J Syst Sci Complex (2016) 29: 789–804

On Implementing the Symbolic Preprocessing Function over Boolean Polynomial Rings in Gröbner Basis Algorithms Using Linear Algebra*

SUN Yao · HUANG Zhenyu · LIN Dongdai · WANG Dingkang

DOI: 10.1007/s11424-015-4085-1
Received: 4 April 2014 / Revised: 11 September 2014

Abstract Some techniques using linear algebra was introduced by Faugère in F4 to speed up the reduction process during Gröbner basis computations. These techniques can also be used in fast implementations of F5 and some other signature-based Gröbner basis algorithms. When these techniques are applied, a very important step is constructing matrices from critical pairs and existing polynomials by the Symbolic Preprocessing function (given in F4). Since multiplications of monomials and polynomials are involved in the Symbolic Preprocessing function, this step can be very costly when the number of involved polynomials/monomials is huge. In this paper, multiplications of monomials and polynomials for a Boolean polynomial ring are investigated and a specific method of implementing the Symbolic Preprocessing function over Boolean polynomial rings is reported. Many examples have been tested by using this method, and the experimental data shows that the new method is very efficient.

Keywords Boolean polynomial rings, Gröbner basis, implementation, linear algebra.

1 Introduction

Gröbner basis is a powerful tool of solving systems of polynomial equations as well as many important problems in algebra. Since Gröbner basis was proposed in 1965[1], many improvements have been made to speed up the algorithms for computing Gröbner bases. One

SUN Yao · HUANG Zhenyu (Corresponding Author) · LIN Dongdai
SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, *China*.
Email: sunyao@iie.ac.cn; huangzhenyu@iie.ac.cn; ddlin@iie.ac.cn.
WANG Dingkang
KLMM, Academy of Mathematics and Systems Science, Chinese Academy of Sciences, Beijing 100090, *China*.
Email: dwang@mmrc.iss.ac.cn.
*This research is supported by the National Key Basic Research Program of China under Grant Nos. 2013CB834203 and 2011CB302400, the National Nature Science Foundation of China under Grant Nos. 11301523, 11371356, 61121062, the Strategic Priority Research Program of the Chinese Academy of Sciences under Grant No. XDA06010701, and IEE's Research Project on Cryptography under Grant Nos. Y3Z0013102, Y3Z0018102, and Y4Z0061A02.
◇ *This paper was recommended for publication by Editor LI Ziming.*

important improvement is that Lazard pointed out the strong relation between Gröbner bases and linear algebra[2]. This idea has been implemented in F4 by Faugère[3], and in XL type algorithms by Courtois, et al.[4] and Ding, et al.[5]. Up to now, Faugère's F4[3] and F5[6] are the most efficient algorithms for computing Gröbner bases, particularly in finite fields.

The efficiency of the F4 algorithm mainly comes from the usage of linear algebra for reductions, because the most costly computation in Gröbner basis algorithms is the reduction of polynomials. Linear algebra has also been introduced to signature-based algorithms to speed up the efficiency, including F5 in matrix style[7, 8] and the GVW algorithm[9] in matrix style[10].

One important step in Gröbner basis algorithms that use linear algebra is to convert polynomials to matrices. The matrices are constructed from two kinds of polynomials, polynomials to be reduced and polynomials used to reduce others. When the matrices are constructed, then the elimination of matrices can be done. The procedure of constructing matrices is done by the Symbolic Preprocessing function in F4. There are similar functions in [7, 10]. Since multiplications of monomials and polynomials are involved in the function, constructing matrices may be very costly if the number of polynomials/monomials is huge. This procedure may be even more expensive over Boolean polynomial rings, because for monomials m_1, m_2, and t in a Boolean polynomial ring, $m_1 \succ m_2$ does not always imply $tm_1 \succ tm_2$ w.r.t. the orderings on monomials.

According to the authors' knowledge, there are few papers discussing about the implementation details of the Symbolic Preprocessing function, particularly over the finite field $GF(2)$. In this paper, we will report our method of implementing the Symbolic Preprocessing function over Boolean polynomial rings. The experimental results show our method is very efficient.

This paper is organized as follows. We introduce some notations and give the description of the Symbolic Preprocessing function in Section 2. We report a method for implementing the Symbolic Preprocessing function over Boolean polynomial rings in Section 3. Some experimental results are given in Section 4, and concluding remarks follow in Section 5.

2 Preliminaries

2.1 Notations

Let $\mathbb{F}_2[X] := \mathbb{F}_2[x_1, x_2, \cdots, x_n]$ be a polynomial ring over the finite field $\mathbb{F}_2 = GF(2)$ in n variables $X = \{x_1, x_2, \cdots, x_n\}$, and we always require $x_1 > x_2 > \cdots > x_n$ in this paper. Given a monomial ordering $\prec$ on monomials in $\mathbb{F}_2[X]$, for a polynomial $f = x^{\alpha_1} + x^{\alpha_2} + \cdots + x^{\alpha_t} \in \mathbb{F}_2[X]$, the leading monomial of f w.r.t. $\prec$ is defined as $\mathrm{lm}(f) := \max_{\prec}\{x^{\alpha_i} \mid i = 1, 2, \cdots, t\}$.

