Computing and Informatics, Vol. 31, 2012, 1329–1344

COMPUTING INDEXES AND PERIODS OF ALL BOOLEAN MATRICES UP TO DIMENSION N = 8

Pavol Zajac, Matúš Jókay

Institute of Computer Science and Mathematics
FEI STU
Ilkovičova 3
812 19 Bratislava, Slovakia
e-mail: {pavol.zajac, matus.jokay}@stuba.sk

Communicated by Marian Vajteršic

Abstract. A set of $n \times n$ Boolean matrices along with the Boolean matrix multiplication operation form a semigroup. For each matrix A it is possible to find index μ and period λ, such that $A^\mu = A^{\mu+\lambda}$, and μ, λ are the smallest positive integers with this property. We are concerned with a question: How many $n \times n$ Boolean matrices have the given index, and period? A new algorithm is presented that was used to compute index and period statistics of all square Boolean matrices up to $n = 8$. Computed statistics are presented in the appendix of the paper.

Keywords: Boolean matrix, enumeration, directed graphs

1 INTRODUCTION

A Boolean matrix is a matrix with only $\{0, 1\}$-entries viewed as elements of Boolean algebra. We can define multiplication of such matrices similar to a classical matrix multiplication using logical OR as addition, and logical AND as multiplication, respectively. We can also intuitively define powers of (square) Boolean matrices, i.e., A^n is n-times multiplied matrix A with itself.

Boolean matrices and Boolean matrix multiplication have a wide range of applications. In general, they can represent binary relations on finite sets, and the Boolean matrix product is used to compute the composition of relations. (Simple) graphs can be represented by Boolean adjacency matrices, where the element a_{ij}

is 1 iff there is an (oriented) edge between vertices v_i, and v_j, respectively. Moreover, powers of adjacency matrix are related to powers of graphs, more specifically, Boolean power $B = A^i$ of adjacency matrix A has $b_{jk} = 1$ iff there exists a path with the length exactly i between vertices v_j, v_k.

It is known that when we compute successive powers of some Boolean matrix, the sequence starts to repeat itself at some point, i.e., we will get $A^m = A^{m+l}$ for some m, l. That is, the minimal m is called the *index*, and l the *period* of matrix A. It is not difficult to compute index and period of a single Boolean matrix, and the bounds on these numbers were given already by Schwarz [5, 6]. However, it is not known, how many $n \times n$ matrices there are with index m, and period l.

Our research was inspired by the recently proposed matrix test for randomness [1]. One of the tests is based on the distribution of indexes and periods of Boolean matrices. A tested (pseudo)random sequences is mapped to a set of Boolean matrices. For each matrix the index and period are computed. The statistics of the set formed by the sequence are compared by the (specific) χ-square test with the statistics of the whole set of Boolean matrices. If the test fails at some level α, the sequence is rejected.

The main problem of the test is that it is difficult to compute the required (complete) statistics of the whole set of matrices (this set contains 2^{n^2} matrices). Grošek et al. in [1] provide statistics for n (size of the matrix) up 5. We have later extended these results to cases $n = 6, 7$ [3, 4]. In this paper, we focus on the problem of computing these statistics for $n = 8$. In some cases the value $n = 8$ is optimal for the implementation of the test, because most of the computers today are byte oriented. Moreover, a single 8×8 Boolean matrix can be packed completely into one 64-bit word.

To compute the desired statistics by the classical method, we should compute 2^{64} indexes and periods of 8×8 Boolean matrices. Although it is a feasible computational effort, it is still too costly. However, we have been able to find a more effective enumeration algorithm, which reduces the complexity of the effort to approximately 2^{56}.[1] We have implemented this algorithm, and executed in a grid environment. The results of the computation are summarized in Appendix A.

The paper is organized as follows. In Section 2, we summarize more precisely the basic facts about Boolean matrices. Our enumeration algorithm is described in Section 3. We provide the details of the implementation in Section 4. Section 5 contains summary of the results, and the details of the computational effort involved. Finally, Section 6 contains our conclusions, remarks, and open questions.

2 PRELIMINARIES

Let $\mathcal{B} = (\{0,1\}, \vee, \wedge)$ be a standard Boolean algebra ($\vee$ denotes logical OR, and $\wedge$ denotes logical AND). Let $\mathcal{M}_n = \mathcal{B}^{(n\times n)}$ denote a set of all $n\times n$ Boolean matrices.

[1] For readers with interest in cryptography, we remark that this effort is similar to breaking DES by the brute force attack.

Let $\odot$ denote a Boolean matrix multiplication, i.e., if $A, B \in \mathcal{M}_n$, then $C = A \odot B \in \mathcal{M}_n$, and $c_{i,j} = \bigvee_{j=1}^{n} a_{i,k} \wedge b_{k,j}$. To simplify notation, we omit the symbol $\odot$ when writing products of different matrices from $\mathcal{M}_n$.

