Computing Groebner Bases in the Boolean Setting with Applications to Counting

Anna Bernasconi
Dipartimento di Informatica, Università degli Studi di Pisa, Pisa (Italy).
e-mail: bernasco@di.unipi.it

Bruno Codenotti
Istituto di Matematica Computazionale del CNR,
Via S. Maria 46, 56126 Pisa (Italy)
e-mail: codenotti@imc.pi.cnr.it

Valentino Crespi
Dipartimento di Informatica, Università degli Studi di Milano, Milano (Italy).
e-mail: crespi@dsi.unimi.it

Giovanni Resta
Istituto di Matematica Computazionale del CNR,
Via S. Maria 46, 56126 Pisa (Italy)
e-mail: resta@imc.pi.cnr.it

ABSTRACT

We take advantage of the special structure of computations in $\mathcal{Z}_2$ to develop algorithms for the computation of *Groebner bases* and of the *Hilbert function* in the Boolean setting. Natural sources of applications for our algorithms are the counting problems. We focus, as a case study, on the computation of the permanent. To this regard, one good feature of the *Groebner approach* is that, unlike other general methods for the exact computation of the permanent, it is intrinsically sensitive to the structure of the specific input, and this makes it possible to use it in order to *recognize* and solve efficiently several easy instances.

1. Introduction

Certain computational problems can be conveniently rewritten in terms of systems of equations, so that their associated decision problem consists of asking for the existence of a solution to the system, whereas the corresponding counting problem consists of asking for the number of solutions. In this paper we build upon the above idea to develop a method for solving hard counting problems which consists of determining the number of solutions to a system of equations which describe the original problem. We compute the exact number of solutions by first developing an algorithm for the computation of a *Groebner basis* in the Boolean setting, and then, from such a basis, we actually compute the number of solutions, which results to be equal to the number of monomials that are not divisible by the *leading terms* of the polynomials in the basis. Informally, the intuition behind this *modus operandi* is that we first transform a system of polynomial equations into an equivalent one (the Groebner basis) with special properties, and then, by taking advantage of these properties, we more easily compute the number of solutions. Intuitively, this process can be viewed as a sort of generalization of triangularization techniques for linear systems, where the triangular form gives immediate information, e.g., on the rank and the determinant of the coefficient matrix, and thus on properties of the solutions.

This paper has three main goals:

- To present an optimized code for the computation of Groebner bases, when each variable x_i is restricted to the Boolean setting by the equation $x_i^2 - x_i = 0$. This code can be used in a variety of applications. As an example, [CEI96] suggests using *Groebner proofs* as an alternative to resolution for the construction of proof systems. The efficiency of such Groebner proofs crucially relies on the possibility of taking advantage of the special structure induced by the presence of the equation $x_i^2 - x_i = 0$, for each of the variables x_i.
- To show that the above outlined approach can sometimes be a viable alternative to existing methods for finding the exact number of solutions to counting problems. For example we show that our algorithms compute the permanent of some classes of matrices much faster than the best known algorithm, which is due to Ryser [R63]. While we do not claim that we can achieve good running times in general, we view our effort as a first step towards providing a unified and highly adaptive computational environment for counting problems. In fact, one good feature of our algorithms is that, unlike Ryser's method, they are intrinsically very sensitive to the structure of the specific input, and this makes it possible to solve efficiently several different classes of easy instances, without tailoring the computation to handle them in a specific way.
- To study properties of special permanents for whose computation it is still unclear whether or not there exist efficient algorithms. For example the best known algorithm to compute the permanent of an $n \times n$ *circulant matrix* with three ones per row takes $O(n2^{\frac{n}{2}})$ time [CCR96], roughly the square root of the time of Ryser method. The analysis of the rich structure of the Groebner bases associated with these matrices (see Section 5) might lead to better algorithms.

This paper provides an attempt to employing *computational algebraic geometry* techniques in the Boolean setting. The reader can see Bayer's thesis for a similar suggestion [BS82a]. The use of algebraic geometry tools in the Boolean domain has a recent history. For example, Smolensky has shown that the Hilbert function is responsible for certain lower bounds [Sm93]. As already mentioned, Clegg et al. have suggested that Groebner basis techniques might be possible alternatives to resolution in the construction of proof systems [CEI96].

The rest of the paper is organized as follows.

In Section 2 we describe our approach to the solution of counting problems with a special attention to the permanent computation. In Section 3 we give a high level description of the two main stages of our algorithms, i.e., the computation of a minimal Groebner basis and the computation of the number of solutions starting from the initial monomials of the Groebner basis. In Section 4 we describe some key implementation issues, focusing on the data structures used and on the peculiarities related to the Boolean setting. We also analyze the computational cost of the algorithms. In Section 5 we present the experimental results obtained and we compare the performance of our algorithms with that of Ryser method and of the Macaulay and CoCoA packages [BS82, CNR93].

