МАТЕМАТИКА И МАТЕМАТИЧЕСКО ОБРАЗОВАНИЕ, 2010
MATHEMATICS AND EDUCATION IN MATHEMATICS, 2010
Proceedings of the Thirty Ninth Spring Conference of
the Union of Bulgarian Mathematicians
Albena, April 6–10, 2010

# A REPRESENTATION OF BINARY MATRICES*

**Hristina Kostadinova, Krasimir Yordzhev**

In this article we discuss the representation of binary matrix using a sequence of positive integers. We examine some advantages and disadvantages of this presentation as an alternative to the standard representation using a two-dimensional matrix. It is shown that the representation of binary matrices using ordered $n$-tuples of natural numbers makes the algorithms faster and saves a lot of memory. In this work we use object-oriented programming using the syntax and the semantic of C++ programming language.

**1. Introduction.** One of the basic principles of object-oriented programming is encapsulation, which means that the client knows objects' data, properties and algorithms, to which events the object reacts, but it is not necessary to have the information about their realization and the algorithms used in the member functions of the corresponding class. Here we ask the following question: if there are two different classes, which describe one and the same object in mathematics or in the real world, and they have the same properties and methods, the question is which one of these two classes we have to choose. The answer is trivial: the class, which objects use less memory and which algorithms work faster, i.e. which algorithms use less standard computer operations.

The present paper is a continuation of the paper [5]. Our aim is to show that the representation of the binary matrices using ordered $n$-tuple of positive integers and the bitwise operations make the realization of a better class (as it was mentioned above) compared with the standard representation of the binary matrices using two-dimensional $n \times n$ matrix of positive integers. About the definition and some examples how to use bitwise operations see [2, 3, 5, 7]. We recommend [4] about the object-oriented programming using C++ language in the area of the computer algebra.

We examine the set $\mathcal{B} = \{0, 1\}$. $\mathcal{B}$ together with the operations conjunction &&, disjunction $\|$ and negation ! form the all known *boolean algebra* $\mathcal{B}(\&\&, \|, !)$, which role is very important in the computers and programming. We use the mentioned symbols for the operations in $\mathcal{B}(\&\&, \|, !)$, have in mind the semantic and the syntax of these operations in the widespread C++ programming language.

*Binary* matrix (or *boolean*, or $(0, 1)$-*matrix*) is a matrix, which elements belong to the set $\mathcal{B} = \{0, 1\}$. We denote by $\mathcal{B}_n$ the set of all $n \times n$ square matrices.

*2000 **Mathematics Subject Classification:** 68N15, 68W40, 15B34.
**Key words:** Binary matrix, object-oriented programming, C++ programming language, bitwise operations, computer algebra.

Let $A = (a_{ij})$ and $B = (b_{ij})$ be matrices of $\mathcal{B}_n$. We consider the semantic and syntax of C++ language and the first index is 0, i.e $i, j \in \{0, 1, 2, \ldots, n-1\}$. We examine the following operations in $\mathcal{B}_n$, defined according to our aim as follows:

**component conjunction**

$$A \,\&\&\, B = C = (c_{ij}) \tag{1}$$

where, by definition $c_{ij} = a_{ij} \,\&\&\, b_{ij}$ for every $i, j \in \{0, 1, 2 \ldots, n-1\}$;

**component disjunction**

$$A \parallel B = C = (c_{ij}) \tag{2}$$

where, by definition $c_{ij} = a_{ij} \parallel b_{ij}$ for every $i, j \in \{0, 1, 2 \ldots, n-1\}$;

**component negation**

$$!A = C = (c_{ij}) \tag{3}$$

where, by definition $c_{ij} = !a_{ij}$ for every $i, j \in \{0, 1, 2 \ldots, n-1\}$;

**transpose**

$$t(A) = C = (c_{ij}) \tag{4}$$

where, by definition $c_{ij} = a_{ji}$ for every $i, j \in \{0, 1, 2 \ldots, n-1\}$;

**logical product**

$$A * B = C = (c_{ij}) \tag{5}$$

where, by definition

$$c_{ij} = \bigvee_{k=0}^{n-1} (a_{ik} \,\&\&\, b_{kj}) = (a_{i\,0} \,\&\&\, b_{0\,j}) \parallel (a_{i\,1} \,\&\&\, b_{1\,j}) \parallel \cdots \parallel (a_{i\,n-1} \,\&\&\, b_{n-1\,j})$$

for every $i, j \in \{0, 1, 2 \ldots, n-1\}$.

