Matrix-represented Constraints Satisfaction Methods: Practical Aspects of Their Implementation

Zuenko A.A., Oleinik Yu.A.
Institute of Informatics and Mathematical Modelling
KSC, the Russian Academy of Sciences
Apatity, Russia
zuenko@iimm.ru
yoleynik@iimm.ru

***Abstract*—The paper proposes an original approach to solving the problem of ineffective processing of qualitative constraints of a subject domain in the framework of constraint programming technology. The approach is based on the use of specialized matrix-like structures, providing a "compressed" representation of constraints over finite domains, as well as using author's inference algorithms on these structures. Compared to the prototypes using the typical representation of multi-place relations in a form of tables, the techniques make it possible to more efficiently reduce the search space. The paper presents practical aspects of implementation of user-developed types of constraints and corresponding algorithms-propagators with the help of constraint programming libraries. The algorithms performance has been assessed to clearly demonstrate the advantages of representation and processing of qualitative constraints of a subject domain by means of the above matrix structures.**

***Keywords*—constraint satisfaction problem, constraint programming, constraint propogation, matrix-like representation of constraints, qualitative constraints**

I. Introduction

According to [6] the *constraint satisfaction problem* (CSP) consists of three components: X, D, C.

X – a set of variables $\{X_1, X_2, \ldots, X_n\}$.

D – a set of domains $\{D_1, D_2, \ldots, D_n\}$ where D_i is the domain of variable X_i.

C – a set of constraints $\{C_1, C_2, \ldots, C_m\}$ that specify allowable combinations of the values of variables.

Each domain D_i describes a set of the admissible values $\{v_1, \ldots, v_k\}$ for variable X_i. Each constraint is a pair $< scope, rel >$ where $scope$ – is a set of variables which participate in the constraint and rel – is the relation defininig admissible combinations of values, which the variables from $scope$ can take on.

Constraints can be presented either explicitly, i.e. by enumeration of all the admissible combinations of the values for a set of variables specified, or implicitly, i.e. as an abstract relation supporting two operations: checking if a tuple is an element of the given relation, and enumeration of all the elements of the relation. The second way, in fact, requires specifying the characteristic function of the given relation.

Each *state* in a CSP is defined by an assignment of values to some (partial assignment) or to all the variables (complete assignment):$\{X_i = v_i, X_j = v_j, \ldots\}$. The *solution* of a CSP is complete assignment which satisfies all the constraints.

As an example, consider a CSP. Let $X = \{X_1, X_2\}$. We assume that $D_1 = D_2 = \{a, b, c\}$. Let a set C consists of an only constraint, that is, $C = \{C_1\}$. Constraint C_1 describes that fact that the values X_1 and X_2 must have different values.

The given constraint can be expressed implicitly, that is:

$$C_1 =<< X_1, X_2 >, X_1 \neq X_2 > . \quad (1)$$

The same constraint can be expressed explicitly, that is:

$$C_1 =<< X_1, X_2 >, \{< a, b >, < a, c >, < b, a >, < b, c >, < c, a >, < c, b >\} > . \quad (2)$$

Note, that constraint (2) can be expressed in a more compressed way:

$$C_1 =<< X_1, X_2 >, \{a\} \times \{b, c\} \cup \cup \{b\} \times \{a, c\} \cup \{c\} \times \{a, b\} > . \quad (3)$$

There is a Table in Fig. 1a, which vividly represents expression (2). Figure 1b shows a matrix corresponding to expression (3). In fact, in case of a matrix representation (Fig. 1b), the sign

X_1	X_2
a	b
a	c
b	a
b	c
c	a
c	b

a)

X_1	X_2
$\{a\}$	$\{b, c\}$
$\{b\}$	$\{a, c\}$
$\{c\}$	$\{a, b\}$

b)

Figure 1. The tabular constraint representation (a); the constraint representation in the form of specialized matrix (b).

of operation $\times$ (Cartesian product) between the components of one row is omitted, and the sign of operation $\cup$ between rows (union of sets) is not written explicitly. In [4], the similar representation is referred to as an "compressed" representation of the relation.