Let $E := \{x_1^2 + x_1, x_2^2 + x_2, \cdots, x_n^2 + x_n\}$ be a set of polynomials in $\mathbb{F}_2[X]$, and we call these polynomials field polynomials. Then $\mathbb{B}_n := \mathbb{F}_2[X]/\langle E\rangle$ is a Boolean polynomial ring in n variables. Elements in $\mathbb{B}_n$ are called Boolean polynomials. For a Boolean polynomial $f = [x^{\alpha_1}] + [x^{\alpha_2}] + \cdots + [x^{\alpha_t}] \in \mathbb{B}_n$, where $[x^{\alpha_i}]$ denotes the coset of $x_i^{\alpha} \in \mathbb{F}_2[X]$ in $\mathbb{F}_2[X]/\langle E\rangle$ and x^{α_i} is a normal form w.r.t. $\langle E\rangle$, we still write $f = x^{\alpha_1} + x^{\alpha_2} + \cdots + x^{\alpha_t} \in \mathbb{B}_n$ in this paper if no confusions occur. Please note that for each monomial in $\mathbb{B}_n$, the degree of each variable is at most 1.

Let $M(X) := \{x^\alpha \mid \alpha \in \{0,1\}^n\}$ be all monomials in $\mathbb{B}_n$. Monomial orderings on $\mathbb{F}_2[X]$ can be deduced to orderings on monomials of $\mathbb{B}_n$ directly. Please note that orderings on $M(X)$ are not real monomial orderings in mathematical senses. With orderings on $M(X)$, leading monomials of Boolean polynomials can be defined similarly.

Let f be a Boolean polynomial in $\mathbb{B}_n$. Then $M(f)$ denotes the set of all monomials appearing in f. For a set of Boolean polynomials $H \subset \mathbb{B}_n$, $M(H)$ denotes the set of all monomials in polynomials in H, i.e., the union of $M(f)$ for all $f \in M$.

2.2 The Symbolic Preprocessing Function

In Gröbner basis algorithm that uses linear algebra for reductions, including F4[3], F5 in matrix style[7] and GVW in matrix style[10], one important procedure is constructing matrices for eliminations. Constructing a matrix in Gröbner basis algorithms usually contains two steps. The first step is to generate polynomials to be reduced. These polynomials generally come from critical pairs/S-pairs. The second step is to generate polynomials that are used to reduce others. These polynomials are always obtained from a given polynomial set. During these two steps, all involved polynomials are converted to rows of a matrix. When all needed polynomials have been put into this matrix, the elimination of this matrix can then be done.

In F4, Faugère gives the following function to construct matrices for eliminations. Notations are slightly different from Faugère's original version. Matrix F5 in [7] and matrix GVW in [10] have similar functions.

Function `Symbolic Preprocessing`

Input : G, a finite subset of $\mathbb{B}_n$;
$L = \{(t, f) \mid t \in M(X), f \in G\}$, a finite subset of $M(X) \times \mathbb{B}_n$.
Output: H, a finite subset of $\mathbb{B}_n$.

```
1  begin
2      H ⟵ {tf | (t, f) ∈ L}
3      Done ⟵ {lm(h) | h ∈ H}
4      while M(H) ≠ Done do
5          m ⟵ a monomial of M(H) \ Done
6          Done ⟵ Done ∪ {m}
7          if there exists g ∈ G such that lm(g) | m then
8              H ⟵ H ∪ {(m/lm(g))g}
9      return H
10 end
```

In the above function, the input set L are obtained from critical pairs which have been generated earlier, and the polynomials in $\{tf \mid (t, f) \in L\}$ are all polynomials to be reduced. Polynomials used to reduce others are obtained from the set G after multiplying proper multipliers at Line 8.

This function is straight, but does not contain implementing details. To implement the Symbolic Preprocessing function efficiently over Boolean polynomial rings, we need to consider

the following two problems.

(a) Multiplying $t \in M(X)$ and $f \in G$ over a Boolean polynomial ring often creates many duplicated monomials, and these duplicated monomials usually will vanish in the product. For example, in the Boolean polynomial ring $\mathbb{B}_4 = \mathbb{F}_2[x_1, x_2, x_3, x_4]/\langle x_1^2 + x_1, x_2^2 + x_2 \ldots, x_4^2 + x_4\rangle$, the product of x_1x_2 and $x_1x_3 + x_2 + x_3 + x_4$ is $x_1x_2x_3 + x_1x_2 + x_1x_2x_3 + x_1x_2x_4$. The monomial $x_1x_2x_3$ appears twice and should vanish. However, the duplication of monomials is usually not easy to be detected in Boolean polynomial rings, because $x^\alpha \prec x^\beta$ does not always imply $x^\gamma x^\alpha \prec x^\gamma x^\beta$ in Boolean polynomial rings, and consequently, monomials in the product are usually not in a sorted order.