That is the way $\mathcal{B}_n$ and the above-described operations $\&\&$, $\parallel$, $!$, $t()$ and $*$ to form the algebra $\mathcal{B}_n(\&\&, \parallel, !, t(), *)$. Here and in the whole article the term *algebra* means *abstract algebra* considering the definition given in [1], and, namely, set equipped with various operations, assumed to satisfy some specified system of axiomatic laws. It is naturally to put the linear order, and exactly the lexicographic order in $\mathcal{B}_n(\&\&, \parallel, !, t(), *)$.

We examine the following set of standard operations with integer arguments in the C++ programming language:

$$\mathbf{Op} = \{+, -, *, /, \%, <<, >>, \&, |, \wedge, \sim, \&\&, \parallel, !, =, if, <, <=, >, >=, ==, !=\}. \tag{6}$$

These operations mean addition, subtraction, multiplication, division, integer division, bitwise left shift, bitwise right shift, bitwise "and", bitwise "or", bitwise "exclusive or", bitwise "negation", conjunction, disjunction, negation, assignment, if check, and comparing. We consider that the time, needed for each of these operations of the set **Op** are proportional, i.e. if $t_1$ and $t_2$ are the times needed to execute arbitraty operations of **Op**, then $t_1 = Ct_2$, where $C$ is a const. The algorithms in this paper are evaluated according to the number of the needed operations of the set **Op**.

The present paper is also an appropriate example how to use bitwise operations in the object-oriented programming courses. This matter does not take enough place in the studying literature(see for example [5]).

**2. Two classes, which describe the algebra $\mathcal{B}_n(\&\&, \|, !, t(), *)$.** To create the first class we use the standard realization of the binary matrices: using a two-dimensional $n \times n$ matrix of positive integers and standard algorithms to execute the operations (1) $\div$ (5). Let us denote this class by Bn_array.

A square binary $n \times n$ matrix, as it is described in [5], can be realized using ordered $n$-tuple of whole nonnegative numbers, which belong to the closed interval $[0,\ 2^n - 1]$. There is one to one correspondence between the representation of the integers in decimal and in binary number system. Let us denote by Bn_tuple the class which describes the algebra $\mathcal{B}_n(\&\&, \|, !, t(), *)$ using ordered $n$-tuple of positive integers.

These two classes have the same specifications (we use the terminology in [6, 8]), and we describe these specification using the name Bn_X, i.e. "X" means "array", or "tuple" depending on the case.

Let the two classes Bn_X have the following specification:

```
class Bn_X {
    int n;
    int *Matr;
  public:
/* constructor without parameter: */
    Bn_X();
/* constructor with parameter n pointing the row of the square matrix: */
    Bn_X (unsigned int);
/*  copy constructor: */
    Bn_X (const Bn_X &);
/* destructor: */
    ~Bn_X();
/* predefines the assignment operator: */
    Bn_X & operator = (const Bn_X &);
/* returns the size (row) of the matrix: */
    int get_n() { return n; };
/* sets value 1 to the element (i,j) of the matrix: */
    void set_1 (int,int);
/* sets value 0 to the element (i,j) of the matrix: */
    void set_0 (int,int);
/* fills row i of the matrix using integer number r
  (just for the class Bn_tuple): */
    void set_row(int,int);  // must not be included when "X" = "array"!
/* gets element (i,j) of the matrix: */
    int get_element (int,int);
/* gets the row i of the matrix
  (just for the class Bn_tuple): */
    int get_row (int);    // must not be included when "X" = "array"!
/* transposes matrix : */
    Bn_X t ();
/* predefines operators according to (1), (2), (3) and (5): */
    Bn_X operator && (Bn_X &);
    Bn_X operator || (Bn_X &);
```

```
    Bn_X operator ! ();
    Bn_X operator * (Bn_X &);
/* defines order (lexicographical) */
    int operator < (Bn_X &);
}
```