Unlike the article mentioned above, this paper concerns two types of matrix structures to represent constraints: C-systems and D-systems. In [3], the set-theoretical operations with the given structures are introduced. The similar structures are also used in [8] to solve pattern recognition and knowledge base compression problems.

Expression (3) can be represented in the form of the C-system:

$$C_1[X_1X_2] = \left[\begin{array}{cc} \{a\} & \{b,c\} \\ \{b\} & \{a,c\} \\ \{c\} & \{a,b\} \end{array} \right]. \quad (4)$$

D-systems allow calculating the complement of the C-systems: a complement is taken for each component-set.

Let's assume that we have a constraint $C_1[X_1X_2]$ meaning that $X_1 \neq X_2$. It is necessary to express the constraint $X_1 = X_2$. Then it is possible to represent D-system as follows:

$$\overline{C_1}[X_1X_2] = \left] \begin{array}{cc} \{b,c\} & \{a\} \\ \{a,c\} & \{b\} \\ \{a,b\} & \{c\} \end{array} \right[. \quad (5)$$

The D-system representation is equivalent to the expression:

$$\overline{C_1} =<< X_1, X_2 >, \{[(D_1 \backslash \{a\}) \times D_2 \cup D_1 \times \times (D_2 \backslash \{b,c\})] \cap [(D_1 \backslash \{b\}) \times D_2 \cup D_1 \times (D_2 \backslash \{a,c\})] \cap \cap [(D_1 \backslash \{c\}) \times D_2 \cup D_1 \times (D_2 \backslash \{a,b\})] > . \quad (6)$$

or

$$\overline{C_1} =<< X_1, X_2 >, \{[\{b,c\} \times D_2 \cup D_1 \times \{a\}] \cap \cap [\{a,c\} \times D_2 \cup D_1 \times \{b\})] \cap \cap [\{a,b\} \times D_2 \cup D_1 \times \{c\}] > . \quad (7)$$

The D-system represented can also be expressed as an intersection of three C-systems of the same scheme $< X_1, X_2 >$ namely $\overline{C_1} = K_1[X_1X_2] \cap K_2[X_1X_2] \cap K_3[X_1X_2]$:

$$\left[\begin{array}{cc} \{b,c\} & * \\ * & \{a\} \end{array} \right] \cap \left[\begin{array}{cc} \{a,c\} & * \\ * & \{b\} \end{array} \right] \cap \left[\begin{array}{cc} \{a,b\} & * \\ * & \{c\} \end{array} \right]. \quad (8)$$

In specifying C-systems, a designation "$*$" (a complete component) may be used, which is equivalent to the indication of domain of the corresponding variable.

There is one more type of dummy components – a empty component (designated as "$\varnothing$"), that is a component containing no value.

Now we shall try to answer the question: "When is the representation and handling of a CSP in a kind of C- and D-systems capable to ensure the highest computing performance". In other words: "In what cases should CSPs be represented and processed as the C- and D-systems?"

Most constraint programming environments are mainly oriented on processing of numerical constraints, which are specified by means of the base set of arithmetical operations, binary relations equal/unequal, more/less, built-in functions, etc. for which specialized procedures-propagators are developed. This lack of balance between tools used in quantitative and qualitative constraints processing can be seen even at a level of languages used to define constraints, which are used by different programming libraries. Thus, for instance, in the Choco library [2], it is impossible to directly specify the constraint $x =$ "a", without having substituted a symbol "a" for a number.

The authors' studies showed that processing of qualitative constraints represented in the form of logical expressions and rules, is not sufficiently effective in the systems like those mentioned above, and cannot be implemented for comprehensible time even at a rather small dimension of problem.

The paper presents practical aspects of implementation of user-developed types of constraints and corresponding algorithms-propagators with the help of specialized constraint programming libraries (the Choco library taken as an example). The algorithms performance has been assessed to clearly demonstrate the advantages of representation and processing of qualitative constraints of a subject domain by means of the above matrix structures.

