematical expectation of $|Z_k|$ has not been proven by the time this report is finished.

5.3 MSLE-Gluing

Applying the tree search gluing procedure on MSLEs is called MSLE-Gluing. This procedure is quiet similar to the Algorithm 3 (Gluing1 in [Sem08]) described in Chapter 3. It has the same running time as (5.11), but requires only `poly` bits of memory space. We will not describe the procedure here. For details and an example of the procedure one may consult Section 3.1.3 in Chapter 3. Note, this algorithm only output one MSLE while the system may has more.

A slightly modified algorithm which can be used on gluing of MSLEs is presented here, see Algorithm 15. In this algorithm e_{s_d} stands for a linear equation system in MSLE at tree depth d. One may analog each e_i as the vector $v_i \in V_i$ from (X_i, V_i). The algorithm terminates in two cases:

1. It traverses all paths from the root to the bottom. I.e. all possible solutions to the system are returned.
2. No path through the search tree is found. I.e. d becomes smaller than 1.

Note, Algorithm 15 only gives one MLSE for the final output. One could modify this algorithm by the method mentioned in Chapter 3 to output all solutions. It should be emphasized that one might want to apply Algorithm 15 to MSLEs which were constructed from cosets generated by Algorithm 10. As we mentioned before, the cosets generated by Algorithm 10 are intersecting each other. For this reason the number of intermediate solutions will increase dramatically. We will discuss this and try to reduce the number of intermediate solutions in the next section.

5.4 Implied System

The number of cosets generated by Algorithm 10 always is less than or equal the number of cosets generated by the Greedy algorithm. Algorithm 10 al-

Algorithm 15: MSLE-Gluing(E)

```
input  : A list of MSLE: E_1, E_2, ..., E_m
output: A glued MSLE

d ← 1
s_1 = 1, ..., s_m = 1   // Initialize all s_i
M_1 ..., M_m ← Initialize all M_i as empty
while 0 < d ≤ m do
    T ← MSLEGlue(M_d, e_{s_d} ∈ E_d)       // Calls MSLEGlue subroutine.
    if T ≠ ∅ then
        M_{d+1} ← T
        s_d ← s_d + 1
        d ← d + 1
    else
        s_d ← s_d + 1  //points to the next e at the same tree depth
        if s_d > length(E_d) then
            // tried ∀e ∈ E_d
            s_d ← 1
            d ← d − 1
if d = m then
    Return M_d
else
    Return Ungluable
```

ways gives the exact number of cosets for minimal covering while the Greedy algorithm returns the approximal number of cosets. One might attempt to solve system (1.1) by MSLEs which is constructed by cosets output by Algorithm 10, hoping that less gluing operations occur during the solving. However, in practice we found that a lot of gluings reoccur. This implies reiterative intermediate solutions are produced. In this section we will analysis this phenomena and describe an algorithm which can slightly reduce the number of gluing steps while solving.

5.4.1 Implication

Assume cosets for the two MSLEs (5.9) are generated by Algoritm 10. Let us define the solution set of $e_{p_i} \circ e_{q_j}$ by $\Omega_{i,j}$ for $1 \leq i \leq s$ and $1 \leq j \leq t$. Then the following two cases are possible:

C1: $\Omega_{i,j} \subseteq \Omega_{i,j1}$;

C2: $\Omega_{i,j} \supseteq \Omega_{i,j1}$.

This is because of there may exist some overlapping subsets in the joined set of solutions to e_{q_u} and e_{q_v} for some $u, v : 1 \leq u, v \leq t, u \neq v$. It might therefore cause that

$$e_{p_i} \circ e_{q_j} \circ e_{w_k} = e_{p_i} \circ e_{q_{j1}} \circ e_{w_k},$$

where e_{w_k} is a linear equation system in the MSLE next to E_q. They obviously share some common solutions. Considering the tree search algorithm, the repeating solutions mean that the algorithm produces repeated nodes which have the same value. This causes that the search tree contains many repeated subtrees.

Two MSLEs are given in the form (5.9), then we say that $e_{p_i} \circ e_{q_j}$ **implies** $e_{p_i} \circ e_{q_{j1}}$, if their solution sets $\Omega_{i,j}$ and $\Omega_{i,j1}$ are in the case C1. $e_{p_i} \circ e_{q_{j1}}$ **implies** $e_{p_i} \circ e_{q_j}$ if their solution sets are in the case C2.

Here A implies B means that all solutions to A are solutions to B. Since B contains all solutions to A, we can simply omit the gluing operation of A while solving a system. Likewise, gluing operations on B can be discarded if B implies A.

5.4.2 Procedure of Reduction

To check the implication relationship for (5.9) one constructs an augmented matrix in the form (5.3) in variables $X = X_1 \cup X_2$:

$$\left(\begin{array}{c|c} A_1 & a_1 \\ B_1 & b_1 \\ \hline B_2 & b_2 \end{array}\right). \tag{5.12}$$

The above matrix has $|X|+1$ columns and the number of rows is the number of all equations in the three systems. We divide the above matrix into two parts by the horizonal line. The upper part consists of the augmented matrix of e_{p_1} and e_{q_1} in variable X, which denoted by P_1. The lower part is the augmented matrix e_{q_2} in variables X, denoted by P_2. One applies row reduction (reduced into REF) operations to (5.12) with the following restrictions:

$\mathbf{U}_1$: Columns in (5.12) are not allowed to exchange;

$\mathbf{U}_2$: Rows in P_2 are not allowed to exchange with rows in P_1.

Note that any two rows inside P_1 and P_2 can only be interchanged in their partition.

Theorem 5.4.1 *Let three linear equation systems in variables X_1 and X_2:*

$$e_1 : AX_1 = a, \qquad e_2 : B_1X_2 = b_1, \qquad e_3 : B_2X_2 = b_2 \tag{5.13}$$

be given. Let us define

$$\begin{pmatrix} P_1 \\ P_2 \end{pmatrix} \tag{5.14}$$

as the augmented matrix of the three systems in (5.13) in variable $X = X_1 \cup X_2$*.* P_1 *is the augmented matrix of* e_1 *and* e_2*, and* P_2 *is augmented matrix of* e_3*. The REF of (5.14) constructed under the restrictions* $\boldsymbol{U}_1$ *and* $\boldsymbol{U}_2$ *is defined by*

$$\begin{pmatrix} P_1^{'} \\ P_2^{'} \end{pmatrix} \tag{5.15}$$

such that, $e_1 \circ e_2$ *implies* $e_1 \circ e_3$ *if and only if* $P_2^{'}$ *is a zero-matrix.*

Lemma 5.4.1 *Let two linear equation systems*

$$C_1X = c_1 \quad \textit{and} \quad C_2X = c_2$$

be given. Solutions of $C_1X = c_1$ *are among the solutions to* $C_2X = c_2$*, if and only if there exits a matrix* U*, such that* $UC_1 = C_2$ *and* $Uc_1 = c_2$*.*

Proof (Lemma 5.4.1), Let $b_1 + \langle B_1 \rangle$ be the solutions to $C_1X = c_1$ and $b_2 + \langle B_2 \rangle$ the solutions to $C_2X = c_2$. Assume $b_1 + \langle B_1 \rangle \subseteq b_2 + \langle B_2 \rangle$. One can take $b_1 = b_2$. We have $C_1B_1 = \bar{0}$ and rows of C_1 generate null-space of B_1. Then $C_2B_1 = \bar{0}$, that is rows of C_2 are in the null-space of B_1. Therefore rows of C_2 are combinations rows of C_1, there exits a matrix U such that $UC_1 = C_2$. We have $C_1b_1 = c_1$ and $C_2b_1 = c_2$. Then $c_2 = C_2b_1 = UC_1b_1 = Uc_1$. This proves the Lemma 5.4.1.
Proof (Theorem 5.4.1), Assume $e_1 \circ e_2$ implies $e_1 \circ e_3$, then we have

$$\begin{pmatrix} A \\ B_1 \end{pmatrix} X = \begin{pmatrix} a \\ b_1 \end{pmatrix} \textit{ implies } \begin{pmatrix} A \\ B_2 \end{pmatrix} X = \begin{pmatrix} a \\ b_2 \end{pmatrix}.$$

By Lemma 5.4.1 there exits a matrix $U = (U_1|U_2)$, such that

$$\begin{pmatrix} I \mid \mathbf{0} \\ U_1|U_2 \end{pmatrix} \begin{pmatrix} A \\ B_1 \end{pmatrix} = \begin{pmatrix} A \\ B_2 \end{pmatrix}, \quad \begin{pmatrix} I \mid \mathbf{0} \\ U_1|U_2 \end{pmatrix} \begin{pmatrix} a \\ b_1 \end{pmatrix} = \begin{pmatrix} a \\ b_2 \end{pmatrix}, \tag{5.16}$$

Where I is the identity matrix and $\mathbf{0}$ denotes the zero matrix. We obtain $U_1A + U_2B_1 = B_2$ and $U_1a + U_2b_1 = b_2$, where a and b_i are column vectors.

While applying the REF computation for (5.14) in $\mathbb{F}_q$, we get

$$U_1A + U_2B_1 + B_2 = \mathbf{0}, \quad U_1a + U_2b_1 + b_2 = \bar{0} \tag{5.17}$$

This implies that all rows in the lower part of the REF matrix (5.15) are zero-row.

Assume P' in (5.14) is a zero matrix, then there exits a matrix $U = U_1|U_2$, such that the two equations (5.17) are true. One constructs the two equations (5.16). This proves Theorem 5.4.1.

One applies the implication steps before gluing two systems, such that it discards gluing step A as long as A implies B. We insert the implication procedure in Algorithm 15 which is shown in Algorithm 16. Note, this algorithm only output one MSLE while the system may has more.

Algorithm 16: MSLE-GluingReduction(E)

```
input  : A list of MSLE: E_1, E_2, ..., E_m
output: A glued MSLE

d ← 1
s_1 = 1, ..., s_m = 1   // Initialize all s_i
M_1, ..., M_m ← Initialize all M_i as empty
while 0 < d ≤ m do
    check ←False
    if M_d ∘ e_{s_d} ∈ E_d implies M_d ∘ e_{s_{d+1}} ∈ E_d then
        T ← MSLEGlue(M_d, e_{s_{d+1}} ∈ E_d)
        check ← True
    else
        T ← MSLEGlue (M_d, e_{s_d} ∈ E_d)
    if T ≠ ∅ then
        M_{d+1} ← T
        if check then
            s_d ← s_d + 2 // points to e_{s_d+2}
        d ← d + 1
    else
        s_d ← s_d + 1// points to the next system at the same tree depth
        if s_d > length(E_d) then
            // tried ∀e ∈ E_d
            s_d ← 1
            d ← d − 1
if d = m then
    Return M_d
else
    Return Ungluable
```

5.5 Edge Removal

While utilizing Algorithm 15 to solve system (1.1) one can see that the solution can only be revealed from a path which goes through the whole search tree, i.e, from the root to the bottom. However, there are many tree branches which do not reach the bottom, that means these tree branches do not give any contribution to the solution set. Nevertheless, the algorithm walks through all these irrelevant tree branches. In order the algorithm to avoid going through these tree branches, we apply a so called edge removing procedure on Algorithm 15.