If the duplicated monomials in the product tf are not detected, more space will be used to store the product and many checks, such as at Line 5, may be done to the same monomial several times.

(b) At Line 5, we have to determine whether a monomial is in $M(H)$ and not in Done. This procedure may be costly when the size of $M(H)$ and Done are huge. Unfortunately, the size of $M(H)$ is often very large in practical problems. For example, to break HFE challenge $1^{[11]}$, a 307126×1667009 matrix over $\mathbb{F}_2$ is constructed. This means about 1.7 million distinct monomials will appear, and billions of monomials will be considered when constructing the matrix.

Moreover, polynomials may be appended to H at Line 8, and consequently, the set $M(H)$ should be also enlarged. In this case, we should make sure no duplicated monomials are appended to $M(H)$, i.e., monomials in both $m/\mathrm{lm}(g)g$ and $M(H)$ should not be put into $M(H)$, since duplicated monomials in $M(H)$ will cost redundant checks at Line 5.

3 On Implementing the Symbolic Preprocessing Function

In this section, we will introduce our method more specifically. First, main ideas of our method are given in Subsection 3.1. In Subsection 3.2, we discuss about basic data structures used in our implementation. Next, we present our method of implementing the Symbolic Preprocessing function in Subsection 3.3. In the next following subsection, a related subfunction is discussed. At last, a toy example is given to illustrate the method in Subsection 3.5.

In our implementation, the monomial ordering is the Graded Reverse Lexicographic (GRLex for short) ordering.

3.1 Main Ideas

We find the two problems given in Subsection 2.2 can be abstracted to the following specific problems:

(A) How to determine whether a monomial appears in one polynomial more than once?

(B) How to determine whether a monomial appears in a set of monomials?

A trivial method of solving the above problems is to compare monomials. That is, for Problem (A), we can simply compare every two monomials in a polynomial to find duplicated ones, and for Problem (B), we can simply compare the monomial with every monomial in the

set. Obviously, this trivial method is not very efficient, even though sorting monomials in the polynomial and the set can decrease the number of comparisons.

Our main idea of solving the above two problems is using a unique monomial list for all monomials. We require the monomial list having two properties: 1) Each monomial is at a unique position of the monomial list; and 2) different monomials are at different positions.

Using this monomial list, Problems (A) and (B) can be solved easily.

For Problem (A), let f be a Boolean polynomial. For any monomial m in f, we can find the position of m in the monomial list first, and then mark the information of f at m's position. If for some m, the information of f appears for even times, then the monomial m should vanish in f. So to check whether a monomial is redundant in a polynomial, we only need to find the position of this monomial in the monomial list, and check the marks for the polynomial.

For Problem (B), let M be a set of monomials. For any monomial m in M, we can find the position of m in the monomial list, and then at m's position we make a mark which shows that m appears in M. Then for any monomial m', to check whether m' appears in M, we only need find the position of m' in the monomial list and check the mark whether m' appears in M.

A key step in our method is to find the position of a given monomial in the monomial list. This step is done efficiently by using a hash table, and it will be discussed in Subsection 3.4.

Compared with the trivial method for solving Problems (A) and (B), our method avoids comparing and sorting monomials, and only needs to access the information of the monomial list repeatedly. So the complexity of our method is relatively low, and the experimental results in Section 4 show our method is indeed very efficient.

3.2 Data Structure

There are two important data structures will be used: Data structures for monomials and for the output of the Symbolic Preprocessing function.

First, we use a monomial list to store monomials that will be used during computations. The monomial list is an array of monomials sorted in an ascending order w.r.t. the GRLex ordering. The length of this monomial list is dominated by a degree, say Deg_Limit. That is, we always store all monomials with degrees smaller or equal to Deg_Limit in the monomial list. Positions of monomials in the monomial list become the unique identity numbers of these monomials throughout the computations. For example, monomials in $\mathbb{B}_4 = \mathbb{F}_2[x_1, x_2, x_3, x_4]/\langle x_1^2 + x_1, x_2^2 + x_2, \cdots, x_4^2 + x_4\rangle$ with Deg_Limit = 2 are stored in the following list, where $x_1 > x_2 > x_3 > x_4$ in letters:

Identity Number	0	1	2	3	4	5	6	7	8	9	10
Monomial	1	x_4	x_3	x_2	x_1	x_3x_4	x_2x_4	x_1x_4	x_2x_3	x_1x_3	x_1x_2

Please note that positions of arrays start from 0 not 1 in this paper. With this monomial list, each monomial can be referred by its identity number, and Boolean polynomials can be stored as arrays of identity numbers. For example, the polynomial $x_2x_3 + x_2 + x_4 + 1$ can be stored as $[8, 3, 1, 0]$.

Second, we use a sparse matrix in row-compressed form to store all polynomials in the output of the Symbolic Preprocessing function. This sparse matrix contains two arrays. The array ids stores all identity numbers of the monomials that appear in the output polynomials. For the first monomial of each polynomial, the position of its identity number in ids is stored in the array startpos. Moreover, the last element of the array startpos gives the length of the array ids.