II. Defining CSPs and algorithms of their solution by means of modern tools of constraint programming (the Choco library taken as an example)

The main techniques used in the solution of CSPs can be classified into three classes [7]. Class 1 includes different variants of backtracking search algorithms, which construct a solution by means of extension of partial instantiation step by step, using various heuristics and applying intelligent strategies to recover from the dead ends of a search tree. Class 2 includes algorithms of constraints propagation which eliminate some non-solution elements from the search space, decreasing the dimension of problem. The algorithms themselves do not form the solution because they eliminate not all the non-solution elements. They are used either to pre-process the problem before another type of algorithm is applied, or interwoven with steps of another kind of algorithms to boost its performance. Finally, the structure-driven algorithms exploit the structure of the primal or dual graph of the problem. There are very different algorithms in this class, including ones which decompose the initial CSP into loosely-coupled subproblems, which can be solved by methods from the previous two classes. Hence, structure-based methods can be also coupled with some other types of algorithms.

Constraints programming environments allow usage of built-in types of constraints and the algorithms of their satisfaction and make it possible to develop original types of constraints, methods of their propagation, as well as construct original search strategies.

The following libraries are most widely used in constraints programming: Choco and JaCoP for Java, as well as GeCode and Z3 for C++.

To implement the original algorithms of inference on constraints that are presented as specialized matrix structures, choice was made of the Choco library. Choco library is the

open source software created to define and solve constraint satisfaction problems [5].

To describe CSPs by means of the Choco library, the following basic abstractions are used:

- Model;
- Variable;
- Constraint;
- Propagator;
- Search Strategy;
- Solver;
- Solution.

A. Model.

Abstraction "*Model*" is presented by a special class of the Choco library, on the basis of which all the further defining of the CSP is formed.

B. Variable.

In the library Choco, variables are represented by specialized classes, depending on their type. There are determined four different types of variables in the library:

- **Boolean variable** is a variable with two possible values: true/false (0/1).
- **Integer variable** is a variable taking values from a set of integers. The domain of an integer variable can be specified as an interval $[a, b]$ (bounded domain). Such a representation consumes a small amount of memory, but does not allow processing the gaps in domains. In other words, it is impossible to eliminate an inadmissible value being inside the interval. Another way to specify domains of integer variables is to explicitly enumerate all possible values of a variable (enumerated domain). In so doing, the values should be linearly ordered.
- **Set variable** is a variable whose values are sets of integers. The variable of this type is specified by two sets, i.e. by the upper and lower boundary. It can take the values which are the subsets of the upper boundary and necessarily includes the lower boundary.
- **Real variable** is a variable taking values from the specified interval with the specified precision. At present this type of variables is supported rather poorly in the library.

The variables are added into the defining of a CSP either by means of the methods of class "*Model*", which associates the variables with the corresponding model, or by means of classes implementing the interfaces of creating the variables of corresponding types. The interfaces associate the variables with the model specified in the class constructor. Thus, variables cannot exist by themselves and should be necessarily associated with the model.

C. Constraint.

A certain logical formula which specifies admissible combinations of values of variables, is referred as a constraint. In the Choco library, a constraint is defined by a set of variables and by propagators (filtering algorithms) which delete the values from the specified variables domains, which do not correspond to the legal assignments. The library contains a set of built-in constraints, for example, global ones, *Alldifferent* in particular, i.e. a constraint meaning that the values of variables in a solution, should differ. Also presented in it are various arithmetical constraints; constraints presented as logical formulae; as rules, etc. There are standard methods-propagators for each built-in type of constraints.

An example of an arithmetical constraint:

model.arithm(*var,"=",*5).

This constraint means that variable var should take the value equal to 5.

The methods of class "*Model*" allow constructing more complicated constraints.

For example:

model.or(*new Constraint*[]{*model.arithm*(*var,"=",*5), *model.arithm*(*var,"=",*6)}).

This constraint means that the variable var takes values 5 or 6, and so it takes, as a parameter, an array of constraints between which an OR-operation is set.

The library also allows constructing original constraints and propagators. To create original program class of constraint, it is necessary to specify propagator (one or several) and set of variables, over which the constraint is set.

An example of user-developed constraint creation:

Constraint c=new Constraint(*"My",new Dpropogator*(*vars*));

"My" – string-name of the constraint;

new Dpropogator(*vars*) – a propagator for a constraint over variables vars (there is a possibility to specify several of these).

