sciendo

*Int. J. Appl. Math. Comput. Sci., 2018, Vol. 28, No. 4, 803–815*
DOI: 10.2478/amcs-2018-0061

amcs

# OPTIMIZATION ON THE COMPLEMENTATION PROCEDURE TOWARDS EFFICIENT IMPLEMENTATION OF THE INDEX GENERATION FUNCTION

GRZEGORZ BOROWIK [a]

[a] Faculty of Internal Security
Police Academy in Szczytno, Marszałka Józefa Piłsudskiego 111, 12-100 Szczytno, Poland
e-mail: g.borowik@wspol.edu.pl

In the era of big data, solutions are desired that would be capable of efficient data reduction. This paper presents a summary of research on an algorithm for complementation of a Boolean function which is fundamental for logic synthesis and data mining. Successively, the existing problems and their proposed solutions are examined, including the analysis of current implementations of the algorithm. Then, methods to speed up the computation process and efficient parallel implementation of the algorithm are shown; they include optimization of data representation, recursive decomposition, merging, and removal of redundant data. Besides the discussion of computational complexity, the paper compares the processing times of the proposed solution with those for the well-known analysis and data mining systems. Although the presented idea is focused on searching for all possible solutions, it can be restricted to finding just those of the smallest size. Both approaches are of great application potential, including proving mathematical theorems, logic synthesis, especially index generation functions, or data processing and mining such as feature selection, data discretization, rule generation, etc. The problem considered is NP-hard, and it is easy to point to examples that are not solvable within the expected amount of time. However, the solution allows the barrier of computations to be moved one step further. For example, the unique algorithm can calculate, as the only one at the moment, all minimal sets of features for few standard benchmarks. Unlike many existing methods, the algorithm additionally works with undetermined values. The result of this research is an easily extendable experimental software that is the fastest among the tested solutions and the data mining systems.

**Keywords:** data reduction, feature selection, indiscernibility matrix, logic synthesis, index generation function.

## 1. Introduction

The amount of raw data coming into computer systems has become more and more troublesome, and sometimes impossible to process. Therefore, computers—just like people in daily life—must use a selection or aggregation process for the incoming data to determine their primary purposes; this is especially true for computations. Another process, called *machine learning*, allows a machine (e.g., a computer) to make decisions in new situations after processing the input training data.

During all the stages of data processing, methods of optimization of combinational circuits—from the field of digital logic—might be useful. These methods could be used during automatic analysis of data and the process of extracting information that is hidden to the human eye, called *data mining* (Stefanowski *et al*., 2017). Thus, it is possible to develop rules that could be further used for a certain analysis, such as patient diagnosis, illegal money transfer, or unauthorized data access.

Many data mining approaches employ feature selection techniques to speed up learning and improve model quality. These techniques are especially important for datasets with tens or hundreds of thousands of features. That is why extensive research from various points of view has been made in recent years for fast and efficient attribute reduction algorithms (e.g., Szemenyei and Vajda, 2017; Liu *et al*., 2015; Martinović *et al*., 2014; Sun *et al*., 2014; Min *et al*., 2014; Borowik and Łuba, 2014; Korzen and Jaroszewicz, 2005; Zhong and Skowron, 2001; Liu and Setiono, 1997; Łuba and Rybnik, 1992).

Most existing methods represent objects with a single vectorial descriptor where it is assumed that each vector has the same number of dimensions. During the reduction, the decision system is simplified so as to get a minimal set

of features/attributes that retain the classifying abilities of the system. The reduction in decision systems consists in determining reducts (a formal definition can be found in the work of Komorowski *et al*. (1999)) and optionally removing redundant objects. The selection of an attribute subset plays an important role in *knowledge discovery*: it forms a basis for more efficient classification, prediction, and building approximation models. It is especially important for combinatorial circuits and FPGA structures, and serves as a basis for the decomposition process, i.e., splitting a function into smaller independent blocks (Borowik and Łuba, 2014).

As was shown by Skowron and Rauszer (1992), the problems of finding all minimal reducts as well as those of the smallest cardinality are NP-hard. Therefore, it is easy to point to examples that are not solvable within the expected amount of time. A barrier of computational complexity allows only moving the 'center of gravity' for computations. Thus, existing algorithms are both systematic, which allows finding all minimal solutions, and heuristic, having higher operational speeds.

The problem of finding reducts can be restricted to finding just those of the smallest size. Such a solution set could be used in all the problems of data mining that do not require the full data set. Among others, the research led by Steinbach and Posthoff (2012) is based on the $\mathrm{DIF}(S_i, \mathrm{NDM}(P))$ approach. In this way, the computation time can be reduced by more than $8 \cdot 10^8$ times, and the result of the algorithm is the set of all minimal shortest solutions (reducts).

One solution that allows increasing the speed is to move the intense transformation to graphical processing units (GPUs). Compared with central processing units (CPUs), they have thousands of cores—for example, Nvidia GeForce® GTX™ TITAN Z has 5760 CUDA cores—among which each can perform independent computations. In the case of algorithms presented by Steinbach and Posthoff (2013), the use of a GPU to compute the minimal reducts of the smallest cardinality allowed reducing the computation time 6.5 times, compared with using a CPU. Employing a similar approach can be similarly effective in the case of the unate complement procedure.

Because of the large computational complexity for large data sets, statistical algorithms are useful. Examples of such approaches are the algorithm of feature extraction described by Korzen and Jaroszewicz (2005) as well as the algorithm of data discretization described by Borowik *et al.* (2015a).

Some applications require the recognition of individual vectors of an object. In such cases, it is essential that the nodes within a single object remain distinguishable after dimension reduction. Szemenyei and Vajda (2017) proposed a new discriminant analysis methods that are able to satisfy two criteria at the same time: a separation between classes and a separation between nodes of an object instance.

Martinović *et al*. (2014) proposed an approach to dimensionality reduction in the form of a method of pattern classification where a feature subset selection uses a wrapper method. Here a designated number of solutions found throughout differential evolution execution is archived and then a method of post-evaluation using k-fold cross-validation is used to generate the final solution.

This paper presents a new systematic method of reducing data using a method of logic synthesis, called the unate complement (UC) procedure. It was proposed by Brayton *et al*. (1984). Introductory research (Borowik and Łuba, 2014) has shown that the unate complement procedure is a much faster method of computing reductions than those implemented in Rough Set Data Explorer 2 (ROSE 2) and Rough Set Exploration System 2.2 (RSES 2.2) (Predki *et al*., 1998; Predki and Wilk, 1999; Bazan *et al*., 2002; Nguyen, 2006). In addition, computations performed using the unate complement procedure can be optimized against memory usage, which makes it possible to analyze larger data sets compared with the expert systems mentioned above.

The primary purpose of this paper is to show how this procedure for data analysis can be used in medical and telecommunication areas (e.g., Łuba *et al.*, 2014; Abraham *et al.*, 2007; Su *et al.*, 2009; Sasao, 2011; 2015). The list of possible areas in which this proposed solution and algorithm can be applied is not limited to these two, however, since many applications can be found in business (stock trading), statistics, and banking (Jankowski *et al*., 2015), among others.