For example, assume $H = \{x_2x_3 + x_2 + x_4 + 1, x_2x_4 + x_3, x_3x_4 + x_1\}$ is the output set. Using the monomial list in the last paragraph, we can store H by the following two arrays:

startpos				0	4	6	8	
ids	8	3	1	0	6	2	5	4

The identity numbers of monomials in the i-th polynomials is stored in the array ids from the position startpos$[i]$ to startpos$[i+1]-1$.

In practical implementation, more auxiliary information should be stored. The monomial list and the sparse matrix are defined below in C++:

```
monnode_t *mon_list;
matrix_t mat;
```

In our implementation, the mon_list array is unique, and is updated due to the maximal degree of polynomials appearing in computations. We may use several matrices of type matrix_t, since several matrices may be constructed.

Definition of matrix_t is given below:

```
struct matrix_t
{
    int num;
    int *startpos;
    int *ids;
};
```

Annotations of members in the above structure is given below:

1) Num, gives the number of rows in a row-compressed sparse matrix, and gives the number of columns in a column-compressed sparse matrix.

2) Startpos, points to an allocated array of integers that stores the starting position of each row (or column). The starting position of the i-th row (or column) in the array ids is startpos $[i]$, where $i = 0, 1, \cdots$, num-1.

3) Ids, points to an allocated array of all identity numbers in this matrix.

The structure monnode_t is presented below:

```
struct monnode_t
{
    mon_t mon;
    char type;
    int appeartimes;
    int lastrowpos;
    int lastpos;
};
```

Here are some annotations for the members of monnode_t. Assume mat is a matrix to be constructed, i.e., mat will contain all information in the output of the Symbolic Preprocessing function.

1) Mon, is a monomial stored in type of mon_t. Each monnode_t corresponds to one and only one monomial, and the monomial is stored in mon. The position of monnode_t in the array mon_list is then the unique identity number of the monomial mon. Since there are many methods of storing a monomial in $\mathbb{B}_n$, the definition of mon_t can be various. Take the monomial $x_1x_3x_4 \in \mathbb{B}_4$ for example, one can either stores exponents of each variable by bits like $[1, 0, 1, 1]$, or only stores the appearing subscripts of x like $[1, 3, 4]$. Ether method can be used as the definition of mon_t.

2) Type, shows the status of the monomial mon. type = 0, means mon does not appear in the output matrix mat; type = 1, means mon has appeared in the output mat but is *not* a leading monomial of some polynomial; type = 2, means mon has appeared in mat and is the leading monomial of some polynomial. type is set to 0 initially. The type of mon is used to decide whether it is necessary to search polynomials from G to eliminate the monomial mon at Line 7 of the Symbolic Preprocessing function.

3) Appeartimes, shows how many times the monomial mon has appeared in the output mat. This value is used to check whether mon really appears in mat, and it is also useful when transforming mat to column-compressed form.

4) Lastrowpos, shows the position of the row where mon appears in mat for the last time. This value is used to check whether mon appears in the same row twice (or for even times). lastrowpos is set to -1 initially.

5) Lastpos, shows where the identity number of mon is stored in mat when mon appears for the last time. The value of lastpos is only used when mon will appear in the same row twice (or for even times). Specifically, if mon has already appeared in some row and the same mon will also be appended to the same row, then both mon should vanish in this row. In practical implementation, the identity number of the second mon will not be appended to mat, and besides, we should also clear the record of first mon in mat. Then lastpos is used.

Please note that, memories for the arrays mon_list, `startpos` and ids should be allocated and adjusted during computations.

3.3 Implementing the Symbolic Preprocessing Function

Our main implementation algorithm for the Symbolic Preprocessing function is given below.

Algorithm 2: Symbolic Preprocessing in Implementation (SPI)

Input : G, a finite subset of $\mathbb{B}_n$;
$L = \{(t, f) \mid t \in M(X), f \in G\}$, a finite subset of $M(X) \times \mathbb{B}_n$;
mon_list, an array of monnode_t;
mat, a matrix of type matrix_t.

Output: The matrix mat.

```
1  begin
2      rowpos⟵0
3      id_length⟵ the length of mon_list
4      for i from 0 to id_length − 1 do
5          mon_list[i].type ⟵ 0
6          mon_list[i].appeartimes ⟵ 0
7          mon_list[i].rowpos ⟵ − 1
8      for each (t, f) ∈ L do
9          multplypoly (t, f, mat, rowpos, mon_list)
10         rowpos⟵rowpos + 1
11     for i from id_length − 1 to 0 do
12         if mon_list[i].type = 1 and mon_list[i].appeartimes > 0 then
13             if there exists g ∈ G such that lm(g) | mon_list[i].mon then
14                 multplypoly (mon_list[i].mon/lm(g), g, mat, rowpos, mon_list)
15                 rowpos⟵rowpos + 1
16     mat.num⟵rowpos
17     return mat
   end
```

The algorithm SPI contains three steps.