For the constraint to be taken into account in solution, it is necessary to call a method "*post*()" after the constraint has been created, otherwise, it will not be taken into account in solution.

```
Constraint C1 = model.arithm(var,"=",5);
Constraint C2 = model.arithm(var,"=",6);
Constraint C3 = model.or(new Constraint[]{C1,C2});
```

Figure 2. Example of constraint creation by means of Choco library.

There are three constraints created on Fig. 2 but the method "*post*()" has been called in no one yet. If *C*1.*post*() is to be called, there will be only value "5" for variable *var* in the solutions. Thus, if it is necessary to add constrain "*var* = 5 or *var* = 6" into the solution, the method *post*() should be called only for constraint *C*3.

D. Propagator.

Abstraction "*Propagator*" is specified in the Choco library as class. Constructing class of propagator, it is necessary to define two main methods: a method *propagate* and a method *isEntailed*. The method *propagate* implements the logic of the constraint propagation. During the propagation, the method will be called repeatedly till it causes changes in variable domains. The method *isEntailed* is called at the end of propagation. The method *isEntailed* can return three parameters:

- *ESat.TRUE* – the constraint is completely satisfied;
- *ESat.UNDEFINED* – the constraint status failed to be determined (the propogation ends but the constraint has not been satisfied);
- *ESat.FALSE* – the constraint cannot be satisfied, backtrack is necessary.

Also, in the course of propagation there may appear contradiction exception. In this case, the propagation is considered to be completed ahead of the schedule, and the propagation result is considered to be equivalent to *ESat.FALSE*.

E. Search strategies.

If the final solution has not been reached with the help of propagators, the search space is further studied in accordance with a certain search strategy. In fact, the search strategy defines the way the CSP solution should be constructed.

The Choco Version 4.0.0 constructs a binary search tree (for example, if assignment "$x = 5$" cannot be extended to a solution, then "$x \neq 5$" is considered). The search strategy like this, which implements backtracking search, is typical for CSPs. However, other search strategies shouldn't be neglected.

The strategy correctly selected for a certain problem allows the solution to be generated much faster. The types of strategies are related to the types of variables, i.e. each type is supplied with a specific set of strategies. Like in case with constraints, the Choco library provides built-in program classes for popular strategies. However, if necessary, user-developed search strategy may be constructed. To describe a user-developed strategy, it is necessary to additionally define two (three, optionally) classes: a strategy to select a particular variable from a set of the CSP variables (Class 1), a strategy to select a particular value of the specified variable from its domain (Class 2), and, optionally, a class implementing the strategy to select the branch of a search tree (Class3).

Let's characterize Class 3 separately. It is necessary to define two basic methods here. The first one, *apply*(*IntVar*, *int*), is the method for processing a pair <*a variable*, *its value*> obtained as a result of the variable selection strategy application and the variable value selection strategy application. The second, *unapply*(*IntVar*,*int*), is the method specifying how modifications introduced by the method *apply* should be droped. If the search-tree branch selection strategy is not determined, the method *apply* assigns the selected value to the specified variable, trying to extend the partial assignment to a complete one. Having worked out this variant, the method *unapply* eliminates the considered value from the domain of the corresponding variable. The search proceeds in alternative directions.

F. Solver.

"*Solver*" is an abstraction presented by the class, which keeps the stages of solution process, search strategies and the CSP solutions if those have been reached. The Solver type object is the field of class "*Model*". After the necessary solver parameters have been specified, its method *solve*() is called to start searching.

G. Solution.

A "*Solution*" is an abstraction presented by the Choco library class, which serves as a storage of a complete or partial assignment.

III. The original approach to implementation of inference procedures

Based on the Choco library, the original classes have been developed, extending the functional of the basic library in order to represent and solve the CSPs in the form of a D-systems set. Unlike the standard constraints of the Choco library and their propagators working with assignments of variables, the propagator for the D-systems works with the system components containing several values. In the process of inference, the D-system is reduced and the amount of the information processed decreases with each iteration.

Due to the requirements concerning the length of the paper, no detailed consideration is given to the methods of inference. Given in [9], [10] are the particular techniques to be used to solve the CSPs. These techniques are based on matrix representation of constraints over finite domains.