For the convenience of the reader, we summarize basic mathematical facts about the operation $\odot$ from [1, 5, 6]. Algebra $(\mathcal{M}_n, \odot)$ is a finite semigroup, i.e., the operation $\odot$ is closed on $\mathcal{M}_n$ and associative. Let I be an identity matrix (with ones on diagonal). Let $A^0 = I$, and let $A^i = A \odot A^{i-1}$ for $i > 0$ integer, i.e., A^i is a usual i-th power of matrix under $\odot$ multiplication. As the number of elements of $\mathcal{M}_n$ is finite, surely $A^j = A^i$ for some $j > i$.

Definition 1. Let $A \in \mathcal{M}_n$. Let $\mu(A), \lambda(A)$ be the smallest positive integers such that $A^{\mu}(A) = A^{\mu(A)+k}$ for any $k > 1$. We call $\mu(A)$ an index of A, and $\lambda(A)$ a period of A.

According to the Euler-Fermat Theorem for Finite Semigroups [5], there exist two numbers M, Λ, such that for each $A \in \mathcal{M}_n : x^{M+\Lambda} = x^M$, with

$$M = \max\{\mu(A); A \in \mathcal{M}_n\},$$

$$\Lambda = \operatorname{lcm}\{\lambda(A); A \in \mathcal{M}_n\}.$$

We call M, Λ universal exponents of $\mathcal{M}_n$ (i.e., the universal index and period, respectively). For the semigroup of Boolean matrices, we can use the following two theorems to compute the universal exponents.

Theorem 1. [6] For the semigroup $\mathcal{M}_n$ of $n \times n$ Boolean matrices, the universal index $M = (n-1)^2 + 1$.

Theorem 2. [6] Let $n = n_1 + n_2 + \ldots + n_k$ be a partition of n. Then $\Lambda = \operatorname{lcm}\{\operatorname{lcm}\{n_1, n_2, \ldots, n_k\}\}$, where the outer least common multiplier is taken across all possible partitions of the integer n.

However, in our research we are not interested directly in the value Λ. Instead, we only want to know the maximum period that can be achieved. This value is then

$$\lambda_{\max} = \max\{\operatorname{lcm}\{n_1, n_2, \ldots, n_k\}\},$$

where the maximum is again taken across all possible partitions of n. Relevant possible exponents for small n's are summarized in Table 1.

n	M	Λ	$\lambda_{\max}$
6	26	60	6
7	37	420	12
8	50	840	15

Table 1. Universal exponents for the semigroup of Boolean matrices (selected n's)

3 ENUMERATION OF MATRICES WITH A GIVEN INDEX AND PERIOD

Let $\mathcal{A} \subset \mathcal{M}_n$ be a set of $n \times n$ Boolean matrices. Let $\nu_{\mathcal{A}}(m,l)$ denote the number of Boolean matrices from the set $\mathcal{A}$ with given index m, and period l, i.e., $\nu_{\mathcal{A}}(m,l) = |\{A \in \mathcal{A}; \mu(A) = m, \lambda(A) = l\}|$. We are interested in the question of effective computation of $\nu_{\mathcal{M}_n}$ for a given (small) n. For simplicity, we will use ν_n in place of $\nu_{\mathcal{M}_n}$. The number of possible indexes and periods is limited (and small). Thus it is possible to use the following enumeration algorithm to compute ν_n:

1. For $m = 0, \dots, M$, $l = 0, \dots, \lambda_{\max}$: initialize counters $C[m,l] \leftarrow 0$;
2. For each matrix $A \in \mathcal{M}_n$: compute $\mu(A)$, $\lambda(A)$, and increment $C[\mu(A), \lambda(A)]$;
3. Output: $\nu_n(m,l) = C[m,l]$.

The complexity of the basic algorithm is 2^{n^2} computations. Grošek et al. [1] were able to use this algorithm for n up to 5. Using internal bit parallelism [4] we were able to compute the values up to $n = 7$, with the estimate for $n = 8$ on 23 000 CPU years [3]. In this section we show the advanced enumeration algorithm with a lower computational complexity (with the same memory requirements).

It is well known that two square matrices, say A, B, are equivalent if there exists a permutation matrix[2] P such that $PAP^T = B$. If A and B are adjacency matrices of graphs G_A, G_B, then these graphs are isomorphic iff A and B are equivalent. The outline of our algorithm is based on the following two lemmas drawn from graph theory.

Lemma 1. Let $A \in \mathcal{M}_n$ be an $n \times n$ Boolean matrix. Let P be an $n \times n$ permutation matrix. Then $\mu(PAP^T) = \mu(A)$, and $\lambda(PAP^T) = \lambda(A)$.

Lemma 1 could be used to simplify the enumeration algorithm. Suppose we compute $\mu(A)$, $\lambda(A)$. Then it is not necessary to compute μ, λ again for each of the matrices PAP^T, we can add to counter $C[\mu(A), \lambda(A)]$ the cardinality of $\{PAP^T\}$. However, this number can be different for different matrices, and can be lower than the number of possible permutation matrices, $n!$ (e.g., when $A = I$). The most difficult problem is to provide a sequence of matrices $\langle A_i \rangle$, such that $A_{i+1} \notin \bigcup_{j=1}^{i} \{PA_jP^T\}$. This problem is equivalent to enumerating all directed graphs (with loops) up to isomorphism.