In the first step (Lines 2–7), the algorithm SPI initializes some data, including the initial position of the row, and the members type, appeartimes, rowpos of each monomial node in the monomial list mon_list.

In the second step (Lines 8–10), the algorithm SPI computes the product of t and f from each pair in L. The product is computed by the procedure multiplypoly, which is given later, and the product tf is appended to the matrix mat.

In the last step (Lines 11–17), the algorithm SPI traverses all monomials in the monomial list to find monomials that appear in mat but are not leading monomials of polynomials. In this algorithm, for each monomial mon_list[i].mon, mon_list[i].appeartimes > 0 means mon_list[i].mon really appears in mat, and mon_list[i]. type $= 1$ means mon_list[i].mon is not a leading monomial of some polynomial. If the monomial mon_list[i]. mon meets the two conditions at Line 12, then the algorithm searches G to find a polynomial g satisfying conditions at Line 13. If such a polynomial g is found, then the product of mon_list[i]. mon/lm(g) and g is computed by the

procedure multiplypoly and appended to mat. Since the algorithm SPI traverses all monomials by a decreasing order on the monomial ordering, when adding a new polynomial (mon_list[i]. mon/lm(g))g to mat at Line 14, monomials bigger than mon_list[i]. mon are not affected.

In the algorithm SPI, the procedure multiplypoly(t, f, mat, rowpos, mon_list) computes the product of t and f, and append the product to mat at the row rowpos. Pseudo codes are given below.

Procedure `multiplypoly`(t, f, *mat, rowpos, mon_list*)

Input: t, a monomial in $\mathbb{B}_n$;
f, a polynomial in $\mathbb{B}_n$;
mat, a matrix of type matrix_t;
rowpos, an integer shows a position of row in mat;
mon_list, an array of monnode_t.

```
begin
    nextpos⟵mat.startpos[rowpos]
    if gcd(t, lm(f)) = 1 then
        id⟵getID(tlm(f))
        mon_list[id].type⟵2
    for each m in M(f) do
        id⟵getID(tm)
        if mon_list[id].lastrowpos = rowpos then
            mon_list[id].lastrowpos⟵ − 1
            mat.ids[mon_list[id].lastpos]⟵ − 1
            mon_list[id].appeartimes⟵mon_list[id].appeartimes − 1
        else
            mon_list[id].lastrowpos⟵rowpos
            mat.ids[nextpos]⟵id
            mon_list[id].lastpos⟵nextpos
            nextpos⟵nextpos + 1
            mon_list[id].appeartimes⟵mon_list[id].appeartimes + 1
            if mon_list[id].type = 0 then
                mon_list[id].type⟵1
    mat.startpos[rowpos + 1]⟵nextpos
end
```

The procedure multiplypoly does two things.

First, if $t\mathrm{lm}(f)$ is the leading monomial of tf, then the procedure multiplypoly sets the type of the monomial $t\mathrm{lm}(f)$ to be 2. In the algorithm SPI, if the type of a monomial is set to 2, it will never change until the algorithm is over. Please note that if $\gcd(t, \mathrm{lm}(f)) = 1$, then we always have $\mathrm{lm}(tf) = t\mathrm{lm}(f)$, but the converse is not always true in Boolean polynomial rings.

Second, for each monomial m appearing in f, the procedure multiplypoly computes the product tm, and then check whether the monomial tm has already appeared in the row rowpos in mat, i.e., check whether tm will appear in the same row twice (or for even times). If tm has already appeared at the row rowpos, then the latest appearing record of tm is cleared at Line 10; otherwise, the identity number of tm is appended to mat.

A key subroutine in the procedure multiplypoly is the function getID, which gets the identity number of a given monomial. We will discuss this function in the next subsection.

Please note that when the algorithm SPI is over, there may exist many -1's in the array ids of mat. So when reading mat, such -1's should be skipped directly.

3.4 Getting Identity Numbers of Given Monomials

In this subsection, we present our method of getting identity numbers of given monomials.

A simple method of getting the identity number of a given monomial m, is to search m in the monomial list. However, no matter which searching algorithm is used, this searching procedure will cost $O(\log(\text{length}))$ comparisons of monomials, where length is the length of the monomial list. After testing many examples, we find this searching method is not very efficient, particularly for complicated examples.

Another method of getting the identity number of a given monomial is through a multiplication table suggested by Cabarcas[12]. That is, for any monomial represented by its identity number and any variable, we can store the identity number of the product in a table in advance. By this method, getID becomes a memory accessing process which is much faster than a searching process. In our tests, we find this method is very efficient, but the main drawback of this method is that the multiplication table will cost too much memory when the number of variables is large.

In our implementation, the identity number of a given monomial is obtained through a hash table. Specifically, let table be an array of integers, table_size be the length of table, and ϕ be a mapping from monomials to integers, then for each monomial m in the monomial list, we set the value table[$\phi(m) \bmod \text{table\_size}$] to the identity number of m. The values of table should be updated when the monomial list is enlarged. In our implementation, chaining is used for collision resolution[13]. By this hash table, for any given monomial m, we can compute the value of $\phi(m) \bmod \text{table\_size}$ first, and then read the identity number of m from table directly. Experimental results in Section 4 show this method is very efficient.