## 2. Fundamental information

**2.1. FPGAs with a built-in memory.** The basic problem of reducing arguments is to find the smallest sets of input parameters of Boolean functions for which the data remains consistent. For example, for a function presented in Table 1, variables $x_1$, $x_3$, and $x_5$ are redundant; after removing them, each pair of rows of various decisions differs by at least one position.

The result from argument reduction presented in Table 2 is just one of three possible solutions.

Argument reduction is fundamental because decreasing the number of input parameters can decrease the time required for further optimization. In the case where the function is implemented in a field programmable gate array (FPGA) chip, after reducing arguments, the in-chip implementation of the function has fewer inputs. Also, parallel decomposition is based on argument reduction (Borowik and Łuba, 2014).

This is particularly important for FPGAs with a built-in memory for which the number of input variables

Table 1. Example of a truth table.

| $U$ | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ | $y$ |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| 2 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| 3 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 |
| 4 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 5 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| 6 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
| 7 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| 8 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| 9 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 |

Table 2. One of three possible solutions of argument reduction for the truth table from Table 1.

| $U'$ | $x_2$ | $x_4$ | $x_6$ | $x_7$ | $y$ |
|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 1 | 0 |
| 2 | 0 | 1 | 1 | 0 | 0 |
| 3 | 1 | 1 | 1 | 0 | 0 |
| 4 | 1 | 0 | 1 | 1 | 0 |
| 5 | 1 | 0 | 0 | 1 | 1 |
| 6 | 0 | 0 | 1 | 0 | 1 |
| 7 | 0 | 0 | 0 | 0 | 1 |
| 8 | 0 | 0 | 1 | 0 | 1 |
| 9 | 1 | 0 | 0 | 1 | 1 |

Table 3. Example of a decision table.

| $U$ | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $y$ |
|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 1 | 1 |
| 2 | 1 | 0 | 0 | 0 | 1 |
| 3 | 0 | 0 | 0 | 0 | 0 |
| 4 | 1 | 1 | 0 | 1 | 0 |
| 5 | 1 | 1 | 0 | 2 | 2 |
| 6 | 2 | 2 | 0 | 2 | 2 |
| 7 | 2 | 2 | 2 | 2 | 2 |

Table 4. Table of minimal rules.

| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $y$ |
|---|---|---|---|---|
| 1 | 0 | – | – | 1 |
| 0 | – | – | – | 0 |
| – | 1 | – | 1 | 0 |
| – | – | – | 2 | 2 |

affects the complexity of the implementation. An example is the synthesis of an index generation function (Sasao, 2011; 2015).

**2.2. Data mining.** At the same time, argument reduction—also known as attribute reduction or attribute selection—has broad applications in data mining. Sometimes the collected data is redundant, and using a subset of the parameters is enough to make an unambiguous decision. Surveys are a good example to that, especially the ones used by doctors while diagnosing patients.

Further, the approach presented in this paper is used for inducing decision rules, that is, generalization of objects in the decision table (Table 3) (Borowik *et al.*, 2015b). The goal is to obtain rules that will enable classifying new data. For example, after generating a table of minimal rules (Table 4) for a row of ones $(1111)$, it is possible, with some probability, to predict the value that should appear in column $y$. In this particular case, it would be the decision equal to '0'. It is easy to imagine that such a procedure could be useful as an aid for medical diagnosis. If the decision is not firm, the computer system could make it conduct additional examinations in order to arrive at a proper diagnosis.

Another direct use of the algorithm is the discretization of continuous data. This issue is described in detail by Borowik (2013) and Komorowski *et al.* (1999).

**2.3. Reduction methods.** During the basic implementation of the reduction method, the term *indiscernibility table* (indiscernibility matrix) is defined (a formal definition is given by Komorowski *et al.* (1999)). It is a matrix that is created by pairwise comparison of all rows of a truth table having different values. Each row of the indiscernibility table contains a '1' for those arguments for which compared rows differ for a given value and '$*$' (indeterminate value) in those places where the values are the same or for which at least one value is indeterminate. In the case of polyvalent discrete data, where arguments and decisions can have more than two values, the procedure of generating comparisons is exactly the same, i.e., rows with different decisions are compared and attribute diversity is marked with the value '1', otherwise the symbol '$*$' is used. The issue of creating an indiscernibility table is shown in Fig. 1.

A term that is directly related to the indiscernibility table is a *minimal column coverage*, which is a set of columns in the indiscernibility table for which the rows contain at least one '1' and no proper subset of this set meets these criteria. For example, for the matrix on the right in Fig. 1, $x_1$ and $x_4$ constitute a minimal column coverage; meanwhile, $x_1$, $x_2$, $x_4$ is a column coverage but is not minimal. Minimal column coverages computed for the indiscernibility table are called *reducts*. They are minimal sets of arguments/attributes required to represent a given Boolean function or decision function.

One of the methods in determining all the minimal column coverages is to represent the indiscernibility table in the form of a discernibility function. Such a function is a logical product of sums of arguments. Then, it is transformed into the form of the sum of products of arguments (Petrick, 1956; Skowron and Rauszer, 1992).

| $U$ | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $y$ |
|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 | 0 |
| 2 | 1 | 1 | 1 | – | 0 |
| 3 | 0 | 0 | 1 | 1 | 1 |
| 4 | 2 | 1 | 0 | 1 | 1 |

| | $x_1$ | $x_2$ | $x_3$ | $x_4$ |
|---|---|---|---|---|
| 1 & 3 | $*$ | $*$ | $*$ | 1 |
| 1 & 4 | 1 | 1 | 1 | 1 |
| 2 & 3 | 1 | 1 | $*$ | $*$ |
| 2 & 4 | 1 | $*$ | 1 | $*$ |

Fig. 1. Indiscernibility table generated comprises comparisons between each pair of objects from different decision classes.