2. The approach

For simplicity, we first describe our method in the case of the permanent computation. At the end of the section, we show how it can be applied to other counting and decision problems.

Let $A = (a_{ij})$ be a $(0,1)$-matrix. We define a matrix X whose (i,j)-th entry $(X)_{ij}$ is x_{ij} if $a_{ij} = 1$ and as 0 if $a_{ij} = 0$. We then consider the following system of $n^2 + 2n$ equations:

$$\begin{cases} \sum_{j=1}^{n} x_{i,j} = 1 & \text{for } i = 1, 2, \ldots, n \\ \sum_{i=1}^{n} x_{i,j} = 1 & \text{for } j = 1, 2, \ldots, n \\ x_{i,j}(1 - x_{i,j}) = 0 & \text{for } i, j = 1, 2, \ldots, n. \end{cases} \tag{1}$$

```
Algorithm 1
Inp : A set of polynomials F = {f_1, ..., f_k}.
Out : A minimal Groebner basis G for F.
1.  B ← {(i,j) | 1 ≤ i < j ≤ t}
2.  G ← F
3.  while B ≠ ∅ do
4.      select (i,j) ∈ B
5.      B ← B − {(i,j)}
6.      if Good(i,j) then
7.          R = S(f_i, f_j)^G
8.          if R ≠ 0 then
9.              t ← t + 1
10.             f_t ← R;
11.             G ← G ∪ {f_t};
12.             B ← B ∪ {(i,t) | 1 ≤ i < t}
13.         endif
14.     endif
15. endwhile
16. Minimize(G)
```

```
NumSol(M,V)
In  :  A set of multilinear monomials
       M = {m_1, m_2, ..., m_k}
       The set of variables
       V = {v_1, v_2, ..., v_n}.
Out :  The number of solutions in Z_2^n.
1. if      k = 0 return 2^n
2. elseif  k = 1 return 2^{n-|m_1|}(2^{|m_1|} - 1)
3. elseif ∃j : |m_j| = 1 return
              NumSol(M[v_j = 0], V - {v_j})
4. else
5.      select v_j ∈ V
6.      return NumSol(M[v_j = 0], V - {v_j}) +
                NumSol(M[v_j = 1], V - {v_j})
7. endif
```

Fact 2.1. *Let A be a $(0,1)$ $n \times n$ matrix and let X be the matrix constructed as described above. Then* $\mathrm{per}(A)$ *is equal to the number of solutions of the system of equations (1).*

Proof. The permanent of A is the number of its *nonzero permutations*, where a permutation σ is nonzero if $a_{k,\sigma(k)} = 1$, for $k = 1, 2, \ldots, n$. It is easy to see that each nonzero permutation σ uniquely corresponds to a solution of (1). We simply let $x_{i,j} = 1$ if $j = \sigma(i)$, and $x_{i,j} = 0$ otherwise. The converse is also true since the third equation restricts the range of the solutions to 0 and 1, and the other two equations select exactly one nonzero entry in each row and column of A. □

Fact 2.1 allows us to bring the computation of the permanent into the framework of algebraic geometry. Indeed it hints at computing the permanent via, e.g., Groebner basis and Hilbert function computation. More precisely, we proceed as follows. We first compute a Groebner basis for the ideal of polyonomials $\sum_{j=1}^{n} x_{i,j} - 1$, $\sum_{i=1}^{n} x_{i,j} - 1$, and $x_{i,j}(1 - x_{i,j})$. Then we compute the number of monomials that are not divisible by any of the leading terms of the Groebner basis, which is equal to the number of solutions of (1). The actual description of the algorithms will be provided in Section 3.

Note that this approach can be extended to other problems. For instance, to describe $3SAT$, in addition to the equations for the restriction of the variables to the Boolean domain, one needs equations of the type $x + y + z - xy - xz - yz + xyz = 1$. In the case of graph isomorphism, the goal is to find permutation matrices P such that $PA = BP$, where A and B are the adjacency matrices of two graphs. Then one can write equations in terms of a matrix X of variables for which one requires that $XA = BX$, in addition to the restrictions that guarantee that the row and column sums are both equal to one, and to the restrictions of the variables to the Boolean domain.

3. Algorithms

Our algorithms are divided into two main stages. We first compute, from the set of equations that describe the original problem, an equivalent set of equations which corresponds to a *minimal* Groebner basis, i.e. a basis with the minimum number of polynomials. Then we use the leading terms of the basis in order to compute the number of solutions of the original set of equations.