Consider a given list of MSLEs in the form (4.16). Then one defines a search tree rooted at $\emptyset$. At each tree depth $d : 1 \leq d \leq m$ one defines a MSLE $E_1 \circ \ldots \circ E_d$. One can see that there is a tree edge between two nodes at tree depth d and $d+1$ if a linear equation system $e_1 \circ \ldots \circ e_d \in E_1 \circ \ldots \circ E_d$ and another linear equation system $e_j \in E_j : \ j \geq d+1$ exists such that $e_1 \circ \ldots \circ e_d \circ e_j$ is possible. A tree branch $e_1 \circ e_2 \circ \ldots \circ e_m \in E_1 \circ E_2 \circ \ldots \circ E_m$ gives a solution to the whole system. One checks the connectivity of a tree branch and lets the algorithm avoid going through the tree branches if and only if the tree branch does not reach to the bottom.

Let $e_i \in E_1$. If at any tree depth $d \geq 2$, the linear equation system e_i does not consistent with all $e_j \in E_d$, the gluing operation will not take place on this tree branch. Otherwise, for each $d \geq 3$ there is at least one $e_k \in E_d$ such that $e_i \circ e_j$ is consistent with e_k, then one extends the tree branch as $e_i \circ e_j \circ e_k$, where $e_k \in E_3$. If they are inconsistent, one extends the branch by gluing e_i with another system in E_2, etc.

Note, in the case $X_d(E_d) \cap X_{d+1}(e_{d+1}) = \emptyset$, one does not need to check the consistency of systems in these two MSLEs. As we mentioned in Section 5.1.1, two systems with no variables in common are always consistent. We summarize the procedure in Algorithm 17. Note, this algorithm only output one MSLE while the system may has more.

Algorithm 17: MSLE-GluingEdgeRemoval(E)

```
input  : A list of MSLE: E_1, E_2, ..., E_m
output: A glued MSLE

d ← 1
s_1 = 1, ..., s_m = 1   // Initialize all s_i
M_1, ..., M_m ← Initialize all M_i as empty
while 0 < d ≤ m do
    k ← d
    while 2 ≤ k ≤ m do
        for ∀e_i ∈ E_k do
            if X_d(M_d) ∩ X_k(E_k) ≠ ∅ then
                if M_d dose not consistent with e_i then
                    d ← d − 1
                    Break While loop
                else
                    k ← k + 1
    T ← MSLEGlue(M_d, e_{s_d} ∈ E_d)       // Calls MSLEGlue subroutine.
    if T ≠ ∅ then
        M_{d+1} ← T
        s_d ← s_d + 1
        d ← d + 1
    else
        s_d ← s_d + 1// points to the next system at the same tree depth
        if s_d > length(E_d) then
            s_d ← 1
            d ← d − 1
if d = m then
    Return M_d
else
    Return Ungluable
```

Chapter 6

Experimental Results

In this chapter the experimental results are presented. We have implemented all other algorithms mentioned in this report except Algorithm 2 and Algorithm 14. Since Algorithm 2 and Algorithm 14 give the same results as the tree search version, but require more memory. We discarded the testing on these two algorithms.

All equations in (1.1) are independently generated over field $\mathbb{F}_2$. Each $f_i(X_i) = 0$ is represented by a symbol (X_i, V_i). The subset X_i of size l is taken at random from the set of all possible $l-$subsets of X, that is with probability of $\binom{n}{l}^{-1}$. All vectors in the vector subset V_i are randomly taken from the vector space $\mathbb{F}_2^l$.

We tested four different parameters in each example of our experiments:

- n the number of variables for the whole system;
- m the number of equations in the system;
- l the sparsity or number of variables in each equation;
- $|V|$ the number of assignments for each equation.

The abscissa indicates the tree depth value. On the ordinate the number of Gluing1 (or MSLE-Gluing) operations at each tree depth is plotted. The number of Gluing1 operations is denoted by $t_{Gluing1}$ and the number of MSLE-Gluing operations is denoted by $t_{MSLE-Gluing}$. In the following diagrams all t are logarithmic scaled.

For the reference Mathematica-code refer to Appendix A. For a description of experimental environments refer to Appendix B. A sample raw data from the experiments is included in Appendix C.

6.1 Cosets Covering

Figure 6.1 and Figure 6.2 show the comparisons of the minimal cosets covering computed by Algorithm 9(Greedy) and Algorithm 10(Exact). On the ordinate the number of cosets for each $V_i(X_i, V_i)$ and on the abscissa the number of symbols in the whole system are plotted.

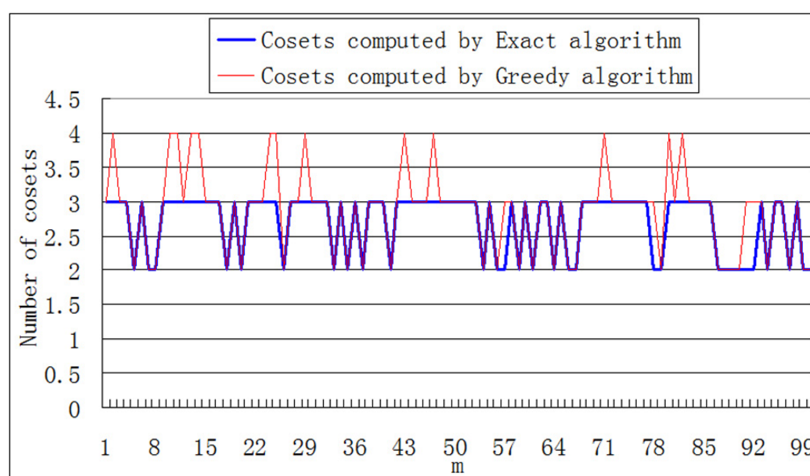

Figure 6.1: Comparison of minimal coset covering for $n = 4,\ m = 100, l = 4,\ |V| \geq 5$. The average minimal cosets computed by Greedy algorithm is 2.7. The average minimal cosets computed by Exact algorithm is 2.87. Approximation ratio of the Greedy algorithm is 0.94.

6.2 Sorting

In order to reduce the gluing operations we applied the sorting procedure on the instances. Figure 6.3 shows the comparisons of the number of Gluing1 and MSLE-Gluing operations occurred at each tree depth. The input to the algorithms are sorted and unsorted instances.

6.3 Comparison of Gluing1 and MSLE-Gluing

The Figure 6.4, Figure 6.5 and Figure 6.6 show the comparisons of the number of gluing operations needed by the Gluing1 and the MSLE-Gluing to solve the same system.

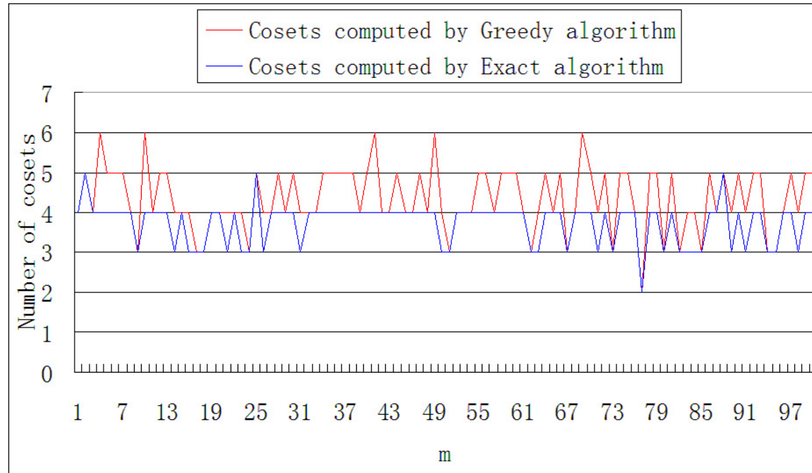

Figure 6.2: Comparison of minimal coset covering for $n = 5$, $m = 100$, $l = 5$ $|V| \geq 8$. The average minimal cosets computed by Greedy algorithm is 4.34. The average minimal cosets computed by Exact algorithm is 3.74. Approximation ratio of the Greedy algorithm is 0.87

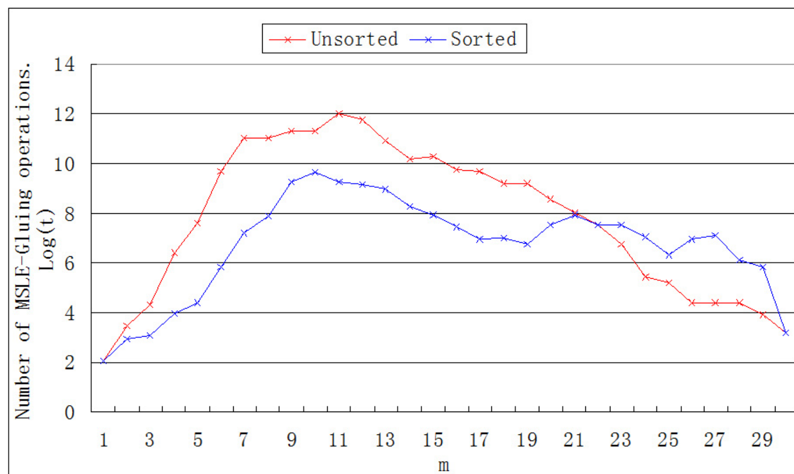

Figure 6.3: MSLE-Gluing operations on unsorted and sorted instance. $n = 32$, $m = 32$, $l = 4$, $|V| \geq 5$. Cosets are computed by Greedy algorithm. Max($t_{unsorted} = 164064$), Max($t_{sorted} = 15784$). $\frac{Max(t_{unsorted})}{Max(t_{sorted})} \approx 10$