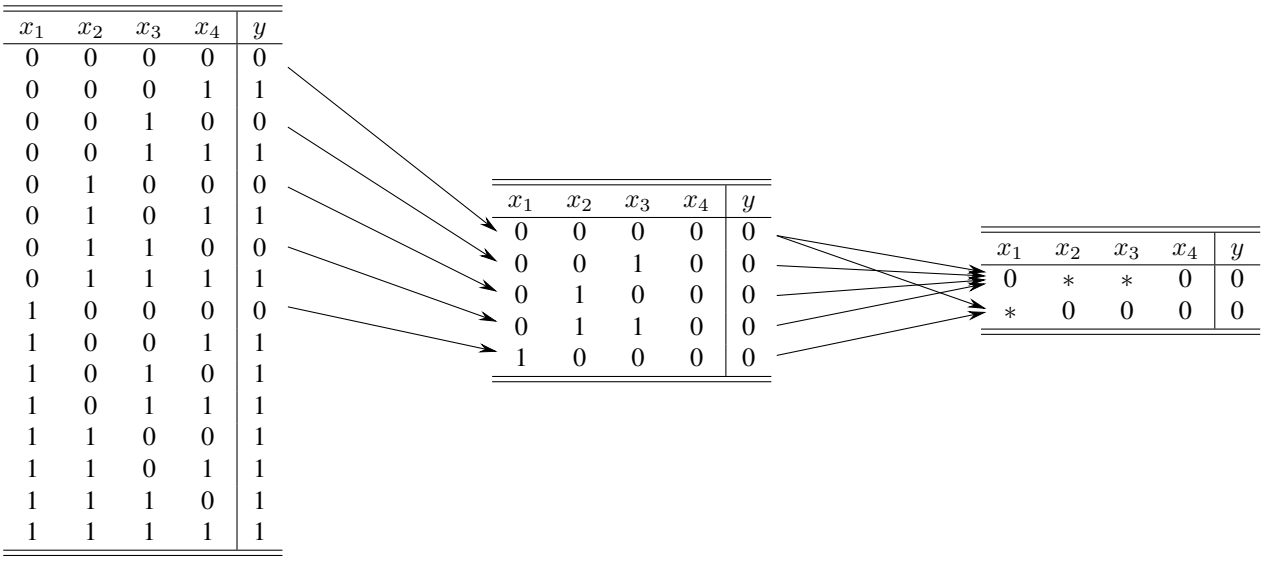


Fig. 2. Illustration of function complementation.

Each factor of such an expression represents one minimal column coverage. For example, the indiscernibility table from Fig. 1 can be written down as a product of sums having the following form:

$$x_4(x_1 + x_2 + x_3 + x_4)(x_1 + x_2)(x_1 + x_3),$$

which is transformed into the form of the sum of products, according to Boolean algebra,

$$x_1x_4 + x_2x_3x_4.$$

In this way, two reducts are created, $x_1$, $x_4$ and $x_2$, $x_3$, $x_4$. Thus, in order to represent the function from Fig. 1, not all arguments presented in the table are required; instead, only arguments $x_1$, $x_4$ or $x_2$, $x_3$, $x_4$ can be used.

The problem of designating all minimal column coverages can be solved by computing the complement of the discernibility function, which takes columns of the indiscernibility table as arguments. It is assumed that the values of the function for all the rows of indiscernibility table are set and equal to '1'. The complement of such a function is the set of rows (cubes) that do not have any elements in common with the other rows of the table.

The construction of the function is done in the following way. All cubes from the indiscernibility table from Fig. 1 are written down as binary vectors and each assumes a value of '1'. For binary vectors not belonging to this set, the value of '0' is assumed (the far left table in Fig. 2). The others, shown in the middle table in Fig. 2, can be minimized (generalized) according to the concept of least upper bound ($lub$) used by Brzozowski and Łuba (1997) as follows:

$$lub\{0000, 0010, 0100, 0110\} = 0 * *0$$

and

$$lub\{0000, 1000\} = *000.$$

The resulting cubes, shown in the far right table in Fig. 2, are identical with the earlier minimal column coverages.

Another way to quickly determine all the minimal column coverages is by using the unate complement procedure (Borowik and Łuba, 2014). This method determines the complements of a monotonic function, i.e., a function for which there is no column having zeros and ones at the same time in the table representation. This procedure allows finding all the largest complementary cubes (i.e., the most general complementary cubes); when applied to the indiscernibility table, it determines all the minimal column coverages.

The unate complement procedure works as follows:

1. recursive decomposition of the function, according to the Shannon scheme;
2. computing complements using De Morgan laws;
3. merging the resulting partial complements.

During the last stage, all redundant cubes are discarded, i.e., cubes that are logically included in others, according to cube calculus (Brzozowski and Łuba, 1997).

For the monotonic function, the formula for Shannon decomposition is simplified to the forms (Brayton *et al.*, 1984)

$$f = x_j f_{x_j} + f_{\overline{x}_j} \tag{1}$$

for a monotonically increasing function and

$$f = f_{x_j} + \overline{x}_j f_{\overline{x}_j} \tag{2}$$

for a monotonically decreasing function. Similarly, for the complement we have

$$\overline{f} = \overline{f}_{x_j} + \overline{x}_j \overline{f}_{\overline{x}_j} \tag{3}$$

for a monotonically increasing function and

$$\overline{f} = x_j \overline{f}_{x_j} + \overline{f}_{\overline{x}_j} \tag{4}$$

for a monotonically decreasing function.

All the methods that implement the algorithm to find the minimal column coverage require earlier creation of the indiscernibility table. The number of rows in such a table depends, quadratically, on the number of objects in the analyzed data set, and can be expressed by using the formula

$$\sum_{(p<r)\in k} n_p n_r,$$

where $k$ is the decision set, $p$ and $r$ run through all the pairs of the set $k$, and $n_p$ and $n_r$ are the numbers of rows for decisions $p$ and $r$, respectively.

**2.4. Rule induction and data discretization.** Data mining problems other than attribute reduction can be converted into the problem of finding a function complement, and so the unate complement procedure can be used. In such situations, the only difference in the application of the UC procedure is the way the input indiscernibility table is generated.

For example, in the process of the induction of decision rules (Borowik *et al.*, 2015b), the indiscernibility tables are generated using methods similar to those for attribute reduction. The difference is that each decision class must have separate indiscernibility matrices generated. These matrices are created as a result of comparing the object of a certain decision class with objects of other decision classes. The number of matrices shows the computational complexity of this task:

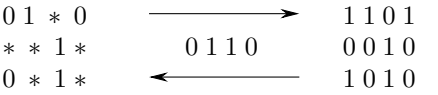


Fig. 3. Representation of a unate function.

usually, it is not solvable in polynomial time. In this case, the restrictions should be interpreted as a tradeoff between solution completeness and reasonable processing time (Borowik and Kowalski, 2015).

During discretization, some initial cut set is required (Borowik, 2013; Komorowski *et al.*, 1999). Next, each object in the indiscernibility table is created by comparing two objects of different decisions so that each cut is checked whether it lies between the values of the analyzed objects. In such cases, the row of the indiscernibility table contains '1' for this cut, or '0' otherwise.

## 3. Algorithm

The basic algorithm for processing the indiscernibility table, using the unate complement procedure, can be described in the following steps:

1. *Matrix representation.*
   The input matrix contains only 0s and 1s (no indeterminate values). Since the unate complement procedure transforms only unate functions, the following transformation can be applied: columns having 0s will contain 1s in the places where 0s initially were, and the indeterminate values will be replaced by 0s. As mask remembering columns have 0s (Fig. 3), such a transformation is exact and reversible. Thus, it is possible to recover the output matrix by using the mask stored in the memory.

2. *Verification.*
   Verification is used if there is a possibility of direct determination of the complement by using De Morgan laws. According to these, if a matrix

   (a) does not contain any row, then a tautology (the matrix containing all possible rows) is its complement; tautologies in cube calculus are represented with a cube containing only '$*$' elements; in memory representation, such a cube will have a form of a row containing only 0s;

   (b) contains a row with only 0s (the representation of the row with only '$*$'), its complement is an empty matrix;

   (c) contains just one row, its complement is a matrix containing that number of rows, as the number of 1s exists in a given row, and each of

those rows has exactly one '1' at the position relevant to 1s of the original row (Fig. 4).