A Groebner basis algorithm for the Boolean setting. We have implemented four algorithms for the computation of a Groebner basis, all based on Buchberger method and its subsequent refinements [CLO92, GM88]. For their description, we need the following definitions. Given two polynomials, f and g, the *S-polynomial* of f and g is the polynomial $S(f,g) = \frac{h}{LT(f)} \cdot f - \frac{h}{LT(g)} \cdot g$, where h is the least common multiple of $LT(f)$ and $LT(g)$, i.e., the leading monomials of f and g. Let $F = (f_1, f_2, \ldots, f_t)$ be a t-tuple of polynomials in $k[x_1, \ldots, x_n]$, ordered according to a fixed monomial ordering. Then every $g \in k[x_1, \ldots, x_n]$ can be expressed as $g = r + \sum_{i=1}^{t} a_i f_i$, where $a_i, r \in k[x_1, \ldots, x_n]$, and either $r = 0$ or none of the monomials of r is divisible by any of the monomials $LT(f_1), \ldots, LT(f_t)$. We call r a *remainder* of g on division by F, and we denote it with $r = \overline{g}^F$.

Buchberger algorithm incrementally computes a Groebner basis G of a set of polynomials $F = \{f_1, f_2, \ldots, f_k\}$. Initially we set $G = F$. Subsequently, we compute, for each pair of polynomials $\{f_p, f_q\} \in G$, $f_p \neq f_q$, the remainder $R = \overline{S(f_p, f_q)}^G$. If $R \neq 0$, then R is added to the basis G. This procedure is iterated until $\overline{S(f_p, f_q)}^G = 0$, for each pair of polynomials in G. The resulting set G is a (non minimal) Groebner basis for F. This basic version of the algorithm is very inefficient, because it takes into account a very large number of pairs of polynomials. Building upon the suggestions reported in [CLO92], we have thus implemented the following version of the algorithm, which is the first of our four algorithms (Algorithm 1).

The criterium `Good(`i, j`)` adopted at line 6 of Algorithm 1 allows us to avoid the computation of $R = \overline{S(f_i, f_j)}^G$ for a non negligible number of pairs (i, j), and is defined as follows:

$$\texttt{Good}(i,j) = \begin{cases} \texttt{False} & \text{if } \mathrm{lcm}(LT(f_i), LT(f_j)) = LT(f_i)\, LT(f_j) \\ \texttt{False} & \text{if } \exists k \notin \{i,j\} : (i,k) \notin B\ (j,k) \notin B \text{ and} \\ & \quad LT(f_k) \text{ does not divide } \mathrm{lcm}(LT(f_i), LT(f_j)) \\ \texttt{True} & \text{elsewhere} \end{cases}$$

After a Groebner basis G has been computed, we *minimize* it (line 16), simply by deleting all the polynomials whose leading term is a multiple of the leading term of another polynomial in the basis. We thus obtain a minimal basis, which in general is not unique. Note that the minimality of the basis is sufficient for our purposes, since in the subsequent computation of the number of solutions we just need the leading terms of G. Algorithm 1 selects a pair from B (line 4) according to a simple LIFO strategy.

We have developed another algorithm (Algorithm 2), which implements Buchberger suggestion to select from B the pair (i, j) for which the degree of $\mathrm{lcm}(LT(f_i), LT(f_j))$ is minimum.

Algorithms 1 and 2 often introduce (unnecessarily) large sets B. This is due to the fact that both algorithms perform insertion in the set B without any check. Gebauer and Möller [GM88] describe a different, more complicated, criterion that can be applied at the time of insertion, and which allows us to avoid all the checks after selection. We have implementated this criterion in Algorithm 3. We have also implemented another version of Buchberger method (Algorithm 4). This consists of the following minor modification of Algorithm 3: via preliminary reductions and sorting of the input polynomials F, we maintain in a minimal form the incrementally built basis. For Algorithm 4 the final step of minimization is thus not necessary any more.