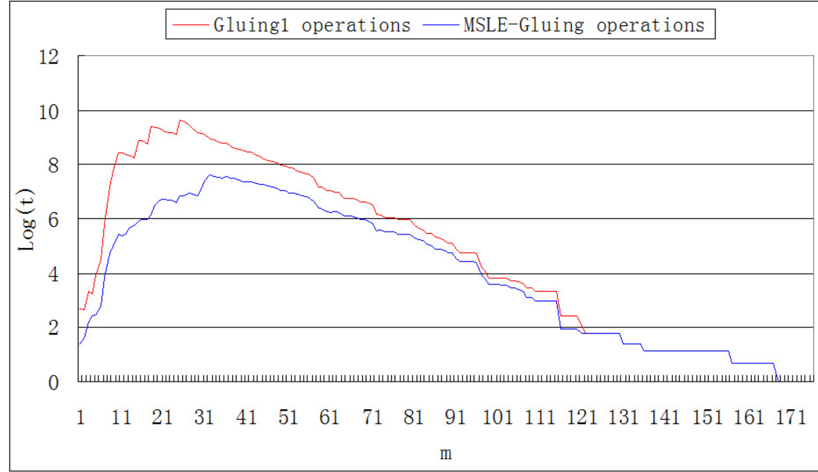

Figure 6.4: Comparison of the number of Gluing1 and MSLE-Gluing operations at each tree depth. $n = 16$, $m = 178$, $l = 4$, $|V| = 15$. Equations are sorted. Cosets are computed by Greedy algorithm. Max($t_{Gluing1} = 15024$), Max($t_{MSLE-Gluing} = 2046$). $\frac{Max(t_{Gluing1})}{Max(t_{MSLE-Gluing})} \approx 7$

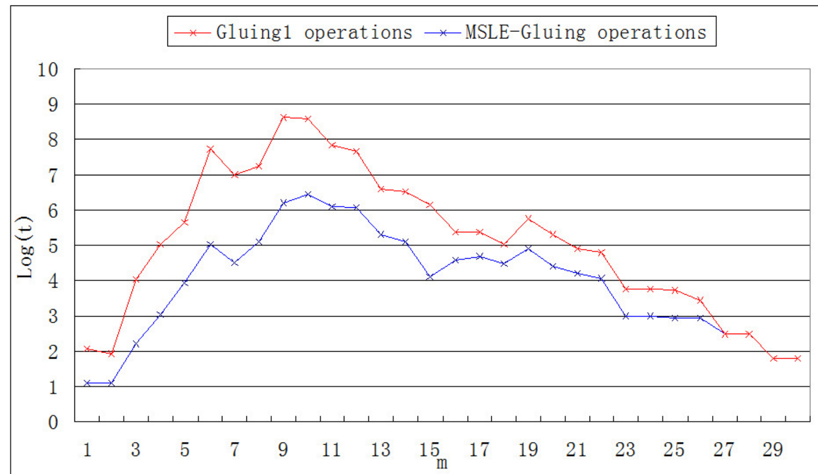

Figure 6.5: Comparison of the number of Gluing1 and MSLE-Gluing operations at each tree depth. $n = 32$, $m = 32$, $l = 4$, $|V| \geq 5$. Equations are sorted. Cosets are computed by Greedy algorithm. Max($t_{Gluing1} = 5600$), Max($t_{MSLE-Gluing} = 627$). $\frac{Max(t_{Gluing1})}{Max(t_{MSLE-Gluing})} \approx 9$

6.4 Reducing of Gluing operations

Figure 6.7 shows the comparisons of the number of MSLE-Gluing operations occurred at each tree level, while MSLE-Gluing applied on MSLEs which are constructed from cosets computed by Exact algorithm and Greedy algorithm. The Algorithm 16 was applied on MSLEs constructed from cosets which are computed by the Exact algorithm.

6.5 Edge Removing

Figure 6.8 shows the gluing operations occurred at each tree level after applying edge removing procedure on Gluing1 and MSLE-Gluing. Equations are sorted. Cosets are computed by the Greedy algorithm. Max($t_{Gluing1}$ = 3258304), Max($t_{MSLE-Gluing}$ = 63648). $\frac{Max(t_{Gluing1})}{Max(t_{MSLE-Gluing})} \approx 51$. The number of gluing operations after edge removing is denoted by t'. Figure 6.9 shows the comparison of the number of MSLE-Gluing operations at each tree level after applying edge removing procedure on MSLE-Gluing.

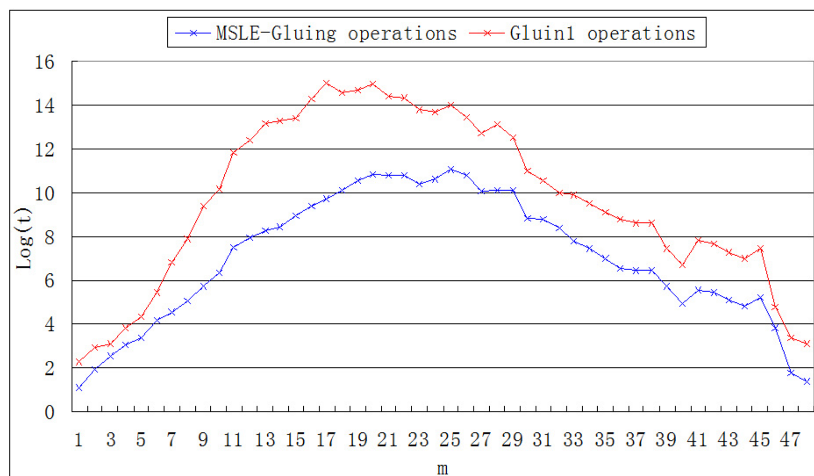

Figure 6.6: Comparison of the number of Gluing1 and MSLE-Gluing operations at each tree depth. $n = 48$, $m = 48$, $l = 4$, $|V| \geq 5$. Equations are sorted. Cosets are computed by Greedy algorithm. Max($t_{Gluing1}$ = 3258304), Max($t_{MSLE-Gluing}$ = 63648). $\frac{Max(t_{Gluing1})}{Max(t_{MSLE-Gluing})} \approx 51$

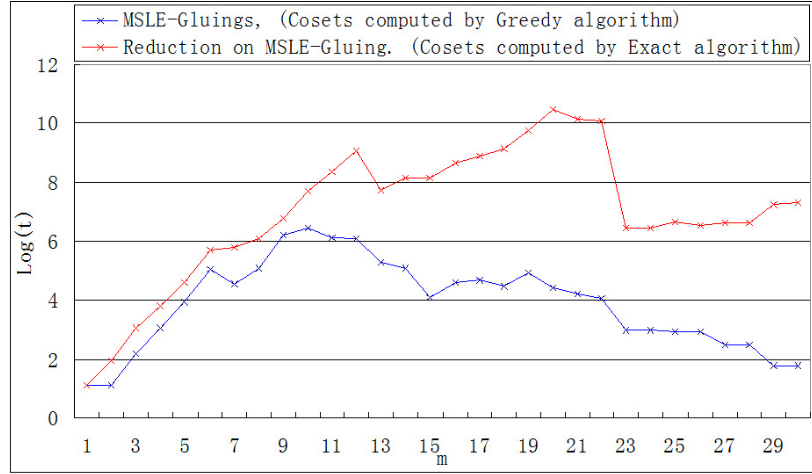

Figure 6.7: Reduction of gluing operations on the MSLE-Gluing. $n = 32$, $m = 32$, $l = 4$, $|V| \geq 5$. Equations are sorted. Cosets are computed by Exact algorithm and Greedy algorithm for the same instance. Max($t_{Exact} = 35248$), Max($t_{Greedy} = 627$). $\frac{Max(t_{Exact})}{Max(t_{Greedy})} \approx 56$

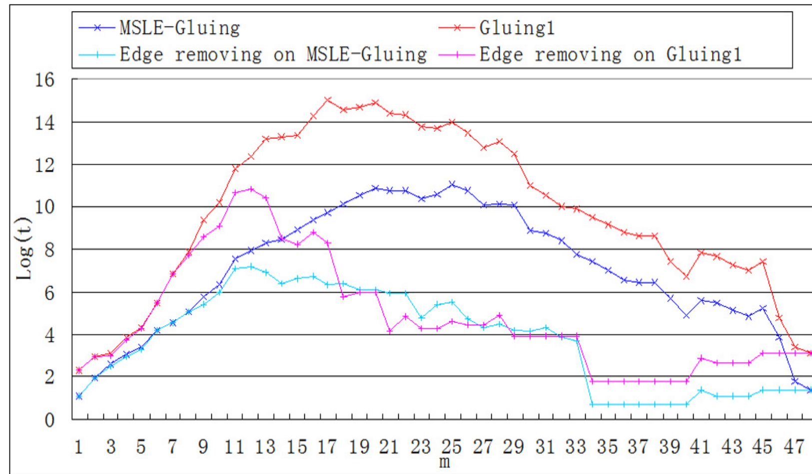

Figure 6.8: Edge removing procedure on MSLE-Gluing and Gluing1. $n = 48$, $m = 48$, $l = 4$, $|V| \geq 5$. Cosets are computed by Greedy algorithm Max($t'_{Gluing1}$) = 49464, Max($t'_{MSLE-Gluing}$) = 1349. $\frac{Max(t_{Gluing1})}{Max(t'_{Gluing1})} \approx 66$. $\frac{Max(t_{MSLE-Gluing})}{Max(t'_{MSLE-Gluing})} \approx 47$

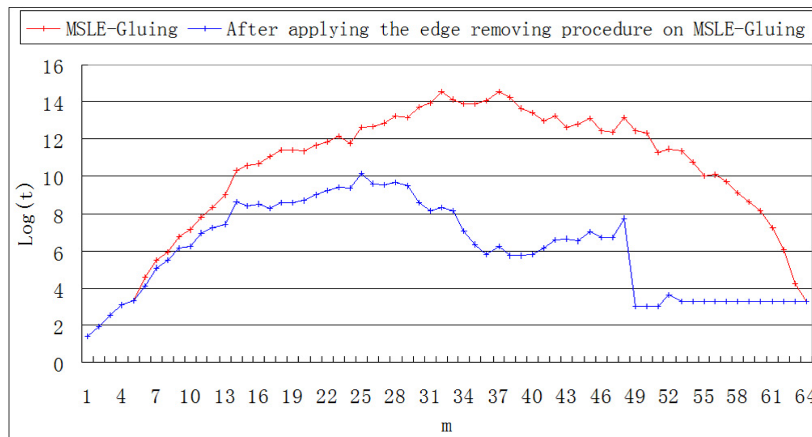

Figure 6.9: Edge removing procedure on MSLE-Gluing.
$n = 64,\ m = 64,\ l = 4,\ |V| \geq 5$. Cosets are computed by Greedy algorithm.
$\mathrm{Max}(t_{MSLE-Gluing}) = 2100252$, $\mathrm{Max}(t'_{MSLE-Gluing}) = 24866$. $\frac{2100252}{24866} \approx 84$
.

Chapter 7

Discussion

In this chapter we will first discuss the result in Section 6.5. Afterwards we will present our conclusion. Finally we will discuss the possible further work.

7.1 Discussion

One can see in Figure 6.7 that the trend of curve generated by Algorithm 16 rises at the tail. Recall that the MSLEs for this algorithm are generated by Algorithm 10. The cosets computed by this algorithm are intersecting. For this reason we have introduced implication relations and tried to reduce the gluing operations. We have implemented reduction procedure only on the "right hand side".

