The Comparison of Different SAT Encodings for the Problem of Search for Systems of Orthogonal Latin Squares

Stepan Kochemazov, Oleg Zaikin, and Alexander Semenov

Matrosov Institute for System Dynamics and Control Theory SB RAS, Irkutsk, Russia
{veinamond,zaikin.icc}@gmail.com,biclop.rambler@yandex.ru

Abstract. In this report we present several different propositional encodings for finding systems of mutually orthogonal Latin squares, and evaluate their effectiveness using state-of-the-art parallel and sequential algorithms for solving Boolean satisfiability problem (SAT). We also apply the widely used SMAC tool to study the possibility of improving the effectiveness of lingeling SAT solver on the considered tests and discuss the results of corresponding computational experiments.

Keywords: latin squares, MOLS, SAT, combinatorial designs.

1 Introduction

A Latin square $A = (a_{ij})$ of order n is an $n \times n$ table filled with symbols from the set $N = \{0, 1, ..., n-1\}$, in such a way that each symbol occurs precisely once in each row and each column [5]. Diagonal Latin square is a Latin square, in which each symbol from N occurs precisely once in its main diagonal and in its main antidiagonal. Two Latin squares $A = (a_{ij})$ and $B = (b_{ij})$ are orthogonal if all ordered pairs (a_{ij}, b_{ij}) are distinct. A set of Latin squares, each two of them orthogonal, is called a set of mutually orthogonal Latin squares (MOLS). A set of diagonal Latin squares, each two of them orthogonal, is called a set of mutually orthogonal diagonal Latin squares (MODLS).

Latin squares represent a well studied combinatorial design with rich history dating several centuries back. They have a number of useful properties that make it possible to use them in various areas, such as experimental design, error correcting codes, etc. Also Latin squares are often used as a basis for mathematical puzzles, such as Sudoku and Magical squares. That being said, there remain several hard combinatorial problems related to Latin squares that have not been solved yet. One of the most complex and widely known problem is the one that consists in answering the question if there exists a triple of MOLS of order 10. Even relatively simple problems, such as finding pairs of orthogonal (diagonal) Latin squares of order 10 are relatively hard and there are no known effective methods for solving them.

In this context it is interesting to apply to this class of problems state-of-the-art combinatorial algorithms. One class of such algorithms is formed by the algorithms for solving Boolean satisfiability problem (SAT) [4]. Boolean satisfiability problem is usually formulated as follows: for an arbitrary Boolean formula to answer the question if it is satisfiable. SAT is historically the first NP-complete problem [6], thus it is possible to reduce many combinatorial problems to SAT. What makes it different from many other NP-complete problems is the fact that in the last twenty years there was achieved a remarkable progress in the effectiveness of SAT solving algorithms. Annual SAT competitions [2] make it possible to quickly evaluate new heuristics and ideas using relatively wide class of tests. Today there exist both sequential and multi-threaded implementations of SAT solving algorithms (the corresponding software complexes are usually referred to as SAT solvers). Interesting feature of these algorithms is that since they incorporate many heuristics, each SAT solver has a number of numeric parameters, that can be easily varied. Given a specific solver, on some classes of tests by carefully choosing proper values of parameters it is possible to achieve quite significant performance gain. In this paper we apply state-of-the-art SAT solving algorithms to the problem of finding pairs and triples of MOLS and evaluate their effectiveness.

Let us present the brief outline of the paper. In the next section we provide basic terms about SAT and consider how the problem of finding MOLS can be reduced to SAT. In the third section we describe state-of-the-art tools for finding effective combinations of SAT solver parameters and describe our experimental setup. In the fourth section we present the results of our computational experiments and their discussion. In the last section we make conclusions and outline our plans for the future.

2 Constructing SAT Encodings for Finding MOLS

Interesting and often forgotten fact about SAT encodings is that usually for the same problem we can construct several different variants of them. For example, it can be done via representing the original problem by some equivalent problem. Latin squares can serve as a good example of such approach, because a Latin square of order n as a combinatorial design is equivalent to orthogonal array, a set of n disjoint transversals, a set of 3 orthogonal matrices, etc. It means that in practice to search for Latin squares with specific properties we can search for any of equivalent objects with desired properties adapted to their form. Another way to construct different SAT encodings for the same problem consists in employing different methods for reducing specific constraints to SAT form. A good parallel here can be drawn with sorting methods: in practice we can sort a numerical array via many different methods, and the context defines which one is better. In the present paper we will follow this path. Note, that the problem of finding MOLS with specific properties via SAT was already considered in a number of papers, for example in [14]. However, previous works did not study several significantly different encodings for the same problem and also did not

apply parametrization algorithms to tune solver performance. In the following subsection let us briefly introduce basic notation related to SAT.