If a given matrix has one of the forms (a)–(c), it must be verified and the complement computed; otherwise, the algorithm proceeds to the next step.

3. *Choice of a variable against which Shannon decomposition will be performed.*
The choice of such a variable can be done in several ways. The most important methods of choosing a variable are discussed further in this paper.

4. *Shannon decomposition using a simplified formula.*
Two new matrices are created (left and right), according to the selected divider variable. The left matrix is created by replacing all 1s with 0s in the column referred by the variable. The right matrix contains only rows that had 0s in the referred column (Fig. 5).

5. *Recursive execution of the algorithm for both parts of the matrix.*
The decomposition must be repeated until all leaves of the matrix tree have the form (a), (b), or (c) from Step 2; then, the complement is computed.

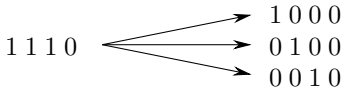


Fig. 4. Complement using De Morgan laws.

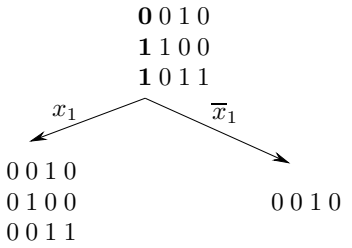


Fig. 5. Matrix decomposition using the Shannon formula.

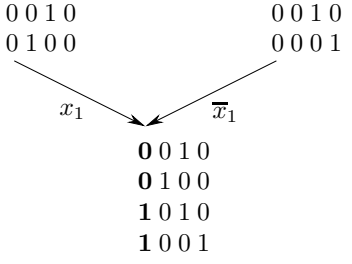


Fig. 6. Matrices merging using the Shannon formula.

6. *Merging the complements created in Step 5.*
Similarly to decomposition, merging is performed using the Shannon formula. The right-hand side complement is multiplied by the divider variable, whereas the left-side complement remains unchanged. The right-hand side matrix is then combined with the left-hand side matrix into a new set (matrix) (Fig. 6). It should be natural that this task is included in the recursive part of the algorithm.

7. *Yield of the complement.*
The matrix created at the top of the recursive algorithm tree is the complement of the original function.

## 4. Analysis of the original implementation and optimization suggestions

The advanced unate complement procedure was implemented in the 1980s in the Espresso software using the C programming language. Despite the use of many breakthrough algorithms, enabling fast computation, this implementation has some drawbacks.

**4.1. Memory operations and partitioning.** Espresso uses a single-dimension array of 32-bit unsigned integer variables to store the indiscernibility table in the memory. This table is virtually divided into rows, and the first field of *unsigned int* is used to store the control flags data; further $k$ fields, especially $k = 1$, contain the actual data. Despite using such a memory-efficient structure, Espresso faces insufficient memory problems. This is because the implementation of the complement procedure, in which a new matrix is created for each node of the algorithm tree, is not well thought out. For both the left- and right-hand side matrices, the memory for the size of the input matrix (the one being decomposed) is allocated regardless of the number of rows that actually will be placed in the matrix. Owing to this approach, de-allocating the matrix memory was very fast; however, the total memory used for the algorithm tree equals the size of the input matrix multiplied by the depth of the algorithm tree.

Probably because of the memory problems, an algorithm for removing redundant rows can be found in the original implementation of unate complement in Espresso. Rows are removed from the input indiscernibility matrix before executing the actual unate complement procedure. Redundant rows are those that are enclosed in other rows, according to the cube calculus, especially when they are enclosed in a single other row. It must be remembered that 0s, in fact, represent indeterminate values (Fig. 7).

The recursive unate complement procedure allows investigating which of the branches (left or right) will be processed first. The authors of the Espresso

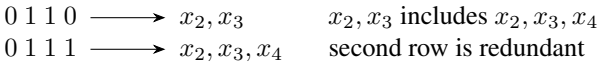


Fig. 7. Illustration of cube redundancy.

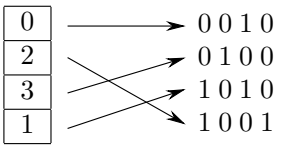


Fig. 8. Pointers to the row map.

implementation decided first to traverse the tree to the left. According to the analysis, after going to the second branch, regardless of which direction was chosen earlier, there is no need to store the previous matrix in the memory; instead of allocating a new memory, the input matrix could be modified. In the case of such an operation, the idea of choosing the right branch as the first for processing seems a much better idea. Then, zeroing the column indicated by the divider variable is the only modification required. When traversing the tree, as in the Espresso implementation, copying must be done at the time of branching left; when branching right, it would be necessary at least to mark rows that would have been ignored during further steps.

For the improvement, minimizing the usage of the memory, only matrix rows containing 0s in a column referenced by the divider variable are copied at the branching right step. To do this, before branching right, the rows to be copied must be counted and allocated in a smaller table in the memory accordingly. The advantage of such an approach is that, if the order of traversing a tree is changed and branching to the right is done first, then copying the matrix during stepping to the left is not necessary. The only required action is zeroing the contents of the column referred by the divider variable. Those two solutions minimize the amount of required memory to the constant amount dependent on the frequency of occurrence of 1s in the indiscernibility table—the more 1s in the indiscernibility table, the bigger memory utilization.

In the case presented above, when branching right, the size of the allocated memory can differ; thus, organizing the memory in blocks of a constant size as in Espresso does not reflect the algorithm behavior and would be inefficient.

**4.2. Memory operations and merging.** In the final implementation, the focus was on memory consumption of the algorithm. Despite considering the speed of the computation as the most important criterion, the biggest advantage of the implementation is extremely low memory consumption. This allows processing the examples given earlier that, when started in other programs, resulted in errors related to the insufficient free memory. In the proposed algorithm, realizing the unate complement procedure, the size of the memory used is approximately the size of the area required to store the indiscernibility table. In this way, Shannon decomposition does not perform matrix copying and has no need to increase the memory during the process of traversing the algorithm tree. To achieve this, when making the step to the right, instead of selecting and copying particular rows, the process of sorting rows is referred by a column indicated by the divider variable. Moreover, during the later stages of the algorithm, only a subset of computed rows is used. When the right side of the tree undergoes processing, it is enough to return to the full matrix size, and restoring the earlier order of the rows is not necessary. When branching left, the column of the divider variable is zeroed; after this branch of the tree has been examined, the appropriate rows must be restored.

Restoring rows turned out to be fairly difficult in the implementation. As opposed to working on particular rows, the solution is to treat the 1s in zeroed columns the same way as if they were 0s. Such a property can be achieved by using a special row, known as the *mask*. Before each reading of the row, the operation of logic AND is performed on a given row and the mask, and the result is passed on for further processing. At the beginning of the algorithm operation, the mask is a row containing just 1s, so executing the AND operation does not affect the processed row. Next, during each step to the left in the tree, on the position of the divider variable the value of '0' is set, and when returning back the value of '1' is set back. This way zeroing the entire column reduces to changing one bit at one position.