Consider the two MSLEs (5.9) in Section 5.1.2. The reduction algorithm discards the gluing operation $e_{p_i} \circ e_{q_j}$ if it implies $e_{p_i} \circ e_{q_{j+1}}$, where $i, j: 1 \leq i \leq s,\ 1 \leq j \leq t$. It is also possible that $e_{p_u} \circ e_{q_v}$ implies $e_{p_{u+1}} \circ e_{q_v}$, where $u, v: \ 1 \leq u \leq s,\ 1 \leq v \leq t$. Thus the algorithm should discard the gluing operation step $e_{p_u} \circ e_{q_v}$. However, the algorithm does not include this procedure. This generates repeated intermediate solutions.

7.2 Conclusion

In this thesis we have presented two approaches to solve system (1.1). The first approach are the Gluing/Agreeing algorithms which have been studied for several years. In order to compare the solving behaviors with the MSLE approach we did not combine the Agreeing algorithm with it. The second approach is the MSLE which is quite new.

The main advantages of MSLE when compare to the Gluing algorithms on symbols appear to be the fact that, any $f(X) = 0$ of bounded number of

variables can be represented by the MSLE in a finite field. The cosets are represented by linear equation systems. Therefore, the MSLE representation is much more compact than the coset itself and easy to implement in linear algebra routines. The number of linear equation systems in a MSLE is much lower than the number of assignments in the corresponding symbol (X, V) ; While considering experimental results in Chapter 6, one sees that comparing to the Gluing1 algorithm the number of gluing operations needed by MSLE-Gluing to solve the same system is lower.

In order to enhance the algorithm we have introduced edge removing procedure on Gluing1 algorithm and MSLE-Gluing algorithm. This procedure gives a significant reduction on the number of gluing operations.

Furthermore, in Section 6.7 we implemented reduction procedure on the MSLE-Gluing algorithm where the MSLEs are constructed by Algorithm 10 (exact algorithm). However, the results are worse then our expectation. This mainly due to the problem we mentioned in Section 7.1. Nevertheless the greedy procedure gives a good approximation for computing minimal coset covering. Therefore it is not necessary to pay too much effort on improving this algorithm.

An interesting future research topic might be implement the MSLE-Gluing algorithm and related procedures in a higher level programming language and compare the solving behavior with other solving techniques like SAT-solvers.

Appendix A

Algorithms Written in Mathematica

This chapter contains routines to run all algorithms we described in previous chapters. We implemented the algorithms in the Mathematica7.0 [Wol92].

In the following context we describe the usage of routines which are used to obtain the experimental results in Chapter6.

- `RandomInst[n, m, l, f]` returns randomly generated symbols correspondent to the system(1.1). Each input argument is represented as
  - n: number of variables in the whole system, i.e, $|X|$;
  - m: number of equations in the system;
  - l: the sparsity, that is the number of variables in each equation;
  - f: length of each $V(S)$, $f \leq |V| \leq 2^l - 1$.
- `Glue[in1, in2]` returns a glued symbol $v_i \circ v_j$, where $v_i \in V_i(S_k), v_j \in V_s(S_s)$. If the two symbols are not gluable it returns an empty set $\{\}$;
- `TreeGlue[in]` returns two arguments, i.e., all solutions to system (1.1) and the number of intermediate solutions while the algorithm solves the system; if the system has no solution it returns **NO Solutions**.
- `MSLEGreedy[in]` gives a list of MSLEs for the whole system. The cosets for the MSLEs are generated by the Greedy algorithm described in Chapter4. The input to this routine is a list of symbols obtained from function `RandomInst[n,m,l,f ]`.
- `MSLEExact[in]` gives list of MSLEs for the whole system. The cosets for the MSLEs are generated by the Exact algorithm described in Chapter4. The input to this routine is a list of symbols obtained from function `RandomInst[n,m,l,f ]`.

- `MSLEGlue[in1, in2]` returns a glued MSLE $e_i \circ e_j$, where $e_i \in E_k$ and $e_j \in E_s$. If the two equations are not glueable it returns an empty set $\{\}$;
- `MSLETreeGlue[in]` returns all solutions to system (1.1) which are represented in MSLE-form. It also returns the intermediate solutions while the algorithm is solving the system. If no solutions are found it returns **No Solutions**.
- `MSLETreeGlueER[in, z]`, this routine cuts all unnecessary branches in the search tree. It returns all solutions and the number of intermediate solutions. If the system has no solution no gluing operation is performed by Algorithm 17 (This is the same as applying the cutting branch procedure on Algorithm 3). In order to get all intermediate solutions we set an argument `z` for this routine. It is determined by Algorithm 15, that is the number of unreached tree levels by Algorithm 15.
- `TreeGlueER[in, z]`, the behaviour is similar to routine `MSLETreeGlueER[in, z]`. The input for this routine is a set of symbols.
- `MSLETreeGlueReduction[in]`, gives all solutions to system (1.1) which are represented in MSLE-form. It also gives reduced intermediate MSLEs. The input to this routine is a list of MSLEs, cosets for which are computed by the Exact algorithm.

A.1 Auxiliary Routines

```
-----------------------------------------------------------------------------------------

1  symbolForm[in_] :=
2           Module[{solu, var, a},
3          (*convert random equations in symbolform (X,V)*)
4           solu := Table[Transpose[in[[i]]], {i, Length[in]}];
5           var := solu[[1, 1]];(*variables in f (X)==0*)
6           a := Table[solu[[i, 2]], {i, Length[solu]}];
7           Return[{var, a}] ]
8
9
10 pairEQ[in1_, in2_] :=
11        Module[{},
12       (*checks two pairs (a,a+h)and (b,b+h) are equal,
13        rerurn True or False*)
14        If[Sort[in1] == Sort[in2], True, False] ]
15
16
17 BaseExpand[in_] :=
18          (*input one coset in the form {v,b1,b2,..bn} give the setform of coset,
```

```
19          v+<b1,b2>---> {v+b1,v+b2, v+(b1+b2),..,v}, updated 2009-9-1*)
20         Module[{i, j, B, v = First[in], b = Rest[in]},
21         B = Subsets[b];
22         For[i = 1, i <= Length[B], i++,
23                     If[Length[B[[i]]] > 1,B[[i]] = {Apply[BitXor, B[[i]]]}];
24                     B[[1]] = Table[0, {Length[Flatten[B[[2]]]]}];
25            ];
26         For[j = 1, j <= Length[B], j++,
27                      B[[j]] = BitXor[Flatten[B[[j]], 1], v]
28            ];
29         Return[Sort[B]]
30         ]
31
32
33 GlueQ[in1_, in2_] :=
34      Module[{},
35      If[in1 === empty, True,
36         If[Glue[{in1}, {in2}] =!= {}, True, False]]]
37
38
39 ConsistenceQ[a_, c_] := Module[{L},
40             (*check consistency of two linear equations system*)
41             Off[LinearSolve::"nosol"];
42             L = LinearSolve[a, c, Modulus -> 2];
43             !Head[L] === LinearSolve]
44
45
46 ConsistencyQ[in1_, in2_] := Module[
47             {X, c, g, T, gt, coeMat1, coeMat2, augment1, augment2, C1, newlin},
48             (*Latest update 2009-9-4*)
49              If[in1 == null, Return[T = 1]];
50              If[in1 == {} || in2 == {} || ! ListQ[in2], Return[T = 0]];
51              X = Union[in1[[1]], in2[[1]]];(*variable X*)
52              coeMat1 = Map[Coefficient[#, X] &, Flatten[in1[[2]]]];
53              coeMat2 = Map[Coefficient[#, X] &, Flatten[in2[[2]]]];
54              (*Coefficient matrix for in1 and in2 in variable X*)
55              augment1 = Transpose[Join[Transpose[coeMat1], in1[[3]]]];
56              augment2 = Transpose[Join[Transpose[coeMat2], in2[[3]]]];
57              (*augmented matrix for in1 and in2*)
58              If[ConsistenceQ[Join[coeMat1, coeMat2],
59                  Flatten[Join[in1[[3]], in2[[3]]], 1]],
60              Return[T = 1], Return[T = 0]]]
61
62
63 Triangulate[A0_] :=
64             Module[{A = A0, n = Length[A0], p, k, m = Length[A0[[1]]], i, j, q, M, T, t},
65             l = Min[m, n];
66             A = Reverse[Sort[A]];
67             For[i = 1, i <= l, i++,
68                  For[j = i + 1, j <= n, j++,
69                      If[(A[[i, i]] == 1 && A[[j, i]] == 1),
70                         A[[j]] = BitXor[A[[i]], A[[j]]]]
71                  ]
72             ];
73             A = Reverse[Sort[A]];
74             Return[A]]
75
76
77 Rowreduction[A0_, C0_] :=
78             Module[{A = A0, C1 = C0, l = Length[A0[[1]]], M, i, j, q, t, T, ul, ur, rr, rl,
79                     r, k, n = Length[A0], m = Length[C0]},
80             q = Min[l, n + m];
```

```
81              A = Triangulate[A];
82              C1 = Triangulate[C1];
83              M = Join[A, C1];
84              For[i = 1, i < q, i++,
85                  For[j = i + 1, j <= n + m, j++,
86                     If[(M[[i,i]]==1&&M[[j,i]]==1)||
87                       (M[[i,i]]==0&&M[[i,i+1]]==1&&M[[j,i+1]]==1),
88                        M[[j]] = BitXor[M[[i]], M[[j]]]
89                     ]
90                  ]
91              ];
92              sort = Reverse[Sort[Take[M, n]]];
93              Table[M[[i]] = sort[[i]], {i, n}];
94              sort = Reverse[Sort[Take[M, -m]]];
95              Table[M[[n + i]] = sort[[i]], {i, m}];
96              ul = Map[Drop[#, -1] &, M];
97              ur = Map[Last[#] &, M];
98              For[i = 1, i < n + m, i++,
99                 If[Union[ul[[i]]] == {0} && ur[[i]] == 1,
100                Return[0]
101                ]
102             ];
103             r = 0;
104             k = n + m;
105             While[k > 0,
106                   If[Union[M[[k]]] == {0},
107                      r++;
108                      k--, Break[ ]]
109             ];
110             If[(r >= m), Return[1], Return[0]]
111               (*if C0 imply A0 return 1, otherwise return 0*)
112
113 Implication[A0_, B0_, C0_] :=
114             Module[{k=0, A, B, C1, coeMat1, coeMat2, coeMat3, rhs1, rhs2, rhs3, X},
115             If[A0 == {}||A0 == null||C0 == {}||(! ListQ[B0])||(! ListQ[C0]), Return[0]];
116             X = Union[A0[[1]], B0[[1]], C0[[1]]];
117             rhs1 = A0[[3]];
118             rhs2 = B0[[3]];
119             rhs3 = C0[[3]];
120             coeMat1 = Transpose[Map[Coefficient[#, X] &, Flatten[A0[[2]]]]];
121             coeMat2 = Transpose[Map[Coefficient[#, X] &, Flatten[B0[[2]]]]];
122             coeMat3 = Transpose[Map[Coefficient[#, X] &, Flatten[C0[[2]]]]];
123             A = Transpose[Join[coeMat1, rhs1]];
124             B = Transpose[Join[coeMat2, rhs2]];
125             C1 = Transpose[Join[coeMat3, rhs3]];
126             Return[Rowreduction[Join[A, B], C1]]
```