When we predefine the operators &&, || and $*$ and if the dimensions of the two operands are not equal, then we receive the zero matrix of order the same as the first operand. But this result is not correct. We can say something more: in this case the operation is not defined, i.e. the result of the operation function is not correct and we have to be very careful in such situations. The situation is the same about the entered linear order, i.e. although the lexicographic order can be put for the words with different length, we examine by definition only the matrices of the same order. The result we receive when we compare the matrices with different dimensions is not correct. We give suitable comments in this situations.

Since the objects of the two classes get dynamic operation memory, using pointers and the new operator, then to realize the applications, using such objects, it is necessary to predefine the operations of "the big three" [6] – copy constructor, destructor and the assignment operator.

The constructor without parameter and the destructor are the same for both classes Bn_array and Bn_tuple. Actually, when we work with objects of the class Bn_X, from the mathematical point of view it is necessary to point the dimension of the matrix and this dimension does not change. In this aspect using a constructor without parameter have not sense and we are not interested in it. But yet we add such a constructor to make our presentation complete and to make, that "the big three" to become "the big four"[6].

We use the universal integer type int to save the exactness in testing the program, this type can be changed to any other whole number type.

To evaluate the effectiveness and speed of the algorithms, which use objects of the algebra $\mathcal{B}_n(\&\&, \|, !, t(), *)$ it is necessary to evaluate the algorithms, which realize the operations &&, $\|$ !, $t()$, including the operation ¡, comparing two elements, and operation = "assignment". In that sense we describe in details just these methods, realizing the above mentioned operations. We suppose that the experienced programmer can easyly create the other methods in each of the two classes.

In the present work we predefine the operator "<" too. Using the same model we can predefine the other relational operators: "<=", ">", ">=", "==" and "!=". If the dimensions of the two matrices, we compare, are not equal, then the relation "<" is not defined and the result we get is the negative number $-1$.

**3. Realization of the class Bn_array.** When "X"=="array" we propose the following (*standard*) realization of the examined methods in the class Bn_array:

```
Bn_array Bn_array :: t () {
    Bn_array temp(n);
    for (int i=0; i<n; i++)
        for (int j=0; j<n; j++)
            *(temp.Matr + i*n+j) = *(Matr + j*n+i);
    return temp;
```

```
}

Bn_array Bn_array :: operator && (Bn_array &B) {
    Bn_array temp(n);
    int n2 = n*n;
    if (B.get_n() != n)
      cout<<"unallowable value of a parameter \n";
    else
        for (int p=0; p<n2; p++)
             *(temp.Matr + p) = *(this->Matr + p) && *(B.Matr + p);
    return temp;
}

Bn_array Bn_array :: operator || (Bn_array &B) {
    Bn_array temp(n);
    int n2 = n*n;
    if (B.get_n() != n)
      cout<<"unallowable value of a parameter \n";
    else
        for (int p=0; p<n2; p++)
             *(temp.Matr + p) = *(this->Matr + p) || *(B.Matr + p);
    return temp;
}

Bn_array Bn_array :: operator ! () {
    int n2 = n*n;
    for (int p=0; p<n2; p++)
        *(this->Matr + p) = *(this->Matr + p) ? 0 : 1;
    return *this;
}

Bn_array Bn_array :: operator * (Bn_array &B) {
    Bn_array temp(n);
    int c;
    if (B.get_n() != n)
      cout<<"unallowable value of a parameter \n";
      else
      for (int i=0; i<n; i++)
        for (int j=0; j<n; j++) {
          c=0;
          for (int k=0; k<n; k++) c = c || (*(this->Matr + i*n+k)
               && *(B.Matr +k*n+j));
          *(temp.Matr +i*n+j)=c;
        }
    return temp;
}
```

```
Bn_array& Bn_array :: operator = (const Bn_array &B) {
int n2=n*n;
    if (B.get_n() != n)
   cout<<"unallowable value of a parameter \n";
else
for (int p=0; p<n2; p++)
   *(this->Matr + p) = *(B.Matr + p);
return *this;
}
int Bn_array :: operator < (Bn_array &B) {
    int p = 0;
    int n2 = n*n;
    if (B.get_n() != n)
       cout<<"unallowable value of a parameter \n";
    else
         while ((*(Matr +p) == *(B.Matr +p)) && (p<n2-1) ) p++;
    if ( *(Matr +p) < *(B.Matr +p) ) return 0;
         else return 1;
}
```

Analogously we can realize the remaining relational operators $<=$, $==$, $>$, $>=$, $==$, $!=$.