Lemma 2. Let $\mathcal{A} \subset \mathcal{M}_n$ be a set of $n \times n$ Boolean matrices. Let P be an $n \times n$ permutation matrix, then $|P\mathcal{A}P^T| = |\mathcal{A}|$.

Corollary 1. Let $\mathcal{A} \subset \mathcal{M}_n$ be a set of $n \times n$ Boolean matrices. Let P be an $n \times n$ permutation matrix and let $\mathcal{B} = P\mathcal{A}P^T$, then for each (m,l): $|\{B \in \mathcal{B} : \mu(B) = m, \lambda(B) = l\}| = |\{A \in \mathcal{A} : \mu(A) = m, \lambda(A) = l\}|$, i.e., the index and period statistics of $\mathcal{A}, \mathcal{B}$ are the same.

[2] A permutation matrix has exactly one 1 in each row and column.

Let $\mathcal{P}$ denote a set of permutation matrices, and let $\mathcal{A} \subset \mathcal{M}_n$. Let $\mathcal{P} \otimes \mathcal{A}$ denote a set $\{B = PAP^T; P \in \mathcal{P}, A \in \mathcal{A}\}$. Let $\mathcal{P}_{\mathcal{A}}$ denote a special set of permutation matrices such that sets $P\mathcal{A}P^T$ are all pairwise disjoint. Using Corollary 1, we can see that in this special case $\nu_{\mathcal{P}_{\mathcal{A}} \otimes \mathcal{A}} = |\mathcal{P}_{\mathcal{A}}| \cdot \nu_{\mathcal{A}}$. That is, we can compute the index and period statistics of the set $\mathcal{P}_{\mathcal{A}} \otimes \mathcal{A}$ (which is at least as large as $\mathcal{A}$, but usually much larger) just by computing the index and period statistics of the set $\mathcal{A}$.

In our algorithm, we partition the set $\mathcal{M}_n$ into disjoint sets $\mathcal{P}_{\mathcal{A}_i} \otimes \mathcal{A}_i$. Each set $\mathcal{A}_i$ is formed in such a way that it is easy to enumerate its elements, and to compute $|\mathcal{P}_{\mathcal{A}_i}|$, respectively. Instead of computing ν_n by computing the index and period of each matrix in $\mathcal{M}_n$, we only compute indexes and periods of matrices in $\bigcup \mathcal{A}_i$, and then compute

$$\nu_n = \sum_i |\mathcal{P}_{\mathcal{A}_i}| \cdot \nu_{\mathcal{A}_i}.$$

The partition in our algorithm is based on the signatures of matrices.

Definition 2. Let $A \in \mathcal{M}_n$ be an $n \times n$ Boolean matrix that can be written in a block form as

$$\begin{pmatrix} A_1 & R \\ L & A_2 \end{pmatrix},$$

where A_1 is $(n-k) \times (n-k)$ matrix, A_2 is $k \times k$ matrix, and L^T, R are $(n-k) \times k$ matrices. We will call $(n-k) \times 2k$ matrix

$$S_k = \begin{pmatrix} L^T & R \end{pmatrix},$$

a k-signature of A.

Definition 3. We will call a k-signature S_k of A an ordered k-signature, if the rows of S_k are lexicographically ordered.

Example 1. Let

$$A = \left(\begin{array}{ccc|c} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ \hline 0 & 0 & 1 & 0 \end{array}\right).$$

Its 1-signature is the matrix:

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

which is an ordered signature. Its 2-signature is the matrix

$$\left(\begin{array}{cc|cc} 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{array}\right),$$

which is not ordered (in lexicographic order).

Lemma 3. Let $\mathcal{A}$ be a set of $n \times n$ Boolean matrices with the same k-signature S_k. Let O_k be a matrix obtained from S_k by ordering its rows lexicographically. Let P_1 be a $(n-k) \times (n-k)$ permutation matrix such that $O_k = P_1 S_k$, and let P be a $n \times n$ permutation matrix

$$P = \begin{pmatrix} P_1 & 0 \\ 0 & I \end{pmatrix}.$$

Then $\mathcal{B} = P\mathcal{A}P^T$ is a set of $n \times n$ Boolean matrices with the same k-signature O_k.

Proof. Let $A \in \mathcal{M}_n$ have signature $S_k = \begin{pmatrix} L^T & R \end{pmatrix}$. We can see that

$$PAP^T = \begin{pmatrix} P_1 & 0 \\ 0 & I \end{pmatrix} \begin{pmatrix} A_1 & R \\ L & A_2 \end{pmatrix} \begin{pmatrix} P_1^T & 0 \\ 0 & I \end{pmatrix} = \begin{pmatrix} P_1 A_1 P_1^T & P_1 R \\ L P_1^T & A_2 \end{pmatrix}.$$

Signature of PAP^T is then

$$\begin{pmatrix} P_1 L^T & P_1 R \end{pmatrix} = P_1 \begin{pmatrix} L^T & R \end{pmatrix} = P_1 S_k = O_k.$$

□

A signature splits a matrix in two parts in a special way. If we apply operation PAP^T, we are in fact swapping rows and columns in the same way. For example, if PA has exchanged rows 1 and 3 (compared to A), then AP^T has exchanged columns 1 and 3. If we change only rows (and columns) from the upper-left part of the matrix (first $n-k$ rows/columns), we do not "destroy" the k-signature of A, we only exchange its rows. For each matrix A we can always find a permutation matrix operating on first $n-k$ rows/columns only, such that the signature of PAP^T is ordered. The set of all matrices can be split into distinct sets according to a common signature up to the order of rows. These subsets can be further split into distinct classes corresponding to different permutations of the signature rows. Each of these classes can be obtained from the set of matrices with an ordered signature and a corresponding permutation matrix. This is exactly the division of the space $\mathcal{M}_n$ required for our algorithm.