To explain the approach based on the similar inference, we shall consider that the CSP constraints may be represented in the form of a D-systems set. In the practical problems, it is a set of the C- and D-systems, numerical constraints, as well as that of global constraints [1], [7].

So, let's consider the affirmations which allow implementation of the equivalent CSP transformations for the case under consideration (constraints propagation). The aim of transformations is to reduce a CSP to a simpler form, with the less number of D-systems, the less number of rows of D-system, columns (attributes) of D-system, values in the attributes domains, values in separate components, etc.

- **Affirmation 1**. If, at least, one tuple (row) of the D-system is empty (all components of the tuple are empty), the D-system is empty (the corresponding system of constraints is inconsistent, the CSP has no solution).
- **Affirmation 2**. If all the components of an attribute are empty, the attribute can be eliminated from the D-system (all the components in the corresponding column are removed) and the pair "the eliminated attribute - its domain" is included into the partial solution.
- **Affirmation 3**. If in the D-system there is a tuple (row) containing only one nonempty component, all the values not included into this component, are deleted from the corresponding domain.
- **Affirmation 4**. If a tuple of the D-system contains, at least, one complete component, this tuple is removed (one can remove the corresponding constraint from the system of constraints).
- **Affirmation 5**. If the component of an attribute of the D-system contains the value not belonging to the corresponding domain, this value is deleted from the component.

IV. Comparison between the algorithms of Choco library and the procedures developed

To determine the effectiveness of the classes and algorithms developed we used the problem of placing n chess queens on an $n \times n$ chessboard so that no two queens threaten each other ("N-Queens problem"), due to the simplicity of its scaling.

The aim of the given example was not to generate solutions of the "N-Queens problem" (one or all possible) as fast as possible. Moreover, the authors realized that, by means of standard numerical constraints, it is possible to define the given problem more implicitly. Taking into account of the chessboard symmetry and constraints propagation techniques based on the interval analysis, allow the solutions to be generated faster than in the procedure described in the study presented. The aim of the analysis made is to demonstrate that qualitative dependencies processing by means of modern constraint programming environments, by the Choco library in particular, is less effective than that by the matrix representation in this paper.

That is why the "N-Queens problem" will be described in the form of a set of qualitative constraints and a comparison will be made between the propagation algorithms performance for the two cases. In the first case, the problem is formulated in the form of logical expressions in the Choco language and standard classes-propagators are used. In the second case, the constraints are formalized in the form of a set of the matrix-like structures suggested. Original constraint propagation algorithms are proposed, on the basis of which own classes-propagators extending the base functional of the Choco library are developed.

Thus, the algorithms compared differ in the qualitative constraints representation and the propagators used. After the propagation has stopped, the search tree branching strategy, typical for Choco, is applied.

Example ("N-Queens" problem). Consider a simplified variant of "N-Queens" problem. In this example, the chessboard size is $n \times n$. It is necessary to find possible variants of four queens placing.

Let's associate the i-th horizontal with variable X_i. Then each variable (attribute) will be defined in the domain as $\{a, b, c, d\}$, where a, b, c, d are the labels of the verticals. As an example, let's formulate the constraint "two queens placed on horizontals 1 and 2 are not threatened by each other" in the form of the D-system:

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} X_1 \\ \{a,b,c,d\} \\ \left] \begin{array}{c} \{b,c,d\} \\ \{a,c,d\} \\ \{a,b,d\} \\ \{a,b,c\} \end{array} \right. \end{array} \begin{array}{c} X_2 \\ \{a,b,c,d\} \\ \left[\begin{array}{c} \{c,d\} \\ \{d\} \\ \{a\} \\ \{a,b\} \end{array} \right. \end{array} \tag{9}$$

In particular, the first row of the given D-system shows that if a queen is on the field a_1(the intersection of the first horizontal and the first vertical), then, in the second horizontal, other queen can occupy fields c_2 and d_2 only. In the language of logic, it is expressed as follows:

$$\begin{array}{l} (x_1 = a) \rightarrow ((x_2 = c) \vee (x_2 = d)) \\ \text{or } \neg(x_1 = a) \vee (x_2 = c) \vee (x_2 = d) \\ \text{or } \neg(x_1 = b, c, d) \vee (x_2 = c) \vee (x_2 = d). \end{array} \tag{10}$$

Comparing different pairs of horizontals, it is possible to write out all the constraints on the inter-relative positioning of all the 4 queens. In our case, the number of constraints is calculated by the formula: 4 C_4^2 (for each pair of horizontals, four constraints are formed, the total number of pairs is – C_4^2), that is $\frac{4^2 \cdot (4-1)}{2}$. For a board of $n \times n$ in dimension, the number of constraints is calculated by the formula: $\frac{n^2 \cdot (n-1)}{2}$. So, the CSP considered can be expressed by a D-systems set (like the D-system shown above) describing to the admissible positions of pairs of queens relative to each other.