2.1 SAT Basics

Boolean formula is constructed from Boolean variables $X = \{x_1, \ldots, x_n\}$, logical operators $\{\neg, \vee, \wedge\}$ and parentheses. A *literal* is either a Boolean variable or its negation. A formula is called *satisfiable* if we can find such an assignment of logical values {TRUE, FALSE} (for simplicity it is usually assumed that TRUE is equivalent to 1 and FALSE is equivalent to 0) to its variables that the result is TRUE. Otherwise the formula is called *unsatisfiable*. An assignment of logical values that makes formula TRUE is called a *satisfying assignment*. Boolean satisfiability problem consists in answering the question if a given formula is satisfiable [4]. In practice a SAT solving algorithm needs to either find a satisfying assignment or to prove that formula is unsatisfiable. Usually, state-of-the-art SAT solving algorithms work with Boolean formulas represented in Conjunctive Normal Form (CNF). CNF is essentially a conjunction of clauses, where by clause we mean a disjunction of several literals or a single (unit) literal. An arbitrary formula can be effectively represented in CNF using Tseitin transformations [11]. In the following subsection we will consider the process of constructing different SAT encodings for finding MOLS in more detail.

2.2 Reducing the Problem of Search for MOLS to SAT

Let us first represent the problem of search for MOLS as a problem of satisfying a set of constraints on Boolean variables. Latin square of order n can be represented as an incidence cube [9] — an $n \times n \times n$ $0-1$ array where dimensions are identified with the rows, columns and symbols of a Latin square. Cell with coordinates (i, j, k) contains 1 if and only if the cell of the corresponding Latin square with coordinates (i, j) contains k. From the definition of Latin square it follows that if we fix two coordinates in incidence cube, then in the remaining 'line' of the cube, produced by varying the remaining coordinate, there should be exactly one 1. Thus to represent the Latin square we use n^3 Boolean variables $\{x(i, j, k)\}$, $i, j, k = 1, \ldots, n$, encoding the incidence cube. Let us introduce the general form of EO predicate: assume that $X = \{x_1, \ldots, x_m\}$ is a set of Boolean variables. Then $EO(X) = EO(x_1, \ldots, x_m) = 1$ if and only if among $x_1, \ldots, x_m$ *exactly one* variable takes the value of TRUE and all remaining variables take the value of FALSE. We will consider the ways this predicate can be transformed to CNF below. Then the Latin square can be specified using following constraints:

$$EO(x(i, j, 1), \ldots, x(i, j, n)), i, j = 1, \ldots, n;$$
$$EO(x(i, 1, k), \ldots, x(i, n, k)), i, k = 1, \ldots, n;$$
$$EO(x(1, j, k), \ldots, x(n, j, k)), j, k = 1, \ldots, n.$$

If we consider a diagonal Latin square, then we add two more constraints on varaibles corresponding to main diagonal and antidiagonal:

$$EO(x(1,1,k),\ldots,x(n,n,k)), k=1,\ldots,n;$$
$$EO(x(1,n,k),\ldots,x(n,1,k)), k=1,\ldots,n.$$

Since we encode the problem of finding MOLS, we also need to specify the orthogonality condition. Let us remind that Latin squares A and B are orthogonal if all ordered pairs of the form (a_{ij}, b_{ij}) are distinct. Assume that variables $\{x^A(i,j,k)\}$ correspond to square A and variables $\{x^B(i,j,k)\}$ correspond to square B. One way (usually referred to as *naive*) to represent orthogonality condition in the form of CNF is to write $\approx n^6$ clauses of the kind:

$$\neg x^A(i_1,j_1,k_1) \vee \neg x^A(i_2,j_2,k_2) \vee \neg x^B(i_1,j_1,k_1) \vee \neg x^B(i_2,j_2,k_2),$$
$$i_1,i_2,j_1,j_2,k_1,k_2 = 1,\ldots,n, i_1 \neq i_2, j_1 \neq j_2.$$

Each of these clauses restricts that the ordered pair (k_1,k_2) appears simultaneously in two different cells with coordinates (i_1,j_1) and (i_2,j_2).

An alternative variant of representing orthogonality condition in CNF relies on the use of auxiliary variables. Assume we use n^2 arrays of n^2 Boolean variables $OC[u,v] = (oc^{u,v}_{1,1},\ldots,oc^{u,v}_{n,n})$, $u,v = 1,\ldots,n$. With each varaible from these arrays we associate the following Boolean equation

$$oc^{u,v}_{i,j} \equiv x^A(i,j,u) \wedge x^B(i,j,v)$$

i.e. if the pair (u,v) is formed by elements of Latin squares with coordinates (i,j) then the corresponding Boolean variable $oc^{u,v}_{i,j}$ takes the value of TRUE. After this the orthogonality condition can be written using the following constraints:

$$EO(OC[k_1,k_2]), k_1,k_2 = 1,\ldots,n.$$

It is clear that by employing different encoding schemes for EO predicate we can produce different SAT encodings for finding MOLS. Let us consider the EO predicate in more detail.