Our algorithm works as follows:

1. Initialize counters $C[m, l] \leftarrow 0$;
2. For each ordered signature

$$S_k = \begin{pmatrix} L^T & R \end{pmatrix}:$$

   (a) For $m = 0, \dots, M$, $l = 0, \dots, \lambda_{\max}$: initialize counters $C_s[m, l] \leftarrow 0$;

   (b) For each matrix

$$A = \begin{pmatrix} A_1 & R \\ L & A_2 \end{pmatrix},$$

   where $A_1 \in \mathcal{M}_{n-k}, A_2 \in \mathcal{M}_k$:
   compute $\mu(A)$, $\lambda(A)$, and increment $C_s[\mu(A), \lambda(A)]$;

(c) Compute N, the number of distinct row permutations of S_k.
(d) For each m, l: $C[m,l] \to C[m,l] + N \cdot C_s[m,l]$.

3. Output $\nu_n(m,l) = C[m,l]$.

N.B. For given n, k an optimal choice is detailed in the following section.

3.1 Complexity of the Enumeration

The number of possible ordered k-signatures (for a given n) is a number of $(n-k)$ combinations of possible 2^{2k} elements with repetitions, that is

$$\binom{2^{2k} + (n-k) - 1}{n-k}.$$

For each ordered k-signature we must check all possible values of the bits of A_1, A_2, that is $(n-k)^2 + k^2$ bits. Thus we must compute the indexes and periods of

$$N = 2^{(n-k)^2+k^2} \binom{2^{2k} + (n-k) - 1}{n-k}$$

matrices. Numeric values for the cases $n = 7, 8$ are presented in Table 2. The case $k = 0$ corresponds to a basic enumeration of all matrices directly. The optimal choice of k depends on n, but for both $n = 7, 8$ the optimal value is $k = 2$.

$n = 7$	$N/2^{25}$	$\log_2 N$
$k = 0$	16 777 216	49.0
$k = 1$	344 064	43.4
$k = 2$	248 064	42.9
$k = 3$	766 480	44.5

$n = 8$	$N/2^{32}$	$\log_2 N$
$k = 0$	4 294 967 296	64.0
$k = 1$	31 457 280	56.9
$k = 2$	13 891 584	55.7
$k = 3$	41 696 512	57.3
$k = 4$	183 181 376	59.4

Table 2. The number of matrices that must be enumerated for $n = 7, 8$

4 IMPLEMENTATION DETAILS

The problem of computing the indexes and periods for the whole set of matrices can be distributed in a straightforward way. In our research we used two parallelization types, namely, internal and external.

In the case of external parallelization, we partition the set of matrices into (suitable) disjunct sets. We then compute statistics for each set separately in parallel computing nodes. Incidentally, during the statistics computation step, no communication is required between nodes. Finally, we need a short post-processing phase to add the results from each node. If the subsets are sufficiently large, the communication and post-processing cost is negligible.

The problem of computing μ, λ for the whole set can also be transformed into a SIMD-type (Single Instruction Multiple Data) of computation. We store the whole set (or a suitable subset) of matrices into a SIMD storage. We execute steps of the matrix multiplication algorithm in parallel. In each step, if we detect a collision between the actually computed and stored matrices, we mark the detected μ and λ in the k-th computing node (further collisions should be ignored). At the end of the computation we collect statistics from the nodes. The algorithm executes $\max\{\mu_k\}+\max\{\lambda_k\}$ matrix multiplications (and the corresponding number of comparisons), even if some of the nodes have finished earlier. However, if the size of the subsets is correctly chosen, the reduction in performance can be acceptable in some scenarios (e.g., when computing on GPUs or when using internal bit parallelization).

4.1 Internal Parallelization

For the core of the computation, we have used a slightly modified version of the program from [4]. For the sake of completeness, we summarize the description of the algorithm also in this section. Our implementation uses so called SWAR (SIMD Within A Register) principle, also known as a bit-slicing technique. To implement the operation $\odot$, we need bit operations AND and OR to work with individual bits of the matrix. However, a typical instruction AND, OR (in contemporary processors) works with the whole vector of 32-bits or 64-bits at once (depending on the architecture of the processor). Moreover, it is possible to utilize SSE2 (Streaming SIMD Extension 2) registers and operations, so we can even work with operations processing 128-bit vectors in one tact of the processor.

A bit-sliced implementation stores b Boolean $n \times n$ matrices in $n \times n$ b-bit words. Bits of the first matrix are stored in the LSB (Least Significant Bit) of each word. The second matrix is stored in the second bits, and so on. We say that we pack b $n \times n$ matrices into $n \times n$ b-bit words. The opposite operation is called unpacking.