For the sake of realization of the algorithm in the memory of a fixed size, it is required to mark these rows that are used in further stages of the procedure. The *Polish Flag* sorting algorithm is chosen for the proposed solution, along with remembering the number of rows with 0s in the column referred by the divider variable. Such a solution could be improved using a map of pointers to rows. Eventually during the sorting process, rows remain in their places and only pointers in the map are modified (Fig. 8).

Thanks to the proposed solution, no objects in the list are exchanged, just values in the related cells; despite the necessity of referring to the rows through the additional table, this decreases the tree traversing time. The difference in the computation time, in this case, is of minor importance, while the stability of the data in the matrix is of primary significance. The use of pointers along with the mask assures complete constancy of the data during the tree traversing, so possible multi-threaded implementation can share the main memory. This is because the position in the tree in which the algorithm is

currently working depends only on the state of the mask, the map, the saved size of the matrix, and the saved divider variables required to generate the complement.

**4.3. Redundancy.** As a result of the unate complement procedure, part of the solution is redundant (reducts are not minimal), and before finalizing the solution they must be filtered out. A redundant solution is one that is enclosed in another reduct that exists in the solution set, according to the cube calculus (Fig. 7). To verify which solutions (rows of a complement) are redundant, they must be compared one with another. Knowing that only the longer reduct (longer in terms of the number of 1s) can be redundant in a certain pair, the following procedure can be used: before starting pairwise comparisons, reducts are sorted by the number of 1s, and then each row is compared only with rows shorter than that one. Two identical reducts will never be produced by the algorithm; therefore, comparing rows of the same length is not necessary. It is important to mention that the shorter the row is, in terms of the number of 1s, the more often it can cross out the other rows. Because of this, when comparing rows, it is worth starting from the shortest ones. A time analysis of this method showed that as the input data size increased, more and more time had to be spent to remove the redundant reducts (Fig. 9). For example, the time required to remove redundant reducts for an input matrix having 30 attributes takes 99% of the total processing time. The process of discarding redundant solutions has $O(n^2)$ complexity; therefore, it causes the biggest delays in the algorithm operation. The number of redundant reducts changes with respect to the number of rows must be checked, so the question is whether indiscernibility table partitioning would provide any expected advantages. Borowik and Łuba (2014) described the results of research regarding these issues.

**4.4. Variable selection.** It is natural to endeavor the fastest reach of one of these forms in which it is possible to determine the complement by using De Morgan laws. To achieve this, rows with the minimal number of 1s could be selected and pursued to get a row with only 0s. Another approach is to choose columns containing the maximum number of 1s and pursuing to get a single row in the matrix. Espresso uses both the approaches simultaneously. Among all the rows, those with the minimum number of 1s are selected, and then 1s are counted in columns of the selected rows in order to choose the best divider variable.

An interesting mechanism implemented in Espresso is choosing multiple divider variables in one step. In an exceptional case, when there is a row with just one '1', after selecting its position for the divider variable, the empty set is the complement of the left side, i.e.,

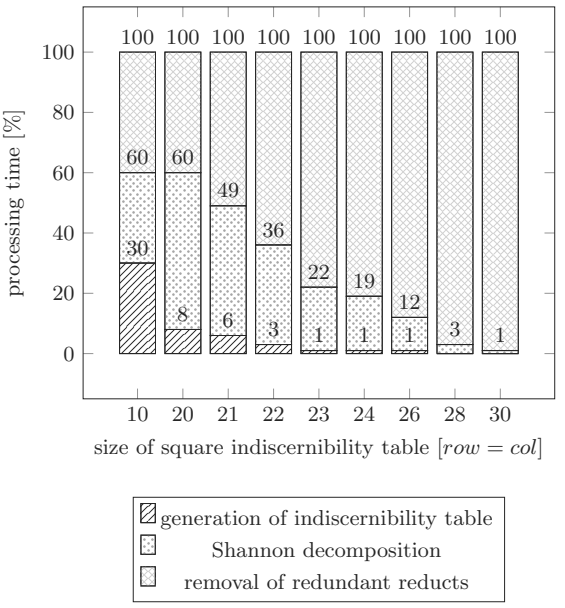


Fig. 9. UC processing time for all stages (in %).

which has the empty set as a complement according to the De Morgan laws. In this situation, the second divider variable can be chosen. Then the right-hand side matrix will be filled only with rows that have 0s at the positions of divider variables. The left-hand side matrix will receive rows having 0s for the first divider variable, remembering that the column for the second divider variable must be zeroed.

Mindful that the right-size memory is filled with rows having '0' at the column referred by the divider variable, further memory savings can be achieved by a well-thought-out selection of divider variables. To achieve this, the algorithm adopted from Espresso was implemented; that is, the selection of the divider variable was based on the row with a minimum number of 1s, and then a column was selected that has '1' in this row and, at the same time, contains more 1s than other columns.

The analysis of the unate complement procedure showed that the method used for choosing divider variables has a tremendous effect on the number of redundant reducts. For example, when using a similar method for choosing the divider variable as the one implemented in Espresso, for the *dermatology* database (Bache and Lichman, 2013), the number of reducts is 143093, and the number of reducts, including the redundant ones, just after traversing the tree is 210688. Thus, for the most effective choice, every third reduct must be rejected. When choosing the divider variable randomly without analyzing data in the matrix, this number increases to 2522447—only every 17th is required for the solution. According to computational complexity, a tenfold growth in the number of reducts

leads to around a hundredfold increase in the time spent on rejecting redundant reducts; so, in this case, it increases from 1 minute to 90 minutes. In addition, the time of traversing the algorithm tree grows from below 1 second to 17 seconds. Even ignoring the significant increase in computation time, for a large database, such as *lung-cancer* (Bache and Lichman, 2013), the number of redundant reducts in the case of randomly choosing a divider variable is so huge that the program terminates the operation due to insufficient memory. An ideal solution would be to choose the divider variable in such a way that no redundant reducts are produced. However, *this* research showed that it is easy to construct examples for which redundant solutions will always be generated.

**4.5. Concurrency.** When concurrent programming is used to traverse the tree faster, computing reducts in the leaves of an algorithm tree are essential. Thanks to a vector of present divider variables, each node has the complete information necessary to compute the complement for a given part of the input matrix. In such cases, multiple threads can separately compute complements for different (independent) nodes in the tree. It is important that, during each step of the algorithm, the problem of finding the complement can be divided into two separate threads. Therefore, if any of the threads finished computations of their part of the tree, another thread can assign it computations related to another, newly created node in another part of the tree in order to balance the processing load. The concurrent algorithm can be described using the following steps:

1. One of the threads starts computations, and the rest of the threads are declared as waiting.
2. Before repeating the algorithm for two new matrix parts, the thread checks if there is any waiting thread:
   (a) if a waiting thread exists, the checking thread delegates the right-hand side part of the matrix to the waiting thread and itself proceeds with computations for the left-hand side part of the matrix;
   (b) if a waiting thread does not exist, the checking thread continues computations for both sides of the matrix.

Repeating this operation in each node guarantees that threads will not remain too long in the waiting state. On the other hand, if small matrices dividing them might be unprofitable, it is better if matrices above a certain size are passed to concurrent processing.