A.2 Instance Generator

```
---------------------------------------------------------------------------------------------

1   RandomInst[n_, m_, l_, f_] := Module[{},
2            Print["number of varialbles = ", n];
3            Print["number of equations= ", m];
4            Print[ "sparsity= ", l];
5            Table[RandomSymbol[n, l,f], {m}]
6
7
```

```
8   RandomSymbol[n_, l_, f_] :=
9               Module[{X, T, pos, Var, r, v},
10             (*last updat 2009-10-14*)
11              X = Table[Subscript[x, i   ], {i, n}];
12              T = Flatten[Outer[List, Sequence @@ Table[{0, 1}, {l}]], l - 1];
13             (*generates n variables randomly*)
14              While[True,
15                            pos = Union[Table[Random[Integer, {1, n}], {l}]];
16                     (*choose l variables from variable set X*)
17                            If[Length[pos] == l, Break[ ]]
18                      ];
19              Var = Table[X[[pos[[i]]]], {i, Length[pos]}];
20              While[True,
21                    r = Union[Table[Random[Integer, {1, 2^l}],
22                    {Random[Integer, {1, 2^l}]}]];
23                    If[Length[r] >= f, Break[]]
24                   ];
25                   (*Generates vectors of length>=f *)
26               v = Flatten[Map[T[[#]] &, {r}], 1];
27               Table[Transpose[{Var, v[[i]]}], {i, Length[v]}]
28            ]
```

A.3 Algorithms in Chapter 3

```
------------------------------------- Algorithm 1 ------------------------------------------

1   Glue[in1_, in2_] :=
2        Module[{i, j, s11, s1, s2, s22, Y, r, T1, T2, X1, X2, p1, p2},
3        (*Last updated 2009-10-5*)
4        If[! ListQ[in2[[1]]], Return[{}]];
5        If[in1 === empty, Return[in2]];
6        If[in1 == {} || in2 == {{}}, Return[{}],
7        s1 = Transpose[Flatten[in1, 1]];(*(X1,V1)*)
8        X1 = Union[s1[[1]]];
9        s2 = Transpose[Flatten[in2, 1]]; (*(X2,V2)*)
10       X2 = Union[s2[[1]]];         (*x1,x2,..xn*)
11       Y = Intersection[X1, X2];
12       If[Y == {},Return[Flatten[Table[Union[in1[[i]], in2[[j]]],
13                              {i, Length[in1]},{j, Length[in2]}],1]],
14              (*gives the trivial solution*)
15           (*else*)
16             p1 = Flatten[Map[Position[X1, #]&, Y]];
17             p2 = Flatten[Map[Position[X2, #]&, Y]];
18             T1 = Table[in1[[i, p1[[j]]]], {i, Length[in1]}, {j, Length[p1]}];
19             T2 = Table[in2[[i, p2[[j]]]], {i, Length[in2]}, {j, Length[p2]}];
20             r = {};
21                   For[i = 1, i <= Length[T1], i++,
22                     For[j = 1, j <= Length[T2], j++,
23                             If[T1[[i]] == T2[[j]], r = Append[r, Union[in1[[i]], in2[[j]]]]];
24                             ]
25                   ]
26          ]
27              ];
28       Return[r]]
29
30
31  ------------------------------------- Algorithm 2 ------------------------------------------
32
```

```
33
34 GlueSet[in_] :=
35         Module[{l, Z, counter, i},
36         counter = {};
37         l = Length[in];
38         Z = in[[1]];
39         Do[Z = Glue[Z, in[[i]]];
40            counter = Append[counter, Length[Z]],{i, l}];
41         Return[If[Z =!= {},{Z, counter},{"UnGluable", counter}]]]
42
43
44 ------------------------------------- Algorithm 3 ------------------------------------------
45
46
47 TreeGlue[in_] :=
48         (*Last updated 2009-8-31*)
49         Module[{d = 1, M, i, counter, T, m, index, A, solution = {}},
50         Off[Part::"partw"];
51         $HistoryLength = 0;
52         A = Append[in, {{}}];
53         m = Length[A];
54         index = Table[1, {Length[A]}];
55         M = Table[empty, {Length[A] + 1}];
56         counter = Table[0, {m + 1}];
57         While[d <= (m) && d > 0,
58               T = Glue[M[[d]], {A[[d]][[index[[d]]]]}];
59                        If[T != {},
60                 M[[d + 1]] = T;
61                 counter[[d]]++;
62                 index[[d]]++;
63                 d++,
64                  (*else*)
65                  index[[d]]++;
66                 (*if the current vector could not Glue with Md,
67                  try the next vector at the same tree level*)
68                  If[index[[d]] > Length[A[[d]]],
69                     (*if Glue operation could not go further then go one
70                     step back and try another vecor from previous symbol*)
71                     index[[d]] = 1;
72                     d--;
73                     If[d == 1 && (index[[1]] > Length[A[[1]]]), Break[ ]]
74                     ]
75                     (*after trying all vectors in the first symbol and no
76                     solution found, then break the loop*)
77                 ];
78                 If[d==m, solution=Append[solution,M[[d]]]]
79             ];
80   counter = Drop[counter, -2];
81   Print["Result of TreeGlue: "];
82   Return[If[d == 1,(*then*){solution, counter},
83             (*else*){"No Solution", counter}]]]
84
85
86 ------------------------------ Branches Cutting on Algorithm 3 --------------------------------
87
88
89 TreeGlueER[in_, z_] :=
90            Module[{d = 1, M, i, counter, T, m, index, A, solution = {}},
91            Off[Part::"partw"];
92            $HistoryLength = 0;
93            A = Append[in,{{}}];
94            m =Length[A];
```

```
95            index =Table[1,{Length[A]}];
96            M=Table[empty, {Length[A] + 1}];
97            counter=Table[0, {m + 1}];
98            While[d <= (m) && d > 0,
99                   k=d;
100                 While[k < (m - z), k++,
101                        t = {};
102                        A1 = A[[k]];
103                        For[i = 1, i <= Length[A1], i++,
104                        If[Intersection[symbolForm[M[[d]]][[1]],
105                            symbolForm[A1][[1]]] == {}, Break[ ]];
106                            (*check intersecting variables*)
107                            t = Append[t, GlueQ[M[[d]], {A1[[i]]}]]
108                          ];
109                        If[Union[t]=={0},
110                        d =d-1;  Break[ ]];
111                       ];
112                 T=Glue[M[[d]], {A[[d]][[index[[d]]]]}];
113                        If[T!={},
114                       M[[d+1]]=T;
115                       counter[[d]]++;
116                       index[[d]]++;
117                       d++,
118                      (*else*)
119                       index[[d]]++;
120                       If[index[[d]]>Length[A[[d]]],
121                       index[[d]] = 1;
122                       d--;
123                       If[d==1&&(index[[1]]>Length[A[[1]]]), Break[ ]]]];
124                       If[d==m, solution=Append[solution,M[[d]]]]
125                  ];
126             counter = Drop[counter, -2];
127             Print["Result of Glue After edge removing: "];
128             Return[If[d==1,(*then*){solution, counter},
129                       (*else*){"No Solution",counter}]]]
130
131
132 ------------------------------------------ Sorting ------------------------------------------
133
134
135 Sorting[in_] :=
136         Module[{ T1 = in[[1]], T2 = in[[2]]; R1, R2, RR, s, n, u, e, sorted, Rr, S, i},
137         S = in;
138         n = Length[Union[Flatten[Table[symbolForm[S[[i]]][[1]], {i,Length[S]}]]]];
139             (*number of variables*)
140         Do[If[Length[Union[symbolForm[S[[i]]][[1]], symbolForm[S[[j]]][[1]]]]
141               < Length[Union[symbolForm[T1][[1]], symbolForm[T2][[1]]]],
142            (*then*)
143                 T1 = S[[i]];
144                 T2 = S[[j]]],
145                   {i, Length[S] - 1}, {j, i + 1, Length[S]}];
146        (*searching s[i],s[j] whose length of union variables is smallest, i.e. choosing the
147         smallest |Xi\[Union]Xj|*)
148         u = DeleteCases[S, T1];
149         S = DeleteCases[u, T2];
150         sorted = {};
151         While[Length[S] > 0,
152                       s = n;
153                        Rr = {T1, T2};
154                        RR = Union[symbolForm[T1][[1]], symbolForm[T2][[1]]];
155                        For[i = 1, i <= Length[S], i++,
156                                  If[Length[Union[RR, symbolForm[S[[i]] ][[1]] ]] <= s,
```

```
157                   (*then*)
158                    s = Length[Union[RR, symbolForm[S[[i]]][[1]]]];
159                    e = S[[i]]]
160                   ];
161                       S = DeleteCases[S, e];
162                       sorted = Append[sorted, e]
163             ];
164          Return[Join[Rr, sorted]] ]
165
```