The implementation of the operation $\odot$, which processes b packed matrices is then straightforward. We use a classical algorithm with the matrix multiplication replaced by the vector AND operation, and the addition replaced by the vector OR operation. A more complicated situation arises when we want to compare individual matrices. We cannot use vector comparison as each of n^2 vectors contains bits from b different matrices. We could unpack matrices after each multiplication, but this is quite costly, both computationally and memory-wise.

Next, let us introduce some notation to clearly illustrate the implementation. Let $\overline{x}[i,j] = (A_1^r[i,j], \ldots, A_b^r[i,j])$ and $\overline{y}[i,j] = (A_1^s[i,j], \ldots, A_b^s[i,j])$. We can compare individual bits of matrices $A_1^r, \ldots, A_b^r$ and $A_1^s, \ldots, A_b^s$ by the vector operation

$\overline{x}[i,j]$ XOR $\overline{y}[i,j]$. In the resulting vector only bits where $A_k^r[i,j] \neq A_k^s[i,j]$ are set. Finally, to compare whole matrices we must compute

$$\overline{z} = \bigvee_{\forall i,j} (\overline{x}[i,j] \text{ XOR } \overline{y}[i,j]),$$

where $\bigvee$ denotes the sum using the vector OR operation. Bits in vector $\overline{z}$ are set to 0 if and only if the corresponding matrices are equal. We thus need only $2n^2$ operations to compare b matrices instead of just one.

To finalize the bit-sliced implementation, we store a bit mask, with 0's corresponding to matrices with already computed index and period (it is initialized to all 1's). After each comparison, we compute the number of new hits (repeated matrices, indicating the cycle) with $c = w_H(\overline{m} \text{ AND NOT } \overline{z})$ (we add c to global statistics). Here w_H is a Hamming weight which can be computed in $\log_2 b$ steps (some processors even have dedicated instructions for this task). Finally, we update the mask with operation $\overline{m} := \overline{m}$ AND $\overline{z}$. The algorithm is finished (over the set of packed matrices) when $\overline{m} = (0, 0, \ldots, 0)$.

4.2 External Parallelization

The computation was split into three phases. The initial phase involves pre-computation: A set of ordered matrix signatures is generated (for the specified parameters n, k). Signatures are stored in the task file. Each signature denotes a unique identifier for a discrete computational task (TASKID). The signatures are stored in human readable format as hexadecimal numbers.

The main phase of the computation is run in parallel on the cluster. Each of the parallel tasks is assigned its TASKID (by the scheduler). Each node then computes the statistics of the set of matrices corresponding to this TASKID. In summary when $n = 8, k = 2$ (the chosen parameters of the computation) the single task computes all matrices of the form:

```
C0  C1  C2  C3  C4  C5  T2  T3
C6   x   x   x   x   x  T6  T7
 x   x   x   x   x   x  T10 T11
 x   x   x   x   x   x  T14 T15
 x   x   x   x   x   x  T18 T19
 x   x   x   x   x   x  T22 T23
 T1  T5  T9 T13 T17 T21  x   x
 T0  T4  T8 T12 T16 T20  x   x
```

Here, the individual symbols denote:

C0 — C6: 7 bits used by the internal parallelization (bit-slicing). The total of 128 matrices is packed for a single SIMD-type computation of indexes of periods.

T0 — T19: 24 bits of the (ordered) signature (TASKID, LSB is T0),

x: 33 bits, the elements of matrix enumerated in the cycle. Each task processes 2^{33} bitsliced operations (computations of indexes and periods).

Each task computes the statistics of the assigned set of matrices and encodes it into a prescribed output file. A common storage space was used for the results. MPI was used to manage the tasks and the repository of the results. Partial results were processed in the final phase of the computation and the final statistics were computed. More technical details are available in [2].

5 COMPUTATIONAL RESULTS

In the previous research [4], the computing time for $n = 7$ was reduced from the originally estimated 125 years to the 3.33 years using bit-sliced implementation (64-bit) or to 600 days using SSE2-enabled implementation (128-bit). An implementation of the brute-force search was executed on the parallel cluster at the GRID laboratory at FIIT STU in Bratislava using 50 computing nodes. The whole task for $n = 7$ took 125 hours (real time). The estimate for the brute-force search for the case $n = 8$ in the same configuration was 460 years[3] [3].

The new algorithm was run with parameters $n = 8, k = 7$ on NorGrid at the University of Bergen. The grid consists of 5 500 AMD Opteron 285 (E6) 2.6 GHz cores. The tasks were merged into blocks for 32-tuples of processors. The whole computation thus consisted of 1 696 blocks, with 32 tasks in each block. The average time to compute one block of tasks was approximately 12 hours. The total cost of the computation was 661 642.41 CPU-hours, 65 days in the real time (with average allocated load of 10 blocks or 320 processors). If it were possible to utilize the whole grid at 100 % just for our computation it would take less than 5 days. On one processor, the task would require approximately 75 years.

In comparison with the brute-force algorithm: one task for $n = 8$ takes 19 hours on the processors used in [3]. The whole effort with the new algorithm would take approx. 2.25 years, instead of 460 years predicted for the brute-force algorithm. That is, we were able to compute the statistics 200-times faster than with the brute force approach[4]. The final statistics are summarized in Appendix A.