It is easy to convince that the following proposition is true:

**Proposition 1.** *For each positive integer $n$, for the computer representation via the class Bn_array, using the C++ programming language, of the algebra $\mathcal{B}_n(\&\&, \|, !, t(), *)$, the following propositions are true:*

*(i) When we use standard realization (standard predefining) of the operation functions $\&\&$, $\|$, $!$, $<$, the operation transpose and the assignment operator, then each of them performs $O(n^2)$ operations of the set **Op**;*

*(ii) When we use standard realization (standard predefining) of the operation function $*$, then this operation performs $O(n^3)$ operations of the set **Op**;*

*(iii) For every object of the class Bn_array are necessary $O(n^2) *$ sizeof (int) bytes of the operating memory;*

*(iv) To make initialization of the object of the class Bn_array $O(n^2)$ operations are performed of the set **Op**.*

We can prove the propositions (i) and (ii) as we count the number of the inner cycles and the maximal number of the iterations in each of the methods. The propositions (iii) and (iv) are obvious.

**4. Representing the binary matrices using ordered $n$-tuples of nonnegative integers.** As it is shown in [5] there is one to one corresponding between the set of all $n \times n$ binary matrices and the set of all ordered $n$-tuples of whole numbers, which belong to the closed interval $[0,\ 2^n - 1]$, based on the binary representation of the positive integers. This idea takes place in the realization of the class Bn_tuple.

To create the class Bn_tuple we propose the following methods, which realize the examined operations in the algebra $\mathcal{B}_n(\&\&, \|, !, t(), *)$. To create these methods we use bitwise operations: bitwise conjunction &, bitwise disjunction |, bitwise exclusive "or" $\wedge$

and bitwise negation $\sim$, using these operations we raise the effectiveness and make the algorithms work faster.

```
Bn_tuple Bn_tuple :: t() {
Bn_tuple temp(n);
 int k;
for (int i=0; i<n; i++)
for (int j=0; j<n; j++) {
k=get_element(i,j);
if (k) temp.set_1(j,i);
else temp.set_0(j,i);
}
return temp;
}

Bn_tuple Bn_tuple :: operator && (Bn_tuple &B) {
    Bn_tuple temp(n);
    if (B.get_n() != n)
      cout<<"unallowable value of a parameter \n";
    else
        for (int p=0; p<n; p++)
             *(temp.Matr + p) = *(this->Matr + p) & *(B.Matr + p);
    return temp;
}

Bn_tuple Bn_tuple :: operator || (Bn_tuple &B) {
    Bn_tuple temp(n);
    if (B.get_n() != n)
      cout<<"unallowable value of a parameter \n";
    else
        for (int p=0; p<n; p++)
             *(temp.Matr + p) = *(this->Matr + p) | *(B.Matr + p);
    return temp;
}

Bn_tuple Bn_tuple :: operator ! () {
Bn_tuple temp(n);
for (int i=0; i<n; i++) {
 for (int j=0; j<n; j++) {
 if ( get_element(i,j) ) temp.set_0(i,j);
else temp.set_1(i,j);
 }
}
return temp;
}
```

```
Bn_tuple Bn_tuple :: operator * (Bn_tuple &B) {
    Bn_tuple temp(n), TB(n);
    TB = t(B);
    int c, r_i, r_j;
    if (B.get_n() != n)
      cout<<"unallowable value of a parameter \n";
      else
      for (int i=0; i<n; i++)
        for (int j=0; j<n; j++) {
          r_i = this->get_row(i);
          r_j = TB.get_row(j);
          c = r_i & r_j;
          if (c==0) temp.set_0(i,j);
               else temp.set_1(i,j);
        }
    return temp;
}
Bn_tuple& Bn_tuple :: operator = (const Bn_tuple &B) {
if (B.get_n() != n)
   cout<<"unallowable value\n";
else
for (int p=0; p<n; p++)
   *(this->Matr + p) = *(B.Matr + p);
return *this;
}
int Bn_tuple :: operator < (Bn_tuple &B) {
int p = 0;
if (B.get_n() != n)
   cout<<"unallowable value\n";
else
while ((get_row(p)==B.get_row(p))&&(p<n-1 )) p++;
if (get_row(p)<B.get_row(p) ) return 1;
  else return 0;
}
```