When a thread finishes computing the complement for its tree branch, it changes its state to waiting and remains idle until it is assigned a new task. Also, when a thread switches to the waiting state, it must ensure that there exists at least one more thread that is still processing at that time in order to avoid the situation in which all threads are in the waiting state and the complement is already computed.

**4.6. Subset of solutions.** The use of the unate complement procedure, compared with a standard method of transforming a Boolean expression, considerably shortens the computing time and allows processing more complex tasks. This does not mean, however, that every data set can be processed using the proposed complementing method in a reasonable amount of time. Sometimes, computing all the reducts must/might be traded off for getting the solution as their subset in a given amount of time.

To get just a subset of attributes, traversing the entire tree is not necessarily advisable. In such cases, a method of determining the minimal reduct is necessary; the best type of method is if the reduct is minimal, of the smallest length, and found in a minimal amount of time. At the same time, it is worth noting that the number of reducts obtained for large indiscernibility tables, especially because of the number of arguments, is so huge that there can be a problem with storing solutions in the computer's operational memory during the execution of the algorithm while traversing the tree. An ideal solution for this problem is to store the solutions in the hard drive during the algorithm execution, or to stop traversing the tree down after finding the first reduct that satisfies the assumptions regarding the number of attributes.

Espresso does not provide such a possibility because partial solutions were multiplied by related divider variables at the stage of merging results while recursively leaving the tree. To make it possible to output the results when the tree leaf is reached, the implementation of an additional vector is required. Such a vector stores additional divider variables, but, only those that occurred during branching to the right while traversing a tree to reach the particular leaf. Multiplying such a vector with rows of the complement that were created in a certain leaf results in the creation of complete reducts. Such reducts can be multiplied after reaching the leaf of the tree; it is also possible to write down a vector of divider variables and a vector before the complement in the form of a pair. This saves memory space because, instead of storing multiple rows created during applying the De Morgan laws, only one pair of rows is stored. Even so, such a solution is not effective if two identical rows appear in the matrix at the final stage. Despite the fact that one of those rows can be discarded, and the complement can be determined for the remaining row (or written down as a pair with divider variables), in the further stages of the algorithm, the matrix built on the doubled row is decomposed. Decomposition is performed until the empty

matrix is received. Therefore, the complements generated in leaves have exactly one row identical to a row of present divider variables, so storing a pair takes twice less the memory. Thus, to fully utilize this natural method of compression, the additional algorithm step of catching and removing doubled rows from the matrix must be implemented.

## 5. Experimental results

Prior to efficiency tests, the correctness of the proposed algorithm operation was verified using benchmark solutions. They contained all the correct minimal reducts generated by an independent software application. In addition, the proper number of reducts was verified for the smaller result sets, and the results (obtained reducts) were checked as to whether they were identical. In all the cases, the results of the proposed algorithm were correct.

The operation times were measured for four programs, including a software implementation of the proposed algorithm. The testing machine was a PC based on an Intel Core i5-3210M processor with a 2.5 GHz clock, 6 GB RAM, and the MS Windows 7 operating system. The result of the operation was a set of minimal reducts constituting the complement of the indiscernibility table created from the input function (i.e., the Boolean function or the decision table).

Table 5 presents the comparison of operation time for the optimized unate complement procedure with the method of reducing the attributes used in the known expert systems: ROSE2 (Predki *et al.*, 1998; Predki and Wilk, 1999), RSES 2.2 (Bazan *et al.*, 2002), jMAF (Błaszczyński *et al.*, 2013), and to the unate complement procedure (Borowik and Łuba, 2014) developed earlier. For example, for the database *agaricus-lepiota-mushroom* (Bache and Lichman, 2013), the ROSE2 system generated just a subset of reducts after 164 seconds of operation, and RSES 2.2 resulted in a subset after 266 seconds or the entire set of reducts after 1740 seconds. The jMAF system resulted in a complete reduct set after 198 seconds, the unate complement procedure developed by Borowik and Łuba (2014) after 287 seconds, and the algorithm proposed in *this* paper provided a complete set of reducts in just 112 seconds.

**5.1. Efficient implementation of an index generation function: Practical application.** Attribute reduction is the process of finding the smallest set of attributes (a reduct) which is sufficient to produce the same decision from the reduced database as from the original one. When the data represent a Boolean truth table, the reduction decreases the number of input variables of the system in the way that the data remain consistent, and as a result, its implementation.

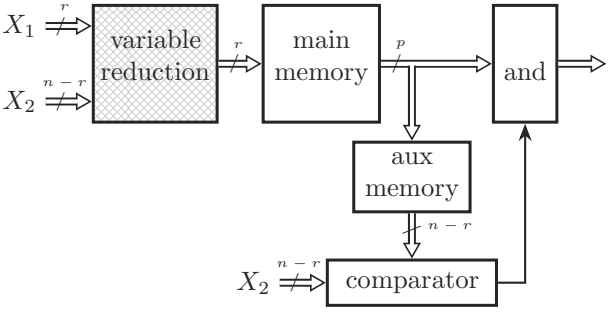


Fig. 10. Index generation function.

For FPGA implementations, such a reduction makes it possible to realize functions as combinational circuits of fewer inputs. An example is the synthesis of an index generation function (index generator), where the function is a strongly undefined Boolean one:

$$f : D^n \to \{1, 2, \ldots, k\}, \tag{5}$$

with $D^n \subseteq \{0,1\}^n$, and $|D^n| = k$.

A characteristic feature of this function is a large number of input variables with a relatively small cardinality of $D^n$ (vectors belonging to the $D^n$ set are called registered vectors). A consequence of this property is an efficient reduction of input variables, enabling the index generator to be implemented in the structure proposed by Sasao (2011). This structure contains the main memory, an auxiliary memory, a comparator, and an AND gate (Fig. 10).

The variable reduction block selects from among all variables a subset of $X_1$ representing a reduct of the index generation function ($X = X_1 + X_2$). The cardinality of the reduct equals $r$; that is also the number of inputs to the main memory. At the output of the main memory there is an input vector index. It is represented by a binary vector with a number of $p \geq \lceil \log_2(k) \rceil$ bits. This index is a valid index of the input vector, considering that it is a registered vector, and only then does this index appear on the outputs of the generator. If the input vector is not a registered vector, the generator yields '0'. The mechanism checking whether the output of the main memory represents the correct index is implemented in the auxiliary memory and the comparator. The auxiliary memory stores the vectors represented by the variables belonging to the set $X_2$. These vectors are fed to the comparator input and compared with the actual $X_2$ vector fed to the comparator's second input. Of course, the vector taken from the auxiliary memory is different from the actual $X_2$ vector when it is not registered. Then, the comparator output signal '0' will block the output of the AND gate of the vector generated in the main memory.