6 CONCLUSIONS

Using our new algorithm we were able to compute the index and period statistics for the whole set of square Boolean matrices with the dimension up to $n = 8$. The whole effort took approximately 75 CPU years. To scale the computation to $n = 9$

[3] Equivalently, to compute the statistics in one month, 275 000 nodes was needed.

[4] The estimate from [3] does not take into account scaling of the matrix multiplication complexity, when changing dimension from $n = 7$ to $n = 8$. If we take this into account, the speedup is closer to the theoretical value of $2^{8.3}$ (see Table 2).

(again with the optimal choice of $k = 2$), it is necessary to compute statistics of the set of $2^{70.4}$ Boolean 9×9 matrices. This is 26 616-times more matrices than for $n = 8$. If we take into account the $O(n^3)$ complexity of the algorithmic step (computing index and period, respectively), we estimate the required processing power to $9^3/8^3 \cdot 26\,616 \cdot 75$ CPU years, i.e., 2.8 million CPU years. This effort is clearly too costly, so without new algorithms it seems infeasible to compute statistics for larger n's. It is an open question, whether the values of c_n can be computed analytically for any given n, λ, μ. We hope that the provided datasets can help further research in this area.

Acknowledgement

This material is based upon work supported under the grant NIL-I-004 from Iceland, Liechtenstein and Norway through the EEA Financial Mechanism and the Norwegian Financial Mechanism, and under the grant VEGA 1/0244/09.

We would like to thank the anonymous reviewer for his useful comments.

A STATISTICS OF THE SET OF MATRICES

This appendix summarizes computed statistics for the dimensions $n = 6, 7, 8$. Statistics for the smaller dimensions are taken from [1].

$\mu \backslash \lambda$	1	2
1	11	1
2	4	0

Table 3. Index and period statistics of the set of 2×2 Boolean matrices

$\mu \backslash \lambda$	1	2	3
1	123	33	2
2	252	12	0
3	66	0	0
4	18	0	0
5	6	0	0

Table 4. Index and period statistics of the set of 3×3 Boolean matrices

$\mu\backslash\lambda$	1	2	3	4
1	2 360	1 042	156	6
2	25 096	2 616	48	0
3	24 036	480	48	0
4	7 164	72	0	0
5	2 004	0	0	0
6	360	0	0	0
7	0	0	0	0
8	0	0	0	0
9	24	0	0	0
10	24	0	0	0

Table 5. Index and period statistics of the set of 4×4 Boolean matrices

$\mu\backslash\lambda$	1	2	3	4	5	6
1	73 023	43 125	9 230	1 080	24	60
2	6 471 160	604 780	26 780	360	0	680
3	16 980 510	305 580	18 720	240	0	840
4	7 190 310	72 180	2 760	240	0	480
5	1 384 530	12 060	240	0	0	240
6	297 960	960	0	0	0	240
7	28 320	0	0	0	0	0
8	8 160	0	0	0	0	0
9	9 120	0	0	0	0	0
10	9 000	0	0	0	0	0
11	720	0	0	0	0	0
12	240	0	0	0	0	0
13	120	0	0	0	0	0
14	120	0	0	0	0	0
15	0	0	0	0	0	0
16	120	0	0	0	0	0
17	120	0	0	0	0	0

Table 6. Index and period statistics of the set of 5×5 Boolean matrices

$\mu\backslash\lambda$	1	2	3	4	5	6
1	3 494 057	2 505 841	585 120	130 350	9 864	20 940
2	6 899 015 014	276 634 710	12 953 700	443 160	2 880	683 640
3	38 728 955 040	357 346 620	15 374 280	284 040	2 880	1 131 720
4	18 226 493 280	111 682 620	3 581 640	225 360	1 440	513 360
5	3 483 235 920	24 613 020	675 360	22 320	1 440	226 800
6	475 306 200	4 104 360	52 560	1 440	0	190 800
7	62 632 080	115 200	5 760	0	0	12 960
8	12 044 160	14 400	1 440	0	0	0
9	6 897 360	184 680	0	0	0	0
10	5 527 920	184 680	0	0	0	0
11	680 400	0	0	0	0	0
12	224 640	0	0	0	0	0
13	145 080	0	0	0	0	0
14	102 600	0	0	0	0	0
15	3 600	0	0	0	0	0
16	93 240	0	0	0	0	0
17	99 720	0	0	0	0	0
18	3 600	0	0	0	0	0
19	0	0	0	0	0	0
29	0	0	0	0	0	0
21	0	0	0	0	0	0
22	0	0	0	0	0	0
23	0	0	0	0	0	0
24	0	0	0	0	0	0
25	720	0	0	0	0	0
26	720	0	0	0	0	0