Computation of the number of solutions. Let $M = \{m_1, \ldots, m_k\}$ be the set of leading monomials of a minimal Groebner basis G, defined in terms of the variables $V = \{v_1, \ldots, v_n\}$, and let M' be the set of the multilinear monomials of M. The value of the permanent is equal to the cardinality of the set $\overline{M}$ of the monomials that are not divisible by any $m_i \in M$. We claim that this number is equal to the number of $\{0,1\}$ solutions of the system obtained equating to zero all the monomials in M'. In fact, by construction, M contains only multilinear monomials and monomials of the form v_i^2. Since $\overline{M}$ is finite, for every $v_i \in V$, either v_i or v_i^2 belong to M, and thus all the

monomials in $\overline{M}$ are multilinear. Hence $\overline{M}$ is also equal to the set of the multilinear monomials that are not divisible by any of the monomials of M'. Given an assigment $S = \{v_i \leftarrow \alpha_i\}_{i=1,\ldots,n}$, with $\alpha_i \in \{0,1\}$, we claim that S is a solution of the system $M' = 0$ if and only if the monomial $m_S = \prod_{i=1}^{n} v_i^{\alpha_i}$ belongs to $\overline{M}$. In fact, if S is a solution, then any given $m \in M'$ contains at least a variable v which takes the value zero in S, and thus, by definition, does not appear in m_S, which thus cannot be divisible by m. Since this holds for any $m \in M'$, we obtain that $m_S \in \overline{M}$. The reverse implication follows in a similar way.

To evaluate the number of solutions of M' we have thus adopted the recursive algorithm described beside Algorithm 1. We denote by $|m|$ the degree of a monomial and by $M[v = \alpha]$ the set of monomials obtained from M by setting $v = \alpha$, for $\alpha \in \{0,1\}$.

In the current version of the algorithm, at line 5 we select the variable which appears more frequently in the monomials of M with smallest degree. Other (more refined) strategies could be used for very large size problems.

4. Implementation Issues

In this section we describe the most important features of our implementations, expecially focusing on the specialization of computations to the Boolean domain. The algorithms have been implemented in the C language, the code compiled with the GNU GCC compiler, and the experiments carried out on a SUN Superspark 20 workstation.

Computation of Groebner Bases. The polynomials that occur in the execution of algorithms 1, 2, 3, and 4 are either of the form $x_i^2 - x_i$, or multilinear. In fact, if a term of the form x_i^k, $k > 1$, appears in a polynomial (different from $x_i^2 - x_i$), then it can be immediately simplified to x_i, since the constraint $x_i^2 - x_i = 0$ implies that $x_i^k = x_i$ for every $k > 1$. This observation led us to choose (in the implementation of all the algorithms) the following representation for monomials and polynomials. A polynomial is represented by a record containing the number of its monomials (the 'length' of the polynomial) and a pointer to an array of monomials. The polynomials of the form $x_i^2 - x_i$ are encoded as polynomials of length -1, and with only a monomial representing x_i. If at most n variables are used, a monomial is identified with a record containing the monomial coefficient and a vector of n bits that represents the exponents of the variables contained in the monomial. Given the nature of the problems at hand, we can restrict the range of the coefficients to $\mathcal{Z}_p$, where p is a prime number greater than of equal to the maximum number of variables that might appear in one equation (for the permanent computation, this is the maximum number of ones appearing in a row or column of the input matrix). We take advantage of the fact that monomials are square-free in order to use only one bit per variable. Although for a large number of variables this representation might waste some space, it has the desirable feature to allow the implementation of all the needed monomial computations as fast bit-wise logical operations. For example, monomial multiplication (as well as the computation of lcm), can be realized with a bit-wise OR, while division can be realized with a bit-wise XOR (exclusive or), once divisibility has been tested. To speed up the execution of these operations, we have arranged bits in larger sets that can be treated as single quantities by the compiler on the target machine. In our implementation we perform logical operations between `unsigned long integer` quantities, that are represented with 32 bits. Thus the multiplication of two monomials needs $\lceil n/32 \rceil$ OR operations on 32-bit integers, independently of the actual number of variables in the monomials, plus a fixed amount of work for the multiplication of the coefficients. Note that, according to this representation, we have, e.g., $xy \cdot yz = xyz$. This is correct since all the variables are restricted to the Boolean domain. In addition, with this representation we achieve a very fast implementation of comparison between monomials. (These comparisons must be performed in order to sort monomials according to, e.g., the lexicographic monomial ordering.) Indeed at most $\lceil n/32 \rceil$ comparisons between 32-bit unsigned integers are sufficient to decide the ordering relation

between two monomials.