For the hash function ϕ, there may be many choices. In our implementation, we use the following function. Let $x^\alpha = x_{i_0}x_{i_1}\cdots x_{i_l} \neq 1$, where $1 \leq i_0 < i_1 < \cdots < i_l \leq n$, then we set

$$\phi(x_{i_0}x_{i_1}\cdots x_{i_l}) = i_0 + i_1 \cdot (n+1) + \cdots + i_l \cdot (n+1)^l.$$

Then $\phi(x^\alpha) \geq 1$ for any $x^\alpha \neq 1$, and we define $\phi(1) = 0$. The idea of this hash function is to regard $(i_0, i_1, \cdots, i_l)$ as an $(n+1)$-adic number. So we have $\phi(x^\alpha) \neq \phi(x^\beta)$ whenever $x^\alpha \neq x^\beta$. But please note that $\phi(x^\alpha) \equiv \phi(x^\beta) \bmod \text{table\_size}$ may hold for $x^\alpha \neq x^\beta$.

Since the time complexity of the above method is dominated by the time of accessing table, the size of table is usually not set very large.

The above procedure of getting identity numbers for monomials can be speeded up significantly by avoiding duplicated hash procedure of monomials. That is, if there are several polynomials $p_1, p_2, \cdots, p_l$ that share many monomials, then instead of getting identity numbers for $M(p_1)$, $M(p_2)$, $\cdots$, $M(p_l)$ separately, we can get identity numbers for $M(p_1, p_2, \cdots, p_l)$ one time. This method avoids many duplicated hash procedure, and makes our implementation of F4 (given in Section 4), which uses the algorithm SPI for constructing matrices, very efficient.

3.5 A Toy Example

In this subsection, we give a toy example to illustrate how our method works.

Example 3.1 Let $\mathbb{B}_4 := \mathbb{F}_2[x_1, x_2, x_3, x_4]/\langle x_1^2 + x_1, \cdots, x_4^2 + x_4\rangle$ be a Boolean polynomial ring in $\{x_1, x_2, x_3, x_4\}$, $G = \{f_1, f_2\}$ be a subset of $\mathbb{B}_4$, where $f_1 = x_1x_2 + x_3x_4 + x_2 + x_4$ and $f_2 = x_2x_3 + x_2$, and $L := \{(x_3, f_1), (x_1, f_2)\}$. The monomial ordering is the Graded Reverse Lexicographical ordering with $x_1 > x_2 > x_3 > x_4$.

Next, we construct a matrix from G and L by the algorithm SPI.

The initial monomial list used in this example is

id	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
mon	1	x_4	x_3	x_2	x_1	x_3x_4	x_2x_4	x_1x_4	x_2x_3	x_1x_3	x_1x_2	$x_2x_3x_4$	$x_1x_3x_4$	$x_1x_2x_4$	$x_1x_2x_3$
type	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
appeartimes	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
lastrowpos	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1
lastpos	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1) The output mat is empty initially. After appending x_3f_1 to mat, we get

startpos	0	3	
ids	14	−1	8

The monomial x_3x_4 vanishes because it appears twice, and the monomial list becomes

id	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
mon	1	x_4	x_3	x_2	x_1	x_3x_4	x_2x_4	x_1x_4	x_2x_3	x_1x_3	x_1x_2	$x_2x_3x_4$	$x_1x_3x_4$	$x_1x_2x_4$	$x_1x_2x_3$
type	0	0	0	0	0	1	0	0	1	0	0	0	0	0	2
appeartimes	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1
lastrowpos	−1	−1	−1	−1	−1	−1	−1	−1	0	−1	−1	−1	−1	−1	0
lastpos	0	0	0	0	0	1	0	0	2	0	0	0	0	0	0

2) After appending $x_1 f_2$ to mat, we get

startpos	0	3	5		
ids	14	−1	8	14	10

The monomial list becomes

id	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
mon	1	x_4	x_3	x_2	x_1	x_3x_4	x_2x_4	x_1x_4	x_2x_3	x_1x_3	x_1x_2	$x_2x_3x_4$	$x_1x_3x_4$	$x_1x_2x_4$	$x_1x_2x_3$
type	0	0	0	0	0	1	0	0	1	0	1	0	0	0	2
appeartimes	0	0	0	0	0	0	0	0	1	0	1	0	0	0	2
lastrowpos	−1	−1	−1	−1	−1	−1	−1	−1	0	−1	1	−1	−1	−1	1
lastpos	0	0	0	0	0	1	0	0	2	0	4	0	0	0	3