Table 7. Index and period statistics of the set of 6×6 Boolean matrices

$\mu \backslash \lambda$	1	2	3	4	5	6	7	8	9	10	11	12
1	251 098 172	203 906 556	46 495 932	14 914 620	2 502 696	4 910 850	720	0	0	1 512	0	1 260
2	40 285 779 030 224	333 304 635 356	12 717 360 460	461 239 380	8 299 368	979 579 300	0	0	0	122 976	0	269 640
3	347 967 810 628 116	1 085 242 534 530	32 157 645 240	604 597 140	6 753 600	2 239 503 000	0	0	0	322 560	0	988 680
4	144 645 479 650 728	461 787 506 670	9 976 676 220	389 401 740	3 351 600	959 610 540	0	0	0	267 120	0	950 880
5	24 643 793 152 776	90 654 179 490	2 280 074 160	63 069 300	2 721 600	352 808 400	0	0	0	156 240	0	589 680
6	3 042 915 246 840	18 005 408 400	346 817 520	5 690 160	236 880	263 378 640	0	0	0	80 640	0	317 520
7	248 546 287 920	1 536 917 760	16 919 280	352 800	10 080	24 323 040	0	0	0	40 320	0	161 280
8	37 412 828 040	396 436 320	3 538 080	50 400	0	725 760	0	0	0	20 160	0	80 640
9	11 820 160 800	540 003 240	13 841 520	0	0	30 240	0	0	0	10 080	0	40 320
10	7 919 650 200	532 884 240	13 750 800	0	0	10 080	0	0	0	10 080	0	20 160
11	1 005 001 200	31 046 400	50 400	0	0	0	0	0	0	0	0	10 080
12	348 839 400	10 316 880	30 240	0	0	0	0	0	0	0	0	10 080
13	224 332 080	5 158 440	0	0	0	0	0	0	0	0	0	0
14	144 117 960	5 158 440	0	0	0	0	0	0	0	0	0	0
15	7 066 080	0	0	0	0	0	0	0	0	0	0	0
16	121 102 800	5 158 440	0	0	0	0	0	0	0	0	0	0
17	132 833 400	5 158 440	0	0	0	0	0	0	0	0	0	0
18	7 292 880	0	0	0	0	0	0	0	0	0	0	0
19	186 480	0	0	0	0	0	0	0	0	0	0	0
20	10 080	0	0	0	0	0	0	0	0	0	0	0
21	5 040	0	0	0	0	0	0	0	0	0	0	0
22	5 040	0	0	0	0	0	0	0	0	0	0	0
23	0	0	0	0	0	0	0	0	0	0	0	0
24	20 160	0	0	0	0	0	0	0	0	0	0	0
25	1 310 400	0	0	0	0	0	0	0	0	0	0	0
26	1 350 720	0	0	0	0	0	0	0	0	0	0	0
27	30 240	0	0	0	0	0	0	0	0	0	0	0
28	0	0	0	0	0	0	0	0	0	0	0	0
29	0	0	0	0	0	0	0	0	0	0	0	0
30	10 080	0	0	0	0	0	0	0	0	0	0	0
31	5 040	0	0	0	0	0	0	0	0	0	0	0
32	5 040	0	0	0	0	0	0	0	0	0	0	0
33	0	0	0	0	0	0	0	0	0	0	0	0
34	0	0	0	0	0	0	0	0	0	0	0	0
35	0	0	0	0	0	0	0	0	0	0	0	0
36	5 040	0	0	0	0	0	0	0	0	0	0	0
37	5 040	0	0	0	0	0	0	0	0	0	0	0