The choice of adopting a special representation for the quadratic polynomials of the form $x_i^2 - x_i$, does not cause any problem, because these polynomials just belong to the initial sets of polynomials, and can not be generated during the computation of G. Accordingly, our monomial multiplication algorithm outputs only multilinear polynomials. Quadratic polynomials can mix with multilinear ones only in the computation of the S-polynomial $S(f_i, f_j)$, when either f_i or f_j is of the form $x^2 - x$. However the S-polynomial has the property of annihilating the leading terms of f_i and f_j, and thus it can be computed without actually multiplying the quadratic term. The case when both f_i and f_j are quadratic, and thus do not share any variable, is avoided *a priori* by appropriate criteria used by the algorithms. In the subsequent division of $S(f_i, f_j)$ by G, the way in which we implement the operations between monomials always leads to square-free partial results, and thus the quadratic polynomials contained in G have not to be taken into account as possible divisors. The 1-bit representation is also compatible with the criterion (`Good()`) adopted by Algorithms 1 and 2. The criteria used in Algorithms 3 and 4 are more complicated and need a temporary representation of monomials with 2 bits per variable. The set of pairs B is implemented in different ways in the four algorithms. In Algorithm 1, B behaves like a *stack* and it consists of a simple array of pairs of numbers, each related to an element of G. The selection of a pair (line 4) is a Pop operation, while the insertion of new pairs (line 12) consists of several Push operations. The sequence of Pop and Push operations maintains the stack sorted with respect to an ordering of pairs, and this is exploited in the implementation of the criterion `Good()`, where we test for the existence of a certain pair in B in logarithmic time using a binary search subroutine.

In Algorithm 2, B is implemented at the same time as a priority Heap and as a Red-Black Tree. The Heap allows us to select the pair (i, j) that minimizes the degree of $\mathrm{lcm}(LT(f_i), LT(f_j))$, while the Tree allows us to perform a fast existence test. For both structures, insertion and deletion (as well as search, for the Tree) can be done in logarithmic time.

Algorithms 3 and 4, which operate according to different criteria, do not need to test if a certain pair (i, j) belongs to B. For this reason, B is simply implemented as a stack. It is worth reporting that, while in our typical runs of Algorithms 1 and 2 the size of B can exceed, say, 30,000, in Algorithms 3 and 4 it is almost always less than 100.

Computation of the number of solutions. The implementation of algorithm `NumSol(`M`,`V`)` does not need to represent polynomials, and uses a different, simpler than above, representation of monomials, which are always multilinear, and whose coefficients are always equal to 1, and thus can be disregarded. In the following we assume that there are at most 255 variables, numbered as $v_1, v_2, \ldots$. A monomial m can thus be represented by the string of the subscripts of its variables. This string, actually an array of bytes, is terminated with a null byte, according to the conventions of the C language. A set (M,V) is implemented as a record containing a string which represents the set of variables V, an integer containing the number of monomials in M, and an array of strings representing M. The set $M[v_i = 0]$ is obtained from M deleting all the monomials (i.e., strings) that contain v_i, while the set $M[v_i = 1]$ is obtained from M deleting the characters corresponding to v_i from its strings. Since `NumSol(`M`,`V`)` can reach very deep levels of recursion, it has been implemented in a non-recursive fashion, using a stack.

Cost Analysis. All the monomial operations have a cost proportional to the number of variables n_v defining the problem. This follows from the fact that each monomial is represented by a 16-bit integer for the coefficient and $\lceil \frac{n_v}{32} \rceil$ 32-bit integers for the exponents of the variables (see above). The cost of the simple polynomial operations that are employed by the four algorithms, namely addition, subtraction, and multiplication by a monomial, have a cost roughly proportional to $n_v l_p \log l_p$, where l_p is the maximum number of monomials in the polynomials that occur during the computation. The logarithmic factor $\log l_p$ arises from the execution of a sorting stage (needed for simplification purposes) after each polynomial computation. The most expensive operation performed by the

algorithms is the computation of the remainder R of the division of an S-polynomial $S(f,g)$ by the polynomials already in the basis G. Let n_G and n_B be the cardinalities of the sets G and B, respectively. At each step of the division, the algorithm scans the n_G elements of G looking for a polynomial whose leading term divides the leading term $LT(R')$ of the current remainder R' (which initially coincides with $S(f,g)$). If the search is unsuccessful, then $LT(R')$ is deleted from R', and appended to R. If there exists $h \in G$ such that $LT(h)$ divides $LT(R')$, then R' is updated according to the rule $R' \leftarrow R' - \frac{LT(R')}{LT(h)}h$. The process terminates when $R' = 0$. The overall time needed to carry out one step of the division algorithm is thus proportional to $n_v n_G + n_v l_p \log l_p$.

Our four algorithms differ for the criteria adopted to execute the following two steps: (i) update the set B (to be performed each time a new element of the basis is found); (ii) select from B a pair to form a new S-polynomial.