2.3 Encoding *EO* predicate

In recent years there had been published several variants of algorithms for encoding it to SAT. The paper [10] contains quite comprehensive and detailed review of them. Below let us briefly cite the encoding variants described in [10] that we will use in our experiments. Assume that $X = (x_1,\ldots,x_m)$. It is convenient to split the EO predicate into two:

$$EO(X) = AMO(X) \wedge ALO(X)$$

where $AMO(X)$ is the predicate encoding that among the variables from the set X there is *at most one* with the value of TRUE, and $ALO(X)$ is a predicate encoding that *at least one* variable in X takes the value of TRUE. The encoding

of *ALO* predicate to CNF is relatively straightforward – it can be written via one big clause:

$$ALO(x_1, \ldots, x_m) = x_1 \vee \ldots \vee x_m.$$

It is the *AMO* predicate that requires a more thorough consideration. The most simple way to transform it to CNF is the so-called *Pairwise encoding*. It implies the construction of $m \times (m-1)/2$ clauses of the kind

$$\neg v_i \vee \neg v_j, i, j = 1, \ldots m, i \neq j.$$

Pairwise encoding is the only encoding we used that does not employ auxiliary variables for encoding *AMO* predicate.

When we use *Binary encoding* to transform *AMO* predicate to CNF we introduce auxiliary variables $b_1, \ldots, b_{\lceil log_2 m \rceil}$. Then with each variable $x_i \in X$, $i = 1, \ldots, m$ we associate a set of clauses:

$$\bigwedge_{j=1}^{\lceil log_2 m \rceil} \neg x_i \vee \psi(i, j),$$

where $\psi(i, j)$ denotes b_i ($\neg b_i$) if the j-th bit of binary representation of $i-1$ is 1(0). The idea behind the encoding is to create such situation that when any x_i is assigned the value of TRUE it leads to the assignment of all auxiliary variables $b_1, \ldots, b_{\lceil log_2 m \rceil}$ and thus to assigning the value of FALSE to all other x_j, $j \neq i$.

The *Commander encoding* can be considered a meta-encoding method as it can employ other encodings. It is based on dividing the set X into k disjoint subsets $G_1, \ldots, G_k$. Then we introduce auxiliary commander variables $c_1, \ldots, c_k$ that act as representatives of corresponding subsets. After this we write the following set of clauses

$$\bigwedge_{i=1}^{k} AMO(\neg c_i \cup G_i) \wedge \bigwedge_{i=1}^{k} ALO(\neg c_i \cup G_i) \wedge AMO(c_1, \ldots, c_k).$$

The idea is that once a variable x_i becomes TRUE, the corresponding commander variable also becomes TRUE, and thus all x_j, $j \neq i$ are assigned FALSE.

In *Product encoding* we introduce $p + q$ auxiliary variables $U = \{u_1, \ldots, u_p\}$ and $V = \{v_1, \ldots, v_q\}$, $p \times q \geq m$. Then each variable x_k, $k = 1, \ldots, m$ is mapped to a pair (u_i, v_j), such that $k = (i-1)q + j$. Then the following set of clauses:

$$AMO(X) = AMO(U) \wedge AMO(V) \bigwedge_{1 \leq i \leq p, 1 \leq j \leq q}^{1 \leq k \leq m, k=(i-1)q+j} (\neg x_k \vee u_i) \wedge (\neg x_k \vee v_j)$$

guarantees that once x_k is assigned TRUE, the corresponding pair of variables (u_i, v_j) are also assigned TRUE and it leads to assigning FALSE to remaining x_t, $t \neq k$.

The *Sequential encoding* implements the idea of sequential counter, where we traverse the set of variables in the ordered fashion and track if we already

met a variable with the value of TRUE. It employs $m-1$ auxiliary variables $s_1, \ldots, s_{n-1}$ in the following set of clauses:

$$(\neg x_1 \vee s_1) \wedge (\neg x_m \vee \neg s_{n-1}) \bigwedge_{1<i<n} (\neg x_i \vee s_i) \wedge (\neg s_{i-1} \vee s_i) \wedge (\neg x_i \vee \neg s_{i-1}).$$

Finally, the *Bimander encoding* combines the features of Binary and commander encodings in the following way. Similarly to commander encoding the original set X is divided into k disjoint subsets $G_1, \ldots, G_k$, such that each group G_i contains $g = \lceil \frac{m}{k} \rceil$ variables. Then we introduce auxiliary variables $b_1, \ldots, b_{\lceil log_2 m \rceil}$ similar to binary encoding and write the following set of clauses:

$$\bigwedge_{i=1}^{k} \left(AMO(G_i) \wedge \bigwedge_{h=1}^{g} \bigwedge_{j=1}^{\lceil log_2 k \rceil} \neg x_{i,h} \vee \psi(i,j) \right),$$

where $\psi(i,j)$ has the same meaning as in binary encoding.

We applied all considered encoding methods to construct different SAT encodings for several problems of finding pairs and triples of MOLS of various order. The results are presented in Table 1. Since the size difference between SAT encodings for finding MOLS and MODLS is relatively little, we compare only encodings for finding MOLS. The columns NAIVE, PW, BN, CM, PR, SQ, BM correspond to naive, pairwise, binary, commander, product, sequential and bimander encoding schemes, respectively. Note, that we applied them only to encode EO predicates corresponding to orthogonality condition. EO predicates corresponding to constraints on incidence cube were encoded using pairwise encoding in all cases. Below we refer to problem of finding k MOLS (MODLS) of order n as $MOLS\_n\_k