A.4 Algorithms in Chapter 4

```
------------------------------------- Algorithm 7 ------------------------------------------

1   MakePair[in_] := groupPair[Pair[in]]
2
3   Pair[a_] :=
4           Module[{n, h, pair},
5           (* input is a set of  vectors V *)
6           n=Length[a[[1]]];
7           h=Flatten[Outer[List, Sequence @@ Table[{0,1}, {n}]], n-1];
8           pair=Mod[Table[{a[[i]],a[[i]]+h[[j]],h[[j]]}, {j,2,Length[h]},{i,Length[a]}],2];
9           (*pairs (a,a+h,h) in V,*)
10          Return[pair]]
11
12
13  groupPair[pair_] :=
14            DeleteCases[Table[Takepair[pair[[i]]], {i,Length[pair]}],{}];
15           (*this function acts like grouping "pairs" for each h, generated by
16            function Pair[ ].
17            Example:groupPair[Pair[symbolForm[V]]],gives all pairs of (a,a+h h)
18            in s *)
19
20
21  Takepair[in_] :=
22           Module[{T = in;, M, p},
23           (* takes all possible pairs (a,a+h) for a paticullar h.
24           This function will be called by groupPair[ ]*)
25           M=Table[If[pairEQ[T[[i]],T[[j]]],T[[i]]],{i,Length[T]-1},{j, i+1,Length[T]}];
26           p=Flatten[DeleteCases[DeleteCases[M,({Null..}|Null),Infinity],{},Infinity],1];
27           Return[p]]
28
29
30  pairEQ[in1_, in2_] :=
31         Module[{},
32         (*checks two pairs (a,a+h)and (b,b+h) are equal,return True or False*)
33         If[Sort[in1]==Sort[in2], True, False]]
34
35
36  ------------------------------------- Algorithm 8 ------------------------------------------
37
38
39  MaximalCosets[in_] := Module[{out, s, Ps, v, t, G, Temp},
40               (*Latest update 2009-9-2, gives all  cosets for a symbol*)
41                Which[
42                      Length[in]==1, Return[{{in[[1]][[1]], in[[1]][[3]]}}],
43                      Length[in]==2, Return[{{in[[1]][[1]], BitXor[in[[1]][[1]], in[[2]][[1]]],
```

```
44                                  in[[1]][[3]]}}],
45                      Length[in]>2,G=Mod[NullSpace[{in[[1]][[3]],
46                                    Table[0,{Length[in[[1]][[3]]]}]}],2];
47                                  s = Table[Mod[G.in[[i]][[1]], 2], {i, Length[in]}];
48                                  v = in[[1]][[3]];
49                                                Ps = MakePair[s];
50                                  Temp=Flatten[Table[MaximalCosets[Ps[[i]]],{i,Length[Ps]}],1];
51                                  (*call recursion*)
52                                  t = Table[LinearSolve[G,Temp[[i]][[j]], Modulus -> 2],
53                                                    {i,Length[Temp]}, {j, Length[Temp[[i]]]}];
54                                                out = Table[Append[t[[i]], v], {i, Length[t]}],
55                                                True, Print["Bad argument"]
56                            ];
57                 Return[out]
58               ]
59
60
61  ------------------------------------- Algorithm 9 -------------------------------------------
62
63
64  MaximalCosetList[in_] :=
65                   Module[{s, p, coset, set, cosetList, i, j},
66                   (*updated 2009-9-17,gives a list of maximal cosets for a symbol*)
67                   (*s=symbolForm[in][[2]];*)
68                    p = MakePair[in];
69                    coset = Flatten[Table[MaximalCosets[p[[i]]], {i, Length[p]}], 1];
70                    set = Table[BaseExpand[coset[[i]]], {i, Length[coset]}];
71                    For[i = 1, i <= Length[set] - 1, i++,
72                                For[j = i + 1, j <= Length[set], j++,
73                                      If[set[[i]] == set[[j]], coset[[i]] = null]
74                                ]
75                       ];
76                    cosetList = DeleteCases[coset, null];
77                    Return[cosetList]]
78
79
80  ------------------------------------- Algorithm 10(Greedy)------------------------------------
81
82
83  MinCosetCover2[symbol_] :=
84                     Module[{out = {}, U = symbolForm[symbol][[2]]},
85                     (*find the minimal cover by Greedy approach*)
86                     While[U != {},
87                            cos = MaximalCosetList[U];
88                             L = Max[Table[Length[cos[[i]]], {i, Length[cos]}]];
89                             For[i = 1, i <= Length[cos], i++,
90                                   If[Length[cos[[i]]] == L, cc = cos[[i]]; Break[]]
91                                ];
92                             U = Complement[U, BaseExpand[cc]];
93                             out = Append[out, cc];
94                             If[Length[U] == 1, out = Append[out,
95                             {Flatten[U, 1], Table[0, {Length[U[[1]]]}]}];
96                              Break[ ] ];
97                           ];
98                     Return[out]]
99
100
101 ------------------------------------- Algorithm 11(Exact)-------------------------------------
102
103
104 MinCosetCover[symbol_, MaximalCoset_] :=
105              Module[{check = True, i = 1, cover, comb, B},
```

```
106            (*Updated 2009-9-17. gives the exact number of minimal coset covering for a symbol.
107              the second argument for input is generated by MaximalCosetList[symbol]*)
108            B = Table[BaseExpand[MaximalCoset[[i]]], {i, Length[MaximalCoset]}];
109            While[check,
110                  comb = Subsets[Range[Length[MaximalCoset]], {i}];
111                  (*indices combination of i-set in MaximalCoset*)
112                   For[j = 1, j <= Length[comb], j++,
113                       If[Union[Flatten[Table[B[[j]], {j, comb[[j]]}], 1]] == symbol,
114                          cover = Table[MaximalCoset[[j]], {j, comb[[j]]}];
115                          check = False];
116                       ];
117                  i++
118                  ];
119            Return[cover]]
120
121
122 ------------------------------------- Algorithm 12 --------------------------------------------
123
124 MSLEExact[in_] := Module[{lin},
125          (*Giving exact minimal number of linear equations for each symbol in the whole
126          system.  Input for this function are symbols generated by RandomInst[ ].*)
127          lin = LinEq[in];
128          Table[{lin[[i]][[1]], {lin[[i]][[2]][[j]]}, {lin[[i]][[3]][[j]]}},
129          {i,Length[lin]}, {j, Length[lin[[i]][[2]]]}]]
130
131
132
133 LinEq[in_] :=
134       Module[{r, V, cost, X, r = in,C1, c, T, p, cos, lin, LinearSys},
135      (*Last updated,2009-9-17; input a set of symbols (X1,V1),(X2,V2),..(Xm,Vm), output a set
136       of linear equation system C.X==c. which is represented as {{X},{C.X},{c}} in the output
137       formate. Cosets for this routine are computed by MinCosetCover[ ] *)
138       V = Table[symbolForm[r[[i]]], {i, Length[r]}];
139       X = Table[V[[i]][[1]], {i,Length[V]}]; (*variables in each polynomial*)
140       cos = Table[MinCosetCover[in[[i]]], {i, Length[in]}];
141       C1 = Table[Mod[NullSpace[Flatten[{Rest[cos[[i]][[j]]],
142                 {Table[0,{Length[cos[[i]][[j]][[1]]]}]}},1]],2],
143                 {i,Length[cos]},{j,Length[cos[[i]]]}];
144       c = Table[Mod[C1[[i]][[j]].cos[[i]][[j]][[1]], 2], {i, Length[cos]},
145                {j,Length[cos[[i]]]}];
146       T = Table[{X[[i]], C1[[i]]}, {i, Length[C1]}];
147       LinearSys = Table[T[[i]][[2]][[j]].T[[i]][[1]], {i, Length[T]},
148                         {j,Length[T[[i]][[2]]]}];(*gives all linear Equations C.X*)
149       lin = Table[{X[[i]], LinearSys[[i]], c[[i]]}, {i,Length[X]}];(*{{X},{C.X},{c}}*)
150       Return[lin]]
151
152
153
154
155
156 MSLEGreedy[in_] := Module[{lin},
157           (*Gives approximal number of minimal linear equation systems for each equation in the
158            whole system. Input for this function is symbol generated by RandomInst[ ].*)
159            lin = LinEq2[in];
160            Table[{lin[[i]][[1]], {lin[[i]][[2]][[j]]}, {lin[[i]][[3]][[j]]}},
161                  {i,Length[lin]}, {j, Length[lin[[i]][[2]]]}]]
162
163
164
165 LinEq2[in_] :=
166        Module[{r, V, cost, r = in, X, C1, c, T, p, cos, lin, LinearSys},
167        (*Last updated,2009-10-21; input a set of symbols  generated by RandomIns[ ].
```

```
168         Output a set of linear equation system C.X==c. which is represented as {{X},{C.X},{c}}
169         in the output formate, This subroutine computes the linear equation system using the
170         minimum cosets computed by greedy algorithm, MinimalCosetCover2[ ]*)
171         V = Table[symbolForm[r[[i]]], {i, Length[r]}];
172         X = Table[V[[i]][[1]], {i,Length[V]}]; (*variables in each polynomial*)
173         cos = Table[MinCosetCover2[in[[i]]], {i, Length[in]}];
174         C1 = Table[Mod[NullSpace[Flatten[{Rest[cos[[i]][[j]]],
175                     {Table[0,{Length[cos[[i]][[j]][[1]]]}]}},1]], 2],
176                     {i, Length[cos]}, {j, Length[cos[[i]]]}];
177         c = Table[Mod[C1[[i]][[j]].cos[[i]][[j]][[1]], 2],
178                  {i, Length[cos]}, {j,Length[cos[[i]]]}];
179         T = Table[{X[[i]], C1[[i]]}, {i, Length[C1]}];
180         LinearSys =Table[T[[i]][[2]][[j]].T[[i]][[1]], {i, Length[T]},
181                        {j,Length[T[[i]][[2]]]}];(*gives all linear Equations C.X*)
182         lin = Table[{X[[i]], LinearSys[[i]], c[[i]]}, {i,Length[X]}];(*{{X},{C.X},{c}}*)
183         Return[lin]]
184
185
186
```