3) Since the type of x_1x_2 is 1 and appeartimes = 1, the polynomial f_1 is found and appended to mat and we get

startpos	0	3	5	9					
ids	14	−1	8	14	10	10	5	3	1

The monomial list becomes

id	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
mon	1	x_4	x_3	x_2	x_1	x_3x_4	x_2x_4	x_1x_4	x_2x_3	x_1x_3	x_1x_2	$x_2x_3x_4$	$x_1x_3x_4$	$x_1x_2x_4$	$x_1x_2x_3$
type	0	1	0	1	0	1	0	0	1	0	2	0	0	0	2
appeartimes	0	1	0	1	0	1	0	0	1	0	2	0	0	0	2
lastrowpos	−1	2	−1	2	−1	2	−1	−1	0	−1	2	−1	−1	−1	1
lastpos	0	8	0	7	0	6	0	0	2	0	5	0	0	0	3

4) Similarly, since the type of x_2x_3 is 1 and appeartimes = 1, the polynomial f_2 is found and appended to mat and we get

startpos	0	3	5	9	11						
ids	14	−1	8	14	10	10	5	3	1	8	3

The monomial list becomes

id	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
mon	1	x_4	x_3	x_2	x_1	x_3x_4	x_2x_4	x_1x_4	x_2x_3	x_1x_3	x_1x_2	$x_2x_3x_4$	$x_1x_3x_4$	$x_1x_2x_4$	$x_1x_2x_3$
type	0	1	0	1	0	1	0	0	2	0	2	0	0	0	2
appeartimes	0	1	0	2	0	1	0	0	2	0	2	0	0	0	2
lastrowpos	−1	2	−1	3	−1	2	−1	−1	3	−1	2	−1	−1	−1	1
lastpos	0	8	0	10	0	6	0	0	9	0	5	0	0	0	3

So far, no more polynomials will be appended to mat, and mat can be eliminated by linear algebra.

4 Experimental Data

We implemented the F4 algorithm (called New-F4) over Boolean polynomial rings in C++ by using the proposed method of constructing matrices. The method of splitting matrices given in [14] is used, and the library M4RI (version 20130416)[15] is used for the eliminations of matrices. The monomial ordering is the Graded Reverse Lexicographic ordering.

We tested our implementation for square polynomial systems generated by Courtois[16] (Exa. "$n \times n$"), and HFE systems from [17]. In the examples generated by Courtois, $n \times n$ means that the input square polynomial system has n polynomials with n variables. Our computation platform is a MacBook Pro with 2.6 GHz Intel Core i7 CPU and 16 GB memory.

In Table 1, we list the size and density of the maximal matrices generated during the computations of Gröbner bases. We compare the time of constructing matrices (Const. time) with the total time of computing the whole Gröbner bases (Total time), and the ratios are also presented (Const./Total).

From Table 1, we can see that the ratio of constructing time and total time is very low, this show that the major time of computing a Gröbner basis is spent on eliminating matrices, which implies our method of constructing matrices is very efficient.

Table 1 Constructing time and total time

Exa.	Max matrix	Density	Const. time	Total time	Const./Total
20×20	23160×15548	3.90%	0.026	0.614	4.2%
21×21	27924×20791	3.67%	0.033	1.310	2.52%
22×22	63636×21870	3.40%	0.051	4.215	1.20%
23×23	41807×28951	4.40%	0.051	7.622	0.67%
24×24	241087×107518	2.08%	0.499	70.736	0.70%
25×25	290361×143449	2.25%	0.658	162.113	0.41%
26×26	347090×190355	2.31%	0.851	365.672	0.23%
27×27	409930×246525	2.36%	1.095	737.831	0.15%
28×28	489311×319837	2.62%	1.419	1582.947	0.09%
HFE_25_96	12479×13680	7.1%	0.012	0.748	1.60%
HFE_30_96	19988×29404	6.5%	0.024	3.622	0.66%
HFE_35_96	30081×55912	5.87%	0.047	12.854	0.37%

The densities of matrices reflect the number of monomials that are involved in constructing the maximal matrices. The value of the density is affected significantly by the algorithm used for computing Gröbner basis. Since many polynomials may have the same leading monomial,

choosing which polynomial to construct the matrices will affect the density of the matrix. According to our experiments, polynomials that are generated later usually have more monomials than the polynomials that are generated earlier, so choosing these lately generated polynomials for constructing matrices may make the matrices denser. But the monomials in the polynomials that are generated later are usually relatively smaller in the monomial ordering, since these polynomials have probably been reduced more times. So using polynomials that are generated later may save some time for eliminating matrices. We tested several strategies for choosing polynomials in our implementation, and find there is no much difference in the timings. We think this is because we use linear techniques of dense matrices for eliminations, and the timings are not so sensitive to the densities of matrices.

We also test our implementation with intrinsic Gröbner basis functions on Maple (version 17, setting "method = fgb"), Singular (version 3-1-6), and Magma (version 2.20-3) for solving the above systems, and the computing times in seconds are listed in Tables 2 and 3.