In the Choco library, the D-system can be represented as a set of built-in logical constraints of the library. For example, the D-system

$$\begin{array}{c} \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} X_1 \\ \{b,c,d\} \\ \left] \begin{array}{c} \{c,d\} \\ \{b,d\} \\ \{b,c\} \end{array} \right. \end{array} \begin{array}{c} X_2 \\ \{a,b,c,d\} \\ \left[\begin{array}{c} \{d\} \\ \{a\} \\ \{a,b\} \end{array} \right. \end{array} \tag{11}$$

can be described, using the Choco library by encoding the values of variables by integers (Fig. 3).

```
Model testmodel=new Model ("Test");
int []x = {2,3,4},y = {1,2,3,4} ;
IntVar X = testmodel.intVar("X",x);
IntVar Y = testmodel.intVar("Y",y);
Constraint node11=testmodel.or(testmodel.arithm (X,"=",3),
                               testmodel.arithm(X,"=",4));
Constraint node12=testmodel.arithm(Y,"=",4);
Constraint node21=testmodel.or(testmodel.arithm(X,"=",2),
                               testmodel.arithm(X,"=",4));
Constraint node22=testmodel.arithm (Y,"=",1);
Constraint node31=testmodel.or(testmodel.arithm(X,"=",2),
                               testmodel.arithm(X,"=",3));
Constraint node32=testmodel.or(testmodel.arithm (Y,"=",1),
                               testmodel.arithm(Y,"=",2));
Constraint firstrow=testmodel.or(node11, node12);
Constraint secondrow=testmodel.or(node11, node12);
Constraint thirdrow=testmodel.or(node31, node32);
testmodel.and(firstrow,secondrow,thirdrow).post();
```

Figure 3. D-system described by the means of Choco library.

It is clear from Fig. 3 that in order to describe even such a small D-system, a significant number of constraints is required: one constraint per each value of the component (cell) of the D-system, one constraint per each component, constraints uniting components of a row, and one common constraint uniting rows of a system. As the system increases in size in such a representation, the number of constraint also increases essentially, which will take much more time to reach the solution. That is why an original representation of the D-system has been developed to specify it by only one constraint.

Presented below are comparative plots of time required for solutions of "N-Queens problems" for both representations

specified. The units of measure for a vertical scale of plots are milliseconds, i.e. the time to solve the problem. However, the time assigned to the solution, has been limited to 2 minutes (120000 msec). Plots in Fig. 4 show the time spent for reaching the first solution of the "N-Queens problem". The plots are broken when the testing computer memory is exhausted. The plots in Fig. 5 show the time spent for reaching all the solutions of the "N-Queens problem". These are broken in the points in which the program was not able to reach all the solutions at the time assigned.

Figure 4. Reaching the first solution of the "N-Queens problem" (ms/N).

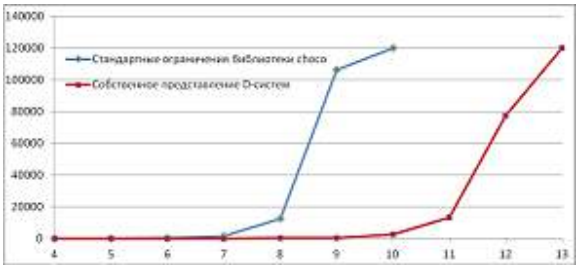

Figure 5. Reaching all the solutions of the "N-Queens problem" (ms/N).