A.5 Algorithms in Chapter 5

```
------------------------------------- Algorithm 13 -------------------------------------------

1   MSLEGlue[in1_, in2_] :=
2          Module[{X, c, g, gt, coeMat1, coeMat2, augment1, augment2, C1, newlin},
3          (*Latest update 2009-9-4*)
4          If[in1 == null, Return[in2]];
5          If[in1 == {} || in2 == {} || ! ListQ[in2], Return[{}]];
6          X = Union[in1[[1]], in2[[1]]];(*variable X*)
7          coeMat1 = Map[Coefficient[#, X] &, Flatten[in1[[2]]]];
8          coeMat2 = Map[Coefficient[#, X] &, Flatten[in2[[2]]]];
9          (*Coefficient matrix for in1 and in2 in variable X*)
10         augment1 = Transpose[Join[Transpose[coeMat1], in1[[3]]]];
11         augment2 = Transpose[Join[Transpose[coeMat2], in2[[3]]]];
12         (*augmented matrix for in1 and in2*)
13         If[ConsistenceQ[Join[coeMat1, coeMat2], Flatten[Join[in1[[3]], in2[[3]]], 1]],
14                     g =  RowReduce[Join[augment1, augment2];
15              g =  DeleteCases[g, Modulus -> 2], Table[0, {Length[g[[1]]]}]];
16                     gt = Transpose[g];
17                     c =  Last[gt];
18                     C1 = Transpose[Drop[gt, -1]];
19                     newlin = C1.X,
20       (*else*)
21               newlin = {}];
22        Return[If[newlin =!= {}, {X, {newlin}, {c}}, {}]]]
23
24
25  ------------------------------------- Algorithm 15 -------------------------------------------
26
27
28  MSLEGlueSet[in_] :=
29            Module[{  counter = {},   Z = in[[1]]},
30           (*2009-9-29*)
31            Do[Z = Union[DeleteCases[Flatten[Table[MSLEGlue[Z[[i]], in[[k]][[j]]],
32               {i, Length[Z]}, {j,Length[in[[k]]]}], 1], {}]];
33               counter = Append[counter, Length[Z]],
```

```
34              {k, Length[in]}];
35           Return[If[Z =!= {}, {Z, counter}, {"UnGluable", counter}]]
36
37
38
39   ------------------------------------- Algorithm 16 -------------------------------------------
40
41
42   MSLETreeGlue[in_] :=
43             Module[{m = Length[A], d=1, M, index,
44                    i, counter, T, A=Append[in,{{}}], solution={}},
45             (* Lateset update 2009-9-4*)
46             Off[Part::"partw"];
47             $HistoryLength = 0;
48             M = Table[null, {Length[A] + 1}];
49             index = Table[1, {Length[A]}];
50             counter = Table[0, {m + 1}];
51             While[d <= (m) && d > 0,
52                    T = MSLEGlue[M[[d]], A[[d]][[index[[d]]]]];
53                    If[T != {},
54                    M[[d + 1]] = T;
55                    counter[[d]]++;
56                    index[[d]]++;
57                    d++,
58                    (*else*)
59                    index[[d]]++;(*if the current linear equation could not Glue with Md,
60                    try the next equation at the same tree level*)
61                    If[index[[d]] > Length[A[[d]]],
62                     (*if Glue operation could not go further then go one step
63                     back and try another linear equation from previous symbol*)
64                        index[[d]] = 1;
65                        d--;
66                        If[d == 1 && (index[[1]] > Length[A[[1]]]), Break[ ]]
67                       (*after trying all linear equations in the first symbol
68                         and no solution found, then break the loop*)
69                     ];
70                     If[d==m,solution=Union[Append[solution,M[[d]]]]]
71                  ];
72              counter = Drop[counter, -2];
73              Print["Result of MSLETreeGlue: "];
74              Return[If[d == 1,(*then*){solution, counter},
75                         (*else*){"No Solution", counter}]]
76
77
78
79   ------------------------------------- Algorithm 17 -------------------------------------------
80
81
82   MSLETreeGlueReduction[in_] :=
83              Module[{m = Length[A], d=1, M, index, i, counter, T, checkinclue, im1, im2,
84                      A = Append[in, {{}}], solution = {}},
85              (* Lateset update 2009-9-4. Input is generated by function MSLEExact[symbol] *)
86              M = Table[null, {Length[A] + 1}];
87              index = Table[1, {Length[A]}];
88              counter = Table[0, {m + 1}];
89              While[d <= (m) && d > 0,
90                     checkinclue = False;
91                   im1=Quiet[Implication[M[[d]], A[[d, index[[d]]]], A[[d, index[[d]] + 1]]]];
92                   im2=Quiet[Implication[M[[d]], A[[d, index[[d]] + 1]], A[[d, index[[d]]]]]];
93                   Which[
94                        im1 == 1,T = MSLEGlue[M[[d]], A[[d]][[index[[d]]+1]]]; checkinclue=True,
95                        im2 == 1,T = MSLEGlue[M[[d]], A[[d]][[index[[d]]]]]; checkinclue = True,
```

```
96                          True,    T = MSLEGlue[M[[d]], A[[d]][[index[[d]]]]]
97                     ];
98                     If[T != {},
99                           counter[[d]]++;
100                          M[[d + 1]] = T;
101                          index[[d]]++;
102                          If[checkinclue, index[[d]]=index[[d]]+1];(*in case of implication true*)
103                          d++,
104                         (*else*)
105                          index[[d]]++;
106                          If[index[[d]] > Length[A[[d]]],
107                          index[[d]] = 1;
108                          d--;
109                          If[d == 1 && (index[[1]] > Length[A[[1]]]), Break[ ]]]
110                    ];
111                    If[d == m, solution = Union[Append[solution, M[[d]]]]]
112                 ];
113                 counter = Drop[counter, -2];
114                 Print["Result of LinearGlue3: "];
115                 Return[If[d == 1,(*then*){solution, counter},
116                           (*else*){"No Solution", counter}]]]
117
118
119
120 ------------------------------------- Algorithm 18 ------------------------------------------
121
122
123 MSLETreeGlueER[in_, z_] :=
124               Module[{m = Length[A], d=1, M, index, i, counter,
125                           T, A = Append[in,{{}}],solution ={}},
126               Off[Part::"partw", Part::"partd", Part::"pspec"];
127               $HistoryLength = 0;
128               M = Table[null, {Length[A] + 1}];
129               index = Table[1, {Length[A]}];
130               counter = Table[0, {m + 1}];
131               While[d <= (m) && d > 0,
132                     k=d;
133                     While[k < (m - z), k++,
134                           (*z=number of 0s from the end of list given by MSLETreeGlue[ ]*)
135                            t = {};
136                            A1 = A[[k]];
137                            For[i = 1, i <= Length[A1], i++,
138                                  If[Intersection[M[[d]][[1]], A1[[i]][[1]]] == {}, Break[ ]];
139                            (*if no intersecting variables, consistency checking is not applied*)
140                                   t = Append[t, ConsistencyQ[M[[d]], A1[[i]]]]
141                            ];
142                            If[Union[t] == {0},
143                            d = d - 1; Break[ ]];
144                    ];
145               T = MSLEGlue[M[[d]], A[[d]][[index[[d]]]]];
146               If[T != {},
147                   M[[d + 1]] = T;
148                   counter[[d]]++;
149                   index[[d]]++;
150                   d++,
151                   (*else*)
152                   index[[d]]++;
153                        If[index[[d]] > Length[A[[d]]],
154                          index[[d]] = 1;
155                          d--;
156                          If[d == 1 && (index[[1]] > Length[A[[1]]]), Break[]]]
157                    ];
```

```
158             If[d==m,
159                 solution=Union[Append[solution,M[[d]]]]]
160            ];
161            counter = Drop[counter, -2];
162            Print["Result of MSLETreeGlue After edge removing: "];
163            Return[If[d == 1,(*then*){solution, counter},
164                             (*else*){"No Solution", counter}]]]
```

Appendix B

Sample Experiments

In this appendix we will illustrate some small examples for the experimental environment. The routine `RandomInst[n,m,l,f ]` generates randomly `symbols` for the experiments. For the description of the input arguments cf. AppendixB.

Then we apply `TreeGlue[ symbols ]` to obtain the solutions to the system and compute how many gluing operations occurred during the solving procedure.

For the randomly generated symbols we apply `MSLEGreedy[ symbols ]` to compute MSLEs. Note that this routine calls subroutine `MinimalCosetCover2[ ]` which computes a coset covering by the greedy approach.

For the MSLEs computed by routine `MSLEGreedy[ symbols ]` we apply `MSLETreeGlue[ MSLE ]` to obtain the solution to the whole system and compute how many gluing operations occurred while solving the system.

1. **Generating Random Instances**

   We want to generate an instance of 8 symbols, where the number of variables in the whole system is 8, the sparsity 4 and for each symbol the $|V_i|$ is greater than 5.

```
In[44]:= s = RandomInst[8, 8, 4, 5];
         Map[symbolForm[#] &, s]

         number of varialbles = 8

         number of equations= 8

         sparsity= 4

In[47]:= s // TableForm

Out[47]//TableForm=
```

x_5	0	0	0	0	0	1	1	1
x_6	0	0	0	1	1	0	0	1
x_7	0	1	1	1	1	0	1	1
x_8	1	0	1	0	1	0	1	0

x_3	0	0	0	0	1	1	1
x_4	0	1	1	1	0	1	1
x_6	0	0	0	1	0	0	1
x_8	1	0	1	1	1	1	1

x_2	0	0	0	0	0	1	1	1	1
x_6	0	0	1	1	1	0	0	1	1
x_7	0	1	0	1	1	0	1	1	1
x_8	0	1	0	0	1	1	0	0	1

x_2	0	0	1	1	1	1	1
x_3	0	1	0	0	0	1	1
x_4	0	1	0	0	1	0	1
x_8	0	1	0	1	0	0	0

x_2	0	0	0	0	1	1
x_4	0	1	1	1	0	1
x_6	1	0	0	1	0	0
x_7	0	0	1	0	1	0

x_1	0	0	0	0	0	0	1	1	1	1	1
x_2	0	0	0	1	1	1	0	1	1	1	1
x_3	0	0	1	0	0	1	0	0	0	1	1
x_8	0	1	1	0	1	0	0	0	1	0	1

x_1	0	0	0	0	0	0	1	1
x_4	0	0	0	1	1	1	0	0
x_5	0	0	1	0	0	1	0	1
x_7	0	1	0	0	1	1	0	1

x_1	0	0	0	0	0	1	1
x_3	0	0	1	1	1	1	1
x_5	1	1	0	0	1	1	1
x_6	0	1	0	1	1	0	1

2. **Solve the System By Gluing1 Algorithm**

   Now we run `TreeGlue[ s ]`. The output of this routine is in the form {{{{*expr1*}}},{*expr2*}}, where *expre1* represents the solutions to the system, *expre1* represents the number of gluing operations occurred at each tree level while solving the system.

```
In[49]:= TreeGlue[s]
         Result of Glue:
Out[49]= {{{{{x1, 0}, {x2, 0}, {x3, 1}, {x4, 1}, {x5, 0}, {x6, 0}, {x7, 1}, {x8, 1}}}}, {8, 16, 18, 5, 2, 2, 2, 1}}
```

3. **Compute MSLE For The System**

   Let `s` be the input, and run the routine `MSLEGreedy[ s ]`. It's output for each $CX = c$ is represented in the form $\{\{\{X\}, \{\{CX\}\}, \{\{c\}\}\}$.

```
In[50]:= l = MSLEGreedy[s]

Out[50]= {{{{x5, x6, x7, x8}, {{x7, x5}}, {{1, 0}}}, {{x5, x6, x7, x8}, {{x7 + x8, x6, x5}}, {{0, 0, 1}}},
           {{x5, x6, x7, x8}, {{x5 + x8, x5 + x7, x5 + x6}}, {{1, 0, 0}}}}, {{{x3, x4, x6, x8}, {{x8, x4}}, {{1, 1}}},
           {{x3, x4, x6, x8}, {{x4 + x8, x6, x3}}, {{1, 0, 0}}}, {{x3, x4, x6, x8}, {{x8, x6, x4, x3}}, {{1, 0, 0, 1}}}},
          {{{x2, x6, x7, x8}, {{x7 + x8, x2}}, {{0, 0}}}, {{x2, x6, x7, x8}, {{x7, x6, x2}}, {{1, 1, 1}}},
           {{x2, x6, x7, x8}, {{x7 + x8, x6, x2}}, {{1, 0, 1}}}, {{x2, x6, x7, x8}, {{x8, x7, x6, x2}}, {{0, 1, 1, 0}}}},
          {{{x2, x3, x4, x8}, {{x8, x2}}, {{0, 1}}}, {{x2, x3, x4, x8}, {{x3 + x8, x3 + x4, x2}}, {{0, 0, 0}}},
           {{x2, x3, x4, x8}, {{x8, x4, x3, x2}}, {{1, 0, 0, 1}}}}, {{{x2, x4, x6, x7}, {{x6, x4, x2}}, {{0, 1, 0}}},
           {{x2, x4, x6, x7}, {{x7, x6, x2}}, {{0, 1, 0}}}, {{x2, x4, x6, x7}, {{x4 + x7, x6, x2}}, {{1, 0, 1}}}},
          {{{x1, x2, x3, x8}, {{x2, x1}}, {{1, 1}}}, {{x1, x2, x3, x8}, {{x3, x1}}, {{0, 0}}},
           {{x1, x2, x3, x8}, {{x2 + x8, x3, x1}}, {{1, 1, 0}}}, {{x1, x2, x3, x8}, {{x8, x3, x2, x1}}, {{0, 0, 0, 1}}}},
          {{{x1, x4, x5, x7}, {{x5, x1}}, {{0, 0}}}, {{x1, x4, x5, x7}, {{x5 + x7, x4, x1}}, {{0, 0, 1}}},
           {{x1, x4, x5, x7}, {{x4 + x7, x5, x1}}, {{0, 1, 0}}}}, {{{x1, x3, x5, x6}, {{x3 + x5, x1}}, {{1, 0}}},
           {{x1, x3, x5, x6}, {{x5, x3, x1}}, {{1, 1, 1}}}, {{x1, x3, x5, x6}, {{x6, x5, x3, x1}}, {{1, 1, 1, 0}}}}}
```