Table 8. Index and period statistics of the set of 7×7 Boolean matrices

$\mu \backslash \lambda$	1	2	3	4	5	6	7	8	10	12	15
1	26 521 267 153	23 091 886 805	4 950 978 172	1 963 546 746	538 270 992	1 073 649 920	1 506 240	5 040	2 453 472	1 963 920	8 064
2	1 198 095 124 984 760 272	1 503 794 564 156 696	35 611 739 327 640	988 443 291 360	17 937 001 824	2 581 161 254 560	322 560	0	624 839 040	1 474 512 480	7 327 488
3	12 637 036 278 543 119 548	11 110 326 740 783 256	199 241 564 211 200	2 954 839 134 240	29 787 188 928	8 932 930 196 400	483 840	0	2 089 672 704	6 705 972 000	42 809 088
4	4 014 337 642 292 351 724	5 443 924 215 164 232	84 388 004 615 040	1 744 551 317 040	14 433 121 920	4 014 267 416 880	322 560	0	1 576 572 480	6 626 544 960	51 152 640
5	513 861 340 785 759 204	1 031 311 648 106 880	16 832 189 401 440	331 172 051 280	9 191 098 560	1 329 712 325 760	161 280	0	1 052 735 040	3 126 164 160	34 863 360
6	58 419 080 106 440 760	151 802 996 214 240	3 167 660 163 360	45 002 815 200	1 095 272 640	861 920 488 800	80 640	0	383 806 080	1 519 197 120	19 514 880
7	4 962 814 075 850 400	18 067 580 516 160	276 970 639 680	2 242 437 120	86 123 520	77 715 005 760	80 640	0	185 310 720	750 798 720	10 160 640
8	302 779 678 834 080	3 595 175 781 120	72 782 055 360	307 238 400	3 870 720	3 453 690 240	0	0	88 784 640	359 009 280	5 160 960
9	59 690 379 390 264	2 627 407 792 800	94 774 881 600	1 328 695 200	403 200	184 383 360	0	0	44 392 320	178 859 520	2 580 480
10	30 879 255 836 184	2 291 789 656 800	93 327 151 680	1 321 356 960	80 640	44 795 520	0	0	41 731 200	88 139 520	1 290 240
11	3 342 356 488 800	212 070 337 920	5 497 914 240	2 177 280	0	1 290 240	0	0	1 693 440	43 263 360	645 120
12	1 136 144 671 200	70 024 187 520	1 891 451 520	483 840	0	80 640	0	0	0	41 731 200	322 560
13	702 730 077 840	42 514 486 560	883 935 360	241 920	0	0	0	0	0	1 209 600	161 280
14	448 652 650 320	32 771 783 520	881 112 960	80 640	0	0	0	0	0	0	80 640
15	22 442 968 800	825 652 800	564 480	0	0	0	0	0	0	0	80 640
16	371 084 510 160	30 625 086 240	880 790 400	0	0	0	0	0	0	0	0
17	405 763 943 760	32 111 261 280	880 790 400	0	0	0	0	0	0	0	0
18	22 570 642 080	825 733 440	0	0	0	0	0	0	0	0	0
19	861 295 680	40 320	0	0	0	0	0	0	0	0	0
20	51 488 640	40 320	0	0	0	0	0	0	0	0	0
21	33 163 200	0	0	0	0	0	0	0	0	0	0
22	23 970 240	0	0	0	0	0	0	0	0	0	0
23	1 290 240	0	0	0	0	0	0	0	0	0	0
24	84 309 120	0	0	0	0	0	0	0	0	0	0
25	3 887 362 080	165 130 560	0	0	0	0	0	0	0	0	0
26	4 051 887 840	165 130 560	0	0	0	0	0	0	0	0	0
27	128 136 960	0	0	0	0	0	0	0	0	0	0
28	927 360	0	0	0	0	0	0	0	0	0	0
29	0	0	0	0	0	0	0	0	0	0	0
30	41 650 560	0	0	0	0	0	0	0	0	0	0
31	21 712 320	0	0	0	0	0	0	0	0	0	0
32	20 865 600	0	0	0	0	0	0	0	0	0	0
33	201 600	0	0	0	0	0	0	0	0	0	0
34	0	0	0	0	0	0	0	0	0	0	0
35	161 280	0	0	0	0	0	0	0	0	0	0
36	20 825 280	0	0	0	0	0	0	0	0	0	0
37	21 147 840	0	0	0	0	0	0	0	0	0	0
38	241 920	0	0	0	0	0	0	0	0	0	0
39	0	0	0	0	0	0	0	0	0	0	0
40	0	0	0	0	0	0	0	0	0	0	0
41	0	0	0	0	0	0	0	0	0	0	0
42	0	0	0	0	0	0	0	0	0	0	0
43	0	0	0	0	0	0	0	0	0	0	0
44	0	0	0	0	0	0	0	0	0	0	0
45	0	0	0	0	0	0	0	0	0	0	0
46	0	0	0	0	0	0	0	0	0	0	0
47	0	0	0	0	0	0	0	0	0	0	0
48	0	0	0	0	0	0	0	0	0	0	0
49	40 320	0	0	0	0	0	0	0	0	0	0
50	40 320	0	0	0	0	0	0	0	0	0	0

Table 9. Index and period statistics of the set of 8×8 Boolean matrices (all zero columns skipped)

REFERENCES

[1] Grošek, O.—Vojvoda, M.—Krchnavý, R.: A New Matrix Test for Randomness. Computing, Vol. 85, 2009, No. 1-2, pp. 21–36.

[2] Jókay, M.: Remarks on the Grid Computation of the Characteristics of Boolean Matrices. In GCCP 2010 Proceedings: 6th International Workshop on Grid Computing for Complex Problems. Bratislava, Slovakia, November 8–10, 2010, pp. 57–63.

[3] Jókay, M.—Zajac, P.: Advances in the Matrix Test Precomputation. In Proceedings. 1st Plenary Conference of the NIL-I-004. Development of Norwegian-Slovak Cooperation in Cryptology, Bergen, Norway, August 24–27, 2009, pp. 1–6.

[4] Jókay, M.—Zajac, P.: Parallelization Techniques for the Matrix Test Precomputation. In 5th International Workshop on Grid Computing for Complex Problems. GCCP 2009, Bratislava, Slovak Republic, October 26–28, 2009, pp. 103–109.

[5] Schwarz, Š.: On the Semigroup of Binary Relations on a Finite Set. Czech. Math. J., Vol. 95, 1970, No. 20, pp. 632–679.

[6] Schwarz, Š.: Sums of Powers of Binary Relations. Mat. Cas., Vol. 24, 1974, No. 2, pp. 161–171 (in Russian).

Pavol Zajac is an Associate Professor at UIM FEI STU. He got his Ph. D. in applied mathematics in 2008. He is currently working on problems of algebraic cryptanalysis and general cryptology research.

Matúš Jókay is a lecturer and researcher at UIM FEI STU. He got his Ph. D. in applied informatics in 2011. His research interests are in steganography and in large distributed computing with applications in cryptology.