Analogously to Proposition 1 we are convinced that the following proposition is true:

**Proposition 2.** *For each positive integer $n$, for the computer representation via the class Bn_tuple, using the C++ programming language, of the algebra $\mathcal{B}_n(\&\&, \|, !, t(), *)$, the following propositions are true:*

*(i) When we use the above-described realization of the operation functions $\&\&$, $\|$, $<$ and the predefining of the assignment operator, then each one of them performs $O(n)$ operations of the set **Op**;*

*(ii) The transpose and negation operation performs $O(n^2)$ operations of the set **Op**.*

*(iii) When we use the above-described realization of the operation functions $*$, then this operation performs $O(n^2)$ operations of the set **Op**;*

*(iv) For every object of the class Bn_tuple $O(n) *$* sizeof (int) *bytes of the operating*

*memory are necessary;*

(*v*) *Initialization of the object of the class Bn_tuple can be performed using* $O(n)$ *operations of the set* ***Op****.*

We see that to create the class Bn_array is easy and it is not a difficult task even for the beginning programmer, when we describe the algorithms we conform to the definitions of the corresponding operations. On the other side, comparing Proposition 1 with Proposition 2 we are convinced that following proposition is true:

**Theorem 1.** *The algorithms using objects of the class Bn_tuple work faster than the algorithms using objects of the class Bn_array and they save a lot of operating memory.*

## REFERENCES


[1] J. Daintith, R. D. Nelson. The Penguin Dictionary of Mathemathics. Penguin books, 1989.

[2] S. R. Davis. C++ for Dummies. IDG Books Worldwide, 2000.

[3] B. W. Kernigan, D. M Ritchie. The C Programming Language. AT&T Bell Laboratories, 1998.

[4] Tan Kiat Shi, W.-H. Steeb, Y. Hardy. Symbolic C++: An Introduction to Computer Algebra using Object-Oriented Programming. Springer, 2001.

[5] K. Yordzhev. An Example for the Use of Bitwise Operations in programming. *Mathematics and Education in Mathematics*, **38** (2009), 196–202.

[6] П. Азълов. Обектно ориентирано програмиране Структури от данни и STL. София, Сиела, 2008.

[7] Е. Л. Романов. Практикум по программированию на C++. БХВ-Петербург, 2004.

[8] М. Тодорова. Програмиране на C++. Част I, част II, София, Сиела, 2002.



Hristina Kostadinova, Krasimir Yordzhev
South-West University "N. Rilsky"
2700 Blagoevgrad, Bulgaria
e-mail: hkostadinova@gmail.com
e-mail: yordzhev@swu.bg, iordjev@yahoo.com


## ВЪРХУ ЕДНО ПРЕДСТАВЯНЕ НА БИНАРНИТЕ МАТРИЦИ


**Христина Костадинова, Красимир Йорджев**



В статията се обсъжда представянето на произволна бинарна матрица с помощта на последователност от цели неотрицателни числа. Разгледани са някои предимства и недостатъци на това представяне като алтернатива на стандартното, общоприето представяне чрез двумерен масив. Показано е, че представянето на бинарните матрици с помощта на наредени $n$-торки от естествени числа води до по-бързи алгоритми и до съществена икономия на оперативна памет. Използуван е апарата на обектно-ориентираното програмиране със синтаксиса и семантиката на езика C++.