4. **Solve The System By MSLE-Gluing Algorithm**

   The output of the Gluing Algorithm on the list of MSLE is as follows.
   The output is represented in the form
   $\{\{\{\{X\}, \{\{CX\}\}, \{\{c\}\}\}\}, \{\textit{number of gluing operations at each tree level}\}\}$

```
In[51]:= TreeLinGlue[l]
         Result of LinearGlue:
Out[51]= {{{{x1, x2, x3, x4, x5, x6, x7, x8}, {{x1, x2, x3, x4, x5, x6, x7, x8}}, {{0, 0, 1, 1, 0, 0, 1, 1}}}},
          {3, 9, 11, 4, 2, 2, 2, 1}}
```

B.1 Experimental Environment

All experiments are run on a PC with Pentium 4, 2GHz, 2 cores and 2GB RAM. Algorithms are implemented in Mathematica 7.0.

Appendix C

Data Corresponding to Figure 6.9

```
In[2]: l = MSLETreeGlue[<< lr64]

Result of MSLETreeGlue:

Out[2]=
 {{}, {4, 7, 13, 22, 28, 96, 240, 376, 888, 1220, 2440, 4038,
8076, 30672, 37872, 44248, 62984, 91552, 88144, 84560, 117872,
137736, 188696, 128792, 294964, 321282, 392412, 550534, 512474,
876200, 1133352,2100252, 1342628, 1056452, 1048188, 1311540,
2047340, 1500114, 825609,669455, 446912, 557243, 295585, 360433,
495121, 253918, 235642, 515525,252805, 223677, 82344, 93148,
83634, 46368, 22294, 23695, 16716, 9115,
5767, 3486, 1427, 416, 70, 26}}

In[3]: le=MSLETreeGlueER[<< lr64]

Result of MSLETreeGlue After edge removing:

Out[3]=
{{}, {4, 7, 13, 22, 28, 60, 156, 242, 468, 513, 1018, 1404, 1646,
5484, 4436, 5015, 3851, 5282, 5273, 5973, 8116, 10060, 12614, 11304,
24866, 14818, 14287, 16164, 12761, 5405, 3536, 4038, 3515, 1143,
574, 334, 492, 310, 310, 334, 466, 742, 762, 698, 1076, 824, 824,
2241, 20, 20, 20, 38, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26}}}
```

The followings are logarithmic scaled output of above data.

```
In[4]: N[Map[Log, l]]

Out[4]= {1.38629, 1.94591, 2.56495, 3.09104, 3.3322, 4.56435,
5.48064, 5.92959, 6.78897, 7.10661, 7.79975, 8.3035, 8.99665,
10.3311, 10.542, 10.6976, 11.0506, 11.4247, 11.3867, 11.3452,
11.6774, 11.8331, 12.1479, 11.766, 12.5946, 12.6801, 12.8801,
13.2186, 13.147, 13.6833, 13.9407, 14.5576, 14.1101, 13.8704,
13.8626, 14.0867, 14.5321, 14.2211, 13.6239, 13.4142, 13.0101,
13.2308, 12.5967, 12.7951, 13.1126, 12.4448, 12.3701, 13.1529,
12.4404, 12.318, 11.3187, 11.4419, 11.3342, 10.7444, 10.0121,
10.073, 9.72412, 9.11768, 8.65991, 8.15651, 7.26333, 6.03069,
4.2485, 3.2581}

In[5]:= N[Map[Log, le]]

Out[5]= {1.38629, 1.94591, 2.56495, 3.09104, 3.3322, 4.09434,
5.04986, 5.48894, 6.14847, 6.24028, 6.9256, 7.24708, 7.4061,
8.60959, 8.39751, 8.52019, 8.25609, 8.57206, 8.57035, 8.695,
9.00159, 9.21632, 9.44256, 9.33291, 10.1213, 9.6036, 9.56711,
9.69054, 9.45415, 8.59508, 8.17075, 8.3035, 8.16479, 7.04141,
6.35263, 5.81114, 6.19848, 5.73657, 5.73657, 5.81114, 6.14419,
6.60935, 6.63595, 6.54822, 6.98101, 6.71417, 6.71417, 7.71468,
2.99573, 2.99573, 2.99573, 3.63759, 3.2581, 3.2581, 3.2581,
3.2581, 3.2581, 3.2581, 3.2581, 3.2581, 3.2581, 3.2581, 3.2581,
3.2581}
```

Bibliography

[ACGK99] Giorgio Ausiello, Pierluigi Crescenzi, Giorgio Gambosi, and Viggo Kann. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Springer, 1st edition, 1999.

[AP01] Ara Aleksanyan and Mihran Papikian. On coset coverings of solutions of homogeneous cubic equations over finite fields. The Electronic Journal of Combinatorics, Volume 8, Available from `http://www.combinatorics.org/Volume_8/PDF/v8i1r22.pdf`, 2001.

[BCJ07] Gregory V. Bard, Nicolas T. Courtois, and Chris Jefferson. Efficient methods for conversion and solution of sparse systems of low-degree multivariate polynomials over GF(2) via SAT-Solvers, available at http://eprint.iacr.org/2007/024. 2007.

[BW93] Thomas Becker and Volker Weispfenning. *Gröbner bases: a Computational Approach to Commutative Algebra*. Springer-Verlag, New York, 1st edition, 1993.

[CKPS00] Nicolas Courtois, Er Klimov, Jacques Patarin, and Adi Shamir. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In *In Advances in Cryptology, Eurocrypt2000, LNCS 1807*, pages 392–407. Springer-Verlag, 2000.

[C.L06] David C.Lay. *Linear Algebra*. Pearson Education, third edition, 2006.

[Coo71] Stephen.A. Cook. The complexity of theorem proving procedures. *Conference Record of Third Annuual ACM Symposium on Theoty of Computation(1971)*, (3-5):151–158, 1971.

[CP02] Nicolas T. Courtois and Josef Pieprzyk. Cryptanalysis of block ciphers with overdefined systems of equations. pages 267–287. Springer, 2002.

[DCP06] Sanjoy Dasgupta and Umesh Vazirani Christos Papadimitriou. *Algorithm*. Alan R.Apt, 2006.

[Die04] Claus Diem. The xl-algorithm and a conjecture from commutative algebra. In *Proceedings of Asiacrypt 2004, LNCS, volume 3329*, pages 323–337. Springer-Verlag, 2004.

[DR99] Joan Daemen and Vincent Rijmen. AES proposal: Rijndael. NIST AES webside: `http://csrc.nist.gov/archive/aes/index.html`, 1999.

[Fra03] John B. Fraleigh. *A First Course In Abstract Agebra*. Greg Tobin, 2003.

[GJ79] Michael Garey and David S. Johnson. *A guide to the theory of NP-completeness - A Series of books in the mathematical sciences*. W.H. Freeman, 1979.

[Gol96] Van Loan Charles F. Golub, Gene H. *Matrix Computations*. Johns Hopkins, 3rd edition, 1996.

[Kar72] Richard.M Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, pages 85–103, 1972.

[KET06] Jon Kleinberg and Éva Tardos. *Algorithm Design*. Pearson Education; International edition, 2006.

[KS99] Aviad Kipnis and Adi Shamir. Cryptanalysis of the HFE public key cryptosystem. Springer, 1999.

[Lev07] Anany Levitin. *Introduction to The design and Analysis of Algorithms*. Greg Tobin, 2nd edition, 2007.

[Rad04] Håvard Raddum. Solving non-linear sparse equation system over GF(2) using graghs. *University of Bergen*, 2004.

[RS06] Håvard Raddum and Igor Semaev. New technique for solving spars equation systems. *Cryptology ePrint*, 2006. Available from: `http://www.eprint.iacr.org/`.

[RS07] Håvard Raddum and Igor Semaev. Solving MRHS linear equations. Cryptology ePrint Archive, Report 2007/285, 2007. `http://eprint.iacr.org/`.

[R.W88] R.W.FareBrother. *Linear Least Squares Computations (Statistics: a Series of Textbooks and Monogrphs)*. Publisher: CRC, 1th edition, 19 February 1988.

[Sch08] Thorsten Ernst Schiling. Comparison of solving techniques for non-linear sparse equations over finite fields with application in cryptanalysis,Master's thesis. avalable from https://bora.uib.no/bitstream/1956/3208/1/47383227.pdf cite 2009/12/23, June 2008.

[Sem08] Igor Semaev. On solving sparse algebraic equations over finite fields. Designs, Codes and Cryptography 49(2008),pp.47-60, 2008.

[sem09a] Igor semaev. Multiple Side Linear Equations and Circuit Lattices. On CECC conference, avalabel from: http://conf.fme.vutbr.cz/cecc09/lectures/semaev.pdf, 25 june 2009.

[Sem09b] Igor Semaev. Sparse algebraic equations over finite fields. SIAM Journal on Computing, 39(2009), pp.388-409, 2009.

[Sem09c] Igor Semaev. Sparse boolean equations and circuit lattices. Cryptology ePrint Archive, Report 2009/252, 2009. http://eprint.iacr.org/.

[Val98] Gabriel Valiente. *Algorithms on Trees and Graphs*. Springer, 1998.

[Wol92] Stephen Wolfram. *Mathematica: A System for Doing Mathematics by Computer*. Addison-Wesley Publishing Company,Inc, second edition, November,1992.

[ZV00] Arkadij Zakrevskij and Irina Vasilkova. Reducing large systems of logical equations. *4th int. Workshop on Boolean Problems, Freiberg University*, pages 21–22, 2000.