For Algorithms 1 and 2, the selection from B and the check of the `Good()` criterion, have a cost proportional to $n_v n_B \log n_B$. The cost of insertion is proportional to n_G for Algorithm 1, and to $n_G n_v \log n_B$ for Algorithm 2, since in the latter case the n_G new pairs are inserted into two dynamic structures (a heap and a tree). For Algorithm 3 and 4, the selection from B has unitary cost since all the checks are performed when B is updated. The criteria used to decide the insertion of new pairs have a cost roughly proportional to $n_v(n_G^2 + n_B)$. The Minimization step at the end of Algorithms 1, 2 and 3, and the Reduction step at the beginning of Algorithm 4, have generally a negligible cost with respect to the rest of the computation. The four algorithms have a different behaviour for what concerns the growth of the quantities n_G, n_B, and l_p, during their execution. Experimentally we observed that Algorithm 1 produces larger bases G (i.e., there are many extra elements, to be deleted during the Minimization step) and executes more divisions than the other algorithms. However it has the best rate of growth for l_p. Algorithm 2 often requires a number of divisions which is 10 times less than the other algorithms, but it also builds sets B 10 times larger than those of Algorithm 1. Furthermore it generates very long polynomials. Algorithms 3 and 4 maintain a much smaller set B, but the cost to manage it is greater than for Algorithms 1 and 2. Algorithm 4 generates small sets G, since it maintains them in minimal form, but, as Algorithm 2, produces a fast growth of l_p. As we will see in Section 5, it turns out that Algorithms 2 and 4 exploit better than the other two algorithms certain structural properties of the matrices. In particular, in the case of circulant matrices of the form $I + P^i + P^j$, Algorithms 2 and 4 are significantly faster either when the matrix is symmetric or when i, j, and the size of the matrix have a large common factor.

5. Experimental Results

Given the fact that the permanent of a $(0,1)$ matrix with at most three ones per row and columns keeps all the difficulty of the general problem [DLMV88], we have decided to tailor our investigation to this kind of matrices. We report the outcomes of experiments on the following classes of matrices, all with at most three ones per row: circulant matrices of the type $I + P + P^j$, random symmetric matrices with ones on the main diagonal, random matrices with ones on the main diagonal, random Hessenberg matrices, and block circulant matrices. The selection of these case studies has been made in order to test our approach against a wide range of *difficulties*. In fact the above classes include examples of convertible matrices, of very structured matrices that nevertheless are not known to have "easy permanents", and of matrices whose permanents are as hard to compute as in the general case.

Circulant Matrices. Despite the fact that the permanent of circulant matrices have been widely investigated, and several of its algebraic properties discovered, there are no clear indications of whether or not it can be computed efficiently, even for matrices with only three ones per row. We have gathered evidence that the methodology employed in this paper can be a valuable tool for investigations on these classes of matrices. In fact, the results of Figure 1 closely reflect known

j	Mac.	H. f.	CoCoA	Alg. 1	Alg. 3	N.S.
2	7.0	2.1	5.4	0.3	0.5	0.02
3	129.4	2.4	67.2	29.9	30.7	0.04
4	400.6	3.2	134.1	229.0	188.3	0.09
5	205.3	3.3	74.5	237.8	171.1	0.09
6	575.7	3.5	242.3	417.5	276.3	0.10
7	240.1	3.4	142.3	279.0	260.9	0.10
8	3.5	2.2	5.6	0.3	0.5	0.03
9	152.0	3.4	94.9	341.9	161.9	0.09
10	536.1	3.6	209.5	550.7	268.4	0.11
11	174.2	3.3	64.5	361.6	172.5	0.08
12	292.9	3.3	121.4	360.7	238.1	0.09
13	155.1	2.4	94.5	124.9	148.9	0.06
14	12.6	2.4	6.2	0.4	0.7	0.02

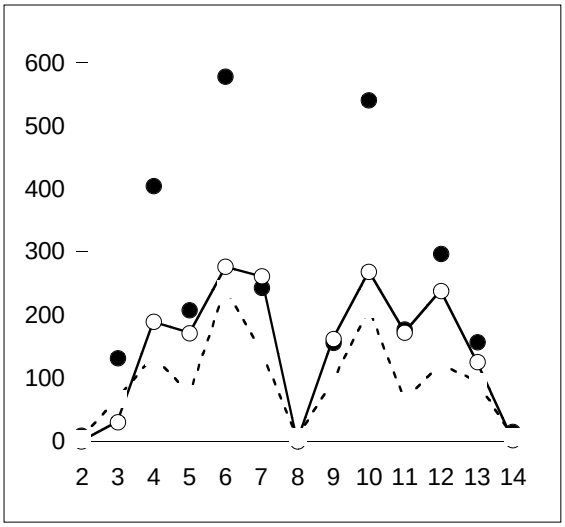

Figure 1: Computation of the permanent of $I+P+P^j$ for $n = 15$, as j varies. The Table on the left reports the time performance (in seconds) for CoCoA and Macaulay packages (Groebner basis and Hilbert function) and for our Algorithms 1 and 3. The last column gives the time performance of the computation of the number of solutions, after Algorithm 1 and/or 3 provided a Groebner basis. We used reversed lexicographic monomial ordering. The graph on the right shows the time performance as a function of j, visualizing the best time achieved by our algorithms (white circle), and the time spent by Macaulay (black circle) and CoCoA (cross).