Research on efficient implementation of the index generator was done using the Boolean function analyzed

Table 5. Experimental results.

| database | time of calculation $[s]$ ROSE2[a] | RSES 2.2[b] | jMAF[c] | UC procedure[d] | **optimized UC procedure** | number of reducts |
|---|---|---|---|---|---|---|
| agaricus-lepiota-mushroom [e] | (164)* | (266)* / 1740 | 198 | 287 | **112** | 507 |
| audiology [e] | (12449)** | (660)** | – | 14.5 | **3** | 37367 |
| breast-cancer-wisconsin [e] | (<1)* | 2 | <1 | 0.8 | **0.2** | 27 |
| dermatology [e] | (36639)** | (1800)** | (145)*** | 212 | **72** | 143093 |
| house [e] | <1 | <1 | <1 | 0.2 | **0.1** | 4 |
| kaz [d] | (1906)** | 2520 | <1 | 0.2 | **0.2** | 5574 |
| kr-vs-kp [e] | 16 | 62 | 9 | 117 | **5.4** | 4 |
| lung-cancer [e] | (7634)** | (3120)** | >360000 | 403020 | **286869** | 3604887 |
| trains [e] | (8)* | (6)* / (20280)** | – | 0.1 | **0.1** | 689 |
| urology [a] | (8560)** | (12240)** | 15 | 42.7 | **1.2** | 23437 |

* method does not generate all the reducts
** time after which the memory error occurred
*** computations were performed on modified data sets, disregarding rows that contained indeterminate values
– computations could not be performed due to the presence of indeterminate values
[a] Predki *et al.*, 1998; Predki and Wilk, 1999
[b] Bazan *et al.*, 2002
[c] Błaszczynski *et al.*, 2013
[d] Borowik and Łuba, 2014
[e] Bache and Lichman, 2013

Table 6. Example of an index generator function (Sasao, 2015).

| | |
|---|---|
| 01100001000100001010011000010001000010010 | 1 |
| 01011111011010100011010111110110101000011 | 2 |
| 11110101011101110000111101010111011100001 | 3 |
| 00011110000100010111000111100001000101111 | 4 |
| 00111100000001000101001111000000010001010 | 5 |
| 01110010010001001001011100100100010010001 | 6 |
| 00100011100011110010001000111000111100010 | 7 |
| 11111111110100011110001000111000111100010 | 8 |
| 11101110001100010110111011100011000100110 | 9 |
| 10100001101001000011101000011010010000011 | 10 |

previously by Sasao (2015) and delivered *here* in Table 6. For this function, Sasao obtained a 5-argument reduct, $r = 5$ (Table 7).

It is very easy to notice that the efficiency of the constructed index generator is determined by the size of the reduct. For these reasons, in the summary of his invited talk at the *EPFL* workshop, Sasao (2015) pointed out that the study on an efficient attribute reduction algorithm is the most important research task in the field of pattern recognition. So it is worth stressing that, using the algorithm proposed in *this* paper, one can calculate a whole series of 4-argument reductions, $r = 4$. One of these solutions is given in Table 8.

## 6. Summary

The article is targeted at both Boolean functions and multi-valued functions. This is possible due to the appropriate transformation that produces from a function the so-called indiscernibility matrix, which is defined in Boolean algebra.

The computational results confirmed that, with the use of the unate complement procedure, it is possible to perform computations related to attribute reduction within a shorter time than in other expert systems. Further, it is presumed that it is now possible to perform computations for databases that have been too big until now because this implementation saves the computer operational memory.

Efficient implementation of complementing the Boolean function could especially be applied to the expert system or the index generation function. Computer programs based on this implementation could solve the problems mentioned above faster, and should be able to solve more complex problems—in terms of the number of attributes and rows.

## References

Abraham, A., Jain, R., Thomas, J. and Han, S.Y. (2007). D-SCIDS: Distributed soft computing intrusion detection system, *Journal of Network and Computer Applications* **30**(1): 81–98, DOI: 10.1016/j.jnca.2005.06.001.

Table 7. 5-Argument reduct for the index generator function from Table 6 published by Sasao (2015).

| |
|---|
| 01100 |
| 01011 |
| 11110 |
| 00011 |
| 00111 |
| 01110 |
| 00100 |
| 11111 |
| 11101 |
| 10100 |

Table 8. One of the 4-argument reductions for the index generator function from Table 6.

| |
|---|
| 0000 |
| 1011 |
| 1111 |
| 1110 |
| 1010 |
| 0010 |
| 0100 |
| 1100 |
| 1101 |
| 0001 |


Bache, K. and Lichman, M. (2013). *UCI Machine Learning Repository*, University of California, Irvine, CA, `http://archive.ics.uci.edu/ml`

Bazan, J.G., Szczuka, M.S. and Wróblewski, J. (2002). A new version of Rough Set Exploration System, *in* J.J. Alpigini *et al.* (Eds.), *Rough Sets and Current Trends in Computing*, Lecture Notes in Computer Science, Vol. 2475, Springer, Berlin/Heidelberg, pp. 397–404, DOI: 10.1007/3-540-45813-1_52.

Błaszczyński, J., Greco, S., Matarazzo, B., Słowiński, R. and Szeląg, M. (2013). jMAF-Dominance-based rough set data analysis framework, *in* A. Skowron and Z. Suraj (Eds.), *Rough Sets and Intelligent Systems—Professor Zdzisław Pawlak in Memoriam*, Springer, Berlin/Heidelberg, pp. 185–209, DOI: 10.1007/978-3-642-30344-9_5.

Borowik, G. (2013). Boolean function complementation based algorithm for data discretization, *in* R. Moreno-Díaz *et al.* (Eds.), *Computer Aided Systems Theory*, Lecture Notes in Computer Science, Vol. 8112, Springer, Berlin/Heidelberg, pp. 218–225, DOI: 10.1007/978-3-642-53862-9_28.

Borowik, G. and Kowalski, K. (2015). Rule induction based on frequencies of attribute values, *Proceedings of SPIE: Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments* **9662**: 96623R, DOI: 10.1117/12.2205899.

Borowik, G., Kowalski, K. and Jankowski, C. (2015a). Novel approach to data discretization, *Proceedings of SPIE: Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments* **9662**: 96623U, DOI: 10.1117/12.2205916.

Borowik, G., Kraśniewski, A. and Łuba, T. (2015b). Rule induction based on logic synthesis methods, *in* H. Selvaraj *et al.* (Eds.), *Progress in Systems Engineering*, Advances in Intelligent Systems and Computing, Vol. 330, Springer International Publishing, Cham, pp. 813–816, DOI: 10.1007/978-3-319-08422-0_118.

Borowik, G. and Łuba, T. (2014). Fast algorithm of attribute reduction based on the complementation of Boolean function, *in* R. Klempous *et al.* (Eds.), *Advanced Methods and Applications in Computational Intelligence*, Topics in Intelligent Engineering and Informatics, Springer International Publishing, Cham, pp. 25–41, DOI: 10.1007/978-3-319-01436-4_2.

Brayton, R.K., Hachtel, G.D., McMullen, C.T. and Sangiovanni-Vincentelli, A. (1984). *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, Dordrecht, DOI: 10.1007/978-1-4613-2821-6.

Brzozowski, J.A. and Łuba, T. (1997). Decomposition of Boolean functions specified by cubes, *Research report CS-97-01*, University of Waterloo, Waterloo.