Table 2 Maple, Singular and Magma vs New-F4

Exam.	Maple	Singular	Magma	New-F4
16×16	4.088	5.210	0.130	0.066
17×17	9.891	12.886	0.230	0.117
18×18	22.340	31.590	0.950	0.209
19×19	48.314	84.771	0.860	0.353
20×20	107.064	265.325	1.000	0.614
21×21	218.479	724.886	2.670	1.310
22×22	839.067	$>$ 1h	7.410	4.215
HFE_25_96	121.681	$>$ 1h	1.160	0.748
HFE_30_96	619.745	$>$ 1h	2.550	3.622
HFE_35_96	2229.239	$>$ 1h	6.950	12.854

Table 3 Magma vs New-F4

Exam.	23×23	24×24	25×25	26×26	27×27	28×28
Magma	15.630	100.600	139.100	306.570	560.150	1169.150
New-F4	7.622	70.736	162.113	365.672	737.831	1582.947

From the above tables, we can see that our implementation of F4 is very efficient when the examples have relative small size. We think this is because the efficient routines of dense matrices from package M4RI are used. As the examples become complicated, the matrices become sparser, and the dense matrix techniques will become not so efficient. So to improve the performance of our implementation for complicated examples, we believe sparse linear algebraic techniques must be used.

5 Conclusions

In this paper, we present a method of implementing the Symbolic Preprocessing function over Boolean polynomial rings in Gröbner basis algorithms using linear algebra. A monomial list is introduced to detect duplicated monomials in the products of monomials and polynomials. The monomial list can also record the statuses of monomials, i.e., whether a monomial appears in the matrix and whether a monomial is a leading monomial of some polynomial. When searching polynomials that are used to reduce others, the algorithm SPI only needs to traverse the monomial list once. Another crucial step in our method is to get the identity numbers of given monomials, and this step is done by using a hash table as well as some techniques for avoiding duplicated hash procedure. The experimental results show the proposed method is very efficient. In the future, we will try to improve our implementation of F4 by using sparse linear algebraic techniques.

References

[1] Buchberger B, Ein algorithmus zum auffinden der basiselemente des restklassenringes nach einem nulldimensionalen Polynomideal, PhD thesis, 1965.

[2] Lazard D, Gröbner bases, Gaussian elimination and resolution of systems of algebraic equations, *Proc. EUROCAL'83, Lect. Notes in Comp. Sci.*, 1983, **162**: 146–156.

[3] Faugère J C, A new effcient algorithm for computing Gröbner bases (F_4), *J. Pure Appl. Algebra*, 1999, **139**(1–3): 61–88.

[4] Courtois N, Klimov A, Patarin J, and Shamir A, Efficient algorithms for solving overdefined systems of multivariate polynomial equations, *Proc. of EUROCRYPT'00*, *Lect. Notes in Comp. Sci.*, 2000, **1807**: 392–407.

[5] Ding J, Buchmann J, Mohamed M S E, Mohamed W S A E, and Weinmann R P, Mutant XL, *Proc. SCC'08*, 2008, 16–22.

[6] Faugère J C, A new effcient algorithm for computing Gröbner bases without reduction to zero (F_5), *Proc. ISSAC'02*, ACM Press, 2002, 75–82, Revised version downloaded from fgbrs.lip6.fr/jcf/Publications/index.html.

[7] Albrecht M and Perry J, F4/5, Preprint, arXiv:1006.4933v2 [math.AC], 2010.

[8] Faugère J C and Rahmany S, Solving systems of polynomial equations with symmetries using SAGBI-Gröbner bases, *Proc. ISSAC'09*, ACM Press, New York, USA, 2009, 151–158.

[9] Gao S H, Volny F, and Wang M S, A new algorithm for computing Gröbner bases, *Cryptology ePrint Archive*, Report 2010/641, 2010.

[10] Sun Y, Lin D D, and Wang D K, An improvement over the GVW algorithm for inhomogeneous polynomial systems, Preprint arXiv:1404.1428, 2014.

[11] Faugère J C and Joux A, Algebraci cryptanalysis of Hidden Field Equation (HFE) cryptosystems using Gröbner bases, *Proc. Advances in Cryptology - CRYPTO* 2003, LNCS, Springer Berlin/Heidelberg, 2003, **2729**: 44–60.

[12] Cabarcas D, An implementation of Faugère's F4 algorithm for computing Gröbner bases, Thesis, 2010.

[13] Skiena S S, *The Algorithm Design Manual*, Second Edition, Springer, 2008.

[14] Faugère J C and Lachartre S, Parallel Gaussian elimination for Gröbner bases computations in finite fields, *Proc. PASCO* 2010, ACM Press, 2010, 89–97.

[15] Albrecht M and Bard G, The M4RI Library — Version 20130416, 2013, http://m4ri.sagemath.org.

[16] Courtois N, Benchmarking algebraic, logical and constraint solvers and study of selected hard problems, 2013, http://www.cryptosystem.net/aes/hardproblems.html.

[17] Steel A, Allan Steel's Gröbner basis timings page, 2004, http://magma.maths.usyd.edu.au/ allan/gb/.

[18] Li D, Liu J, Liu W, and Zheng L, GVW algorithm over principal ideal domains, *Journal of Systems Science and Complexity*, 2013, **26**(4): 619–633.