algebraic differences between matrices of the type $I + P + P^j$, as j varies. Roughly speaking, the permanent turns out to be particularly easy, e.g., either for j very small or for j close to $\frac{n}{2}$ (see [Mi87, CCR96]). In particular, the permanent of the matrix $I+P+P^2$ has a very simple expression, and two of our algorithms, as well as the package Macaulay [BS82], clearly detect it, by running in (observed) linear time. In particular, one of our algorithms is faster than Ryser algorithm, for any $n \geq 20$ (see Table 1).

We have been able to obtain further enhancements in the performance of our algorithms by adding to the set of equations that define the problem a number of redundant equations that do not change the solutions, but are useful to speed-up the computation since they belong to the minimal Groebner basis. Thus, we have implemented a more efficient algorithm, Algorithm 5, that consists in a modification of Algorithm 3, obtained by adding to the input of the problem, such a set of equations (see Figure 2). We think that further pursuing this idea could be fruitful.

Other test matrices. Also for the other test cases, the time performance of the algorithms (expecially algorithms 2 and 4) is widely influenced by the structure of the problems. In particular, it is nice to see that algorithms 2 and 4 exploit the structure of symmetric matrices with ones on the main diagonal, which are known to be easy (see Table 2). Tables and Figures show also that our algorithms often provide a substantial improvement over the packages CoCoa and Macaulay.

Other packages. We have started to test two other computer algebra systems, namely *Macaulay2* and *Singular*. Some preliminary experiments show that Singular outperforms Macaulay and CoCoA on matrices of the form $I + P + P^2$, while in the other cases the results can vary.

j	Alg. 3	Alg. 5
2	1.1	0.9
3	117.3	124.6
4	651.7	627.8
5	1398.2	1168.3
6	168.1	140.3
7	3144.9	2471.2
8	451.1	369.9
9	0.9	0.9
10	329.1	189.4
11	3117.5	1716.1
12	348.2	115.1
13	2214.3	774.9
14	1458.1	623.9
15	609.6	159.8
16	1.1	0.8

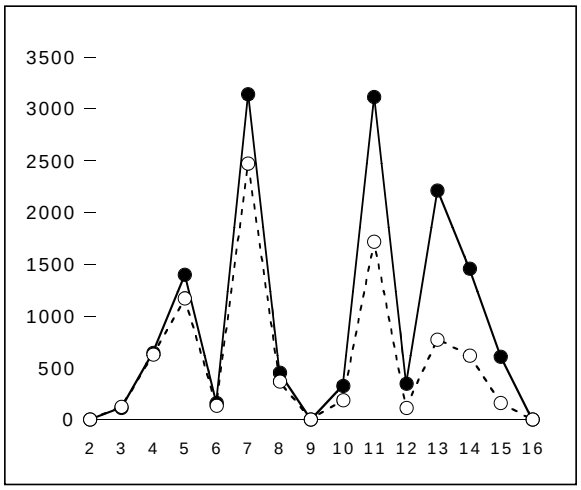

Figure 2: Computation of the permanent of $I + P + P^j$ for $n = 17$ as j varies. The Table on the left reports the time performance (in seconds) for the computation of Groebner bases and Hilbert functions for Alg. 3 and for Alg. 5. The graph on the right shows the time performance as a function of j, visualizing the best time achieved by Alg. 5 (white circle), w.r.t. the time spent by Alg. 3 (black circle).

References

[BS82a] D. Bayer. The Division Algorithm and the Hilbert Scheme. Ph.D. thesis, Harvard, (1982)

[BS82] D. Bayer and M. Stillman. Macaulay: A system for computation in algebraic geometry and commutative algebra (1982-1994). Available via ftp at math.harvard.edu.