Jankowski, C., Reda, D., Mańkowski, M. and Borowik, G. (2015). Discretization of data using Boolean transformations and information theory based evaluation criteria, *Bulletin of the Polish Academy of Sciences: Technical Sciences* **63**(4): 923–932, DOI: 10.1515/bpasts-2015-0105.

Komorowski, J., Pawlak, Z., Polkowski, L. and Skowron, A. (1999). Rough sets: A tutorial, `https://eecs.ceas.uc.edu/~mazlack/dbm.w2011/KomorowskiRoughSets.tutor.pdf`.

Korzen, M. and Jaroszewicz, S. (2005). Finding reducts without building the discernibility matrix, *Proceedings of the 5th International Conference on Intelligent Systems Design and Applications, ISDA'05, Wrocław, Poland*, pp. 450–455, DOI: 10.1109/ISDA.2005.45.

Liu, G., Li, L., Yang, J., Feng, Y. and Zhu, K. (2015). Attribute reduction approaches for general relation decision systems, *Pattern Recognition Letters* **65**: 81–87, DOI: 10.1016/j.patrec.2015.06.031.

Liu, H. and Setiono, R. (1997). Feature selection via discretization, *IEEE Transactions on Knowledge and Data Engineering* **9**(4): 642–645, DOI: 10.1109/69.617056.

Łuba, T., Borowik, G., Kraśniewski, A., Rutkowski, P. and Ługowska, I. (2014). Application of logic synthesis algorithms for data mining in medical databases, *9th International Seminar Statistics and Clinical Practice, Warsaw, Poland*, pp. 36–39.

Łuba, T. and Rybnik, J. (1992). Intelligent decision support: Handbook of applications and advances of the rough sets theory, *in* S.Y. Huang (Ed.), *Rough Sets and Some Aspects of Logic Synthesis*, Springer Netherlands, Dordrecht, pp. 181–199, DOI: 10.1007/978-94-015-7975-9_13.

Martinović, G., Bajer, D. and Zorić, B. (2014). A differential evolution approach to dimensionality reduction for

classification needs, *International Journal of Applied Mathematics and Computer Science* **24**(1): 111–122, DOI: 10.2478/amcs-2014-0009.

Min, F., Hu, Q. and Zhu, W. (2014). Feature selection with test cost constraint, *International Journal of Approximate Reasoning* **55**(1Pt2): 167–179, DOI: 10.1016/j.ijar.2013.04.003.

Nguyen, H.S. (2006). Approximate Boolean reasoning: Foundations and applications in data mining, *in* J.F. Peters and A. Skowron (Eds.), *Transactions on Rough Sets V*, Lecture Notes in Computer Science, Vol. 4100, Springer, Berlin/Heidelberg, pp. 334–506, DOI: 10.1007/11847465_16.

Petrick, S.R. (1956). A direct determination of the irredundant forms of a Boolean function from the set of prime implicants, *Technical report AFCRC-TR-56-110*, Air Force Cambridge Research Center, Cambridge, MA.

Predki, B., Słowiński, R., Stefanowski, J., Susmaga, R. and Wilk, S. (1998). ROSE—software implementation of the rough set theory, *in* L. Polkowski and A. Skowron (Eds.), *Rough Sets and Current Trends in Computing*, Springer, Berlin/Heidelberg, pp. 605–608, DOI: 10.1007/3-540-69115-4_85.

Predki, B. and Wilk, S. (1999). Rough set based data exploration using ROSE system, *in* Z.W. Raś and A. Skowron (Eds.), *Foundations of Intelligent Systems: 11th International Symposium*, Springer, Berlin/Heidelberg, pp. 172–180, DOI: 10.1007/BFb0095102.

Sasao, T. (2011). Index generation functions: Recent developments, *41st IEEE International Symposium on Multiple-Valued Logic, Tuusula, Finland*, DOI: 10.1109/ISMVL.2011.17.

Sasao, T. (2015). Index generation functions, *EPFL Workshop on Logic Synthesis & Verification, Lausanne, Switzerland*.

Skowron, A. and Rauszer, C. (1992). The discernibility matrices and functions in information systems, *in* R. Słowiński (Ed.), *Intelligent Decision Support—Handbook of Applications and Advances of the Rough Sets Theory*, Kluwer Academic Publishers, Dordrecht, pp. 331–362.

Stefanowski, J., Krawiec, K. and Wrembel, R. (2017). Exploring complex and big data, *International Journal of Applied Mathematics and Computer Science* **27**(4): 669–679, DOI: 10.1515/amcs-2017-0046.

Steinbach, B. and Posthoff, C. (2012). Improvements of the construction of exact minimal covers of Boolean functions, *in* R. Moreno-Díaz *et al.* (Eds.), *Computer Aided Systems Theory—EUROCAST 2011*, Lecture Notes in Computer Science, Vol. 6928, Springer, Berlin/Heidelberg, pp. 272–279, DOI: 10.1007/978-3-642-27579-1_35.

Steinbach, B. and Posthoff, C. (2013). Fast calculation of exact minimal unate coverings on both the CPU and the GPU, *in* R. Moreno-Díaz *et al.* (Eds.), *Computer Aided Systems Theory*, Springer, Berlin/Heidelberg, pp. 234–241, DOI: 10.1007/978-3-642-53862-9_30.

Su, M.-Y., Yu, G.-J. and Lin, C.-Y. (2009). A real-time network intrusion detection system for large-scale attacks based on an incremental mining approach, *Computers & Security* **28**(5): 301–309, DOI: 10.1016/j.cose.2008.12.001.

Sun, L., Xu, J. and Li, Y. (2014). A feature selection approach of inconsistent decision systems in rough set, *Journal of Computers* **9**(6): 1333–1340, DOI: 10.4304/jcp.9.6.1333-1340.

Szemenyei, M. and Vajda, F. (2017). Dimension reduction for objects composed of vector sets, *International Journal of Applied Mathematics and Computer Science* **27**(1): 169–180, DOI: 10.1515/amcs-2017-0012.

Zhong, N. and Skowron, A. (2001). A rough set-based knowledge discovery process, *International Journal of Applied Mathematics and Computer Science* **11**(3): 603–619.

**Grzegorz Borowik** holds a PhD in computer engineering and draws from more than 15 years of experience as a scientist and a university professor focused mainly on machine learning and big data algorithms. His research interests include computational intelligence, deep learning, optimization techniques, numerical algorithms, IoT, and cybersecurity. He has worked at the crossroads of advanced technology and business innovation since 2012. In 2015 he was nominated by the Polish Ministry of Science and Higher Education as a participant in the "TOP 500 Innovators" Program at UC Berkeley in the United States. In 2016 he was a postdoctoral researcher at the Knowledge Engineering and Discovery Research Institute at the Auckland University of Technology in New Zealand, where he worked on advanced problems of neuroscience. He is the author of an academic textbook for Polish technical universities and an author or a co-author of 90+ (2018) publications in scientific journals as well as conference proceedings. He is currently an assistant professor at the Police Academy in Szczytno and holds an R&D management position at Nethone, a data science company specialized in AI-driven business intelligence and fraud prevention. ORCID: 0000-0003-4148-4817.

Received: 8 August 2017
Revised: 1 February 2018
Accepted: 10 March 2018