V. Conclusion

The studies have demonstrated that the matrix constraint representation proposed by the authors, as well as the original methods of their propagation are quite suitable for practical application. Moreover, in case of qualitative constraint modelling, the application of the approach proposed gives an essential gain in time against the algorithms of qualitative constraint propagation, which are built in the Choco library. In particular, solving the "N-Queens problem", when it was required to reach all the solutions under the time limit of two minutes, the approach proposed permitted processing the search space of 12^{12} in dimension. In doing so, the standard tools of the Choco library did not give a possibility to study the search space of greater than 9^9 in dimension. When the task was to achieve even one solution for the same two minutes, the time for the standard tools of the Choco library to reach a solution was not enough even when the search space was of 19^{19} in dimension. The methods proposed allowed the authors to reach a solution even for the search space of 76^{76} in dimension.

The authors would like to thank the Russian Foundation for Basic Research (grants No 16-07-00562-a, No 16-07-00377-a, No 16-07-00313-a, No 16-07-00273-a, No 18-07-00615-a) for help in partial of this research.

References

[1] Bartak, R. Constraint Programming: In Pursuit of the Holy Grail. *Proceedings of the Week of Doctoral Students (WDS99)*, Part IV, 1999, pp. 555–564.

[2] Jussien N., Rochart G., Lorca X. Choco: an Open Source Java Constraint Programming Library. *CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)*, 2008, pp. 1-10.

[3] Kulik B.A. A Logic Programming System Based on Cortege Algebra. *Journal of Computer and Systems Sciences International*, 1995, vol. 33, No. 2, pp. 159-170.

[4] More T. Axioms and theorems for a theory of arrays. *IBM Journal of Research and Development*, 1973, No. 17(2), pp. 135–175.

[5] Prud'homme Ch., Fages J-G, Lorca X. Choco Solver Documentation Release 4.0.0. Sep 13, 2016, 37 p.

[6] Russel S., Norvig P. *Artificial Intelligence: A Modern Approach. 3rd edition*. Prentice Hall, 2010, 1132 p.

[7] Ruttkay Zs. Constraint satisfaction a survey. *CWI Quarterly*, 1998, vol. 11, pp. 163–214.

[8] Zakrevskij. A. Integrated Model of Inductive-Deductive Inference Based on Finite Predicates and Implicative Regularities. *Diagnostic Test Approaches to Machine Learning and Commonsense Reasoning Systems IGI Global*, pp. 1-12.

[9] Zuenko A.A. Vyvod na ogranicheniyakh s primeneniem matrichnogo predstavleniya konechnykh predikatov [Constraint inference based on the matrix representation of finite predicates]. *Iskusstvennyi intellekt i prinyatie reshenii* [*Artificial Intelligence and Decision Making*], 2014, No. 3, pp. 21-31.

[10] Zuenko A.A., Lomov P.A., Oleinik A.G. Primenenie metodov rasprostraneniya ogranichenii dlya uskoreniya obrabotki zaprosov k ontologiyam [Application of constraint propagation techniques to speed up processing of queries to ontologies]. *Trudy SPIIRAN* [*SPIIRAS Proceedings*], 2017, No. 1(50), pp. 112-136.

МЕТОДЫ УДОВЛЕТВОРЕНИЯ ОГРАНИЧЕНИЙ, ПРЕДСТАВЛЕННЫХ В МАТРИЧНОЙ ФОРМЕ: ПРАКТИЧЕСКИЕ АСПЕКТЫ ИХ РЕАЛИЗАЦИИ

Зуенко А.А., Олейник Ю.А.

В работе предлагается оригинальный подход к решению проблемы недостаточной эффективности обработки качественных ограничений предметной области в рамках технологии программирования в ограничениях. Подход основан на применении специализированных матрицеподобных структур, обеспечивающих “сжатое” представление ограничений над конечными доменами, а также авторских алгоритмов вывода на данных структурах. По сравнению с прототипами, использующими стандартное представление многоместных отношений в виде таблиц, разработанные методы позволяют более эффективно сокращать пространство поиска. В работе представлены практические аспекты создания пользовательских типов ограничений и алгоритмов их распространения с помощью библиотек программирования в ограничениях. Также было проведено сравнение быстродействия различных алгоритмов, наглядно демонстрирующее преимущества использования описанных матрицеподобных структур для представления и обработки качественных ограничений.