n	Ryser	Mac.	H. f.	CoCoA	Alg. 1	Alg. 2	Alg. 3	Alg. 4	NumSol
10	0.01	1.6	0.38	1.83	0.1	0.2	0.1	0.1	0.01
12	0.02	3.1	0.68	2.76	0.2	1.1	0.3	0.2	0.01
14	0.05	5.3	1.43	4.02	0.3	1.0	0.4	0.3	0.02
16	0.18	8.7	3.36	5.77	0.4	1.6	0.7	0.5	0.03
18	0.72	13.7	*	8.34	0.6	6.1	1.0	0.7	0.04
20	2.96	20.4	*	12.79	0.8	4.6	1.3	1.1	0.07
22	12.23	29.8	*	19.42	1.1	6.5	2.2	1.7	0.13
24	50.98	43.1	*	30.48	1.5	44.6	3.0	2.5	0.21
26	255.66	57.8	*	50.02	1.9	43.2	3.9	3.6	0.35
28	17 min.	78.9	*	85.82	2.4	45.8	6.0	5.0	0.62
30	71 min.	104.2	*	154.36	3.1	42.0	6.6	6.7	1.08
32	5 hours	136.9	*	286.09	3.8	48.5	8.5	9.1	1.89
40	58 days	353.2	*	3188.00	8.8	314.1	21.8	*	17.63

Table 1: Computation of the permanent of $I + P + P^2$ for different values of n. The entries contain running times (in seconds). We filled an entry with * when the computation time exceeded a given bound.

	Non Symmetric + I			Symmetric + I			Hessenberg			Block Circulant		
n	8	12	16	10	20	40	30	40	50	4x5=20	6x6=36	5x9=45
Ryser	0.005	0.02	0.16	0.01	2.75	53 d.	1088	12 d.	36 y.	2.51	82 d.	4 y.
CoCoA	1.21	5.75	241	1.92	8.18	54.57	9.10	18.07	33.40	6.81	32.46	61.31
Alg. 1	0.18	43.57	-	3.79	-	-	0.26	0.54	0.91	59.69	-	-
Alg. 2	0.15	13.30	-	0.30	1.96	5.36	0.84	1.86	3.08	0.55	2.86	4.91
Alg. 3	0.16	16.34	-	0.91	-	-	0.58	1.14	1.35	115.18	-	-
Alg. 4	0.06	5.77	250	0.07	0.37	3.25	0.11	0.24	0.64	0.17	1.06	1.46
NumSol	0.01	0.02	0.10	0.01	0.07	10.12	0.01	0.01	0.01	0.06	0.47	5.43

Table 2: Computation of the permanent for different kinds of test matrices, for different values of n. The entries contain running times (in seconds). The entries with the symbol "-" correspond to instances for which the given algorithm exceeded a certain time bound. The running time of Ryser algorithm for large instances have been estimated.

[BW93] T. Becker and V. Weispfenning. Groebner Bases. *Springer-Verlag* (1993).

[BS95] R.A. Brualdi, and B.L. Shader. Matrices of sign-solvable linear systems. *Cambridge University Press* (1995).

[CNR93] A. Capani, G. Niesi, and L. Robbiano. CoCoA 3, a System for doing Computations in Commutative Algebra (1993-1996). Available via anonymous ftp from `lancelot.dima.unige.it`.

[CEI96] M. Clegg, J. Edmonds, and R. Impagliazzo. Using the Groebner Basis Algorithm to Find Proofs of Unsatisfiability. *Proc. 28th ACM Symp. on the Theory of Comput.* (1996).

[CCR96] B. Codenotti, V. Crespi, and G. Resta. On the Permanent of Certain $(0,1)$ Toeplitz Matrices. *Linear Algebra and Its Applications, to appear* (1996).

[CLO92] D. Cox, J. Little, D. O'Shea. Ideals, Varieties, and Algorithms. *Springer-Verlag, New York* (1992).

[DLMV88] P. Dagum, M. Luby, M. Mihail, and U. Vazirani. Polytopes, Permanents, and Graphs with Large Factors. *Proc. 27th IEEE Symp. on Found. of Comput. Sc.* (1988).

[FL92] U. Feige, and C. Lund. On the Hardness of Computing the Permanent of Random Matrices. *Proc. 24th ACM Symp. on the Theory of Comput.* 643-654 (1992).

[GM88] R. Gebauer, and H.M. Möller. On an Installation of Buchberger's Algorithm. *J. Symbolic Computation* 6:275-286 (1988).

[Mi87] H. Minc. Permanental Compounds and Permanents of $(0,1)$ Circulants. *Linear Algebra and its Appl.* 86:11-42 (1987).

[R63] H.J. Ryser. Combinatorial Mathematics. *Carus Mathematical Monograph* No. 14 (1963).

[Sm93] R. Smolensky. On Representations by Low-Degree Polynomials. Proc. of the 34^{th} IEEE Symposium on the Foundations of Computer Science, pp. 130-138, 1993.

[Va79] L.G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science* 8:189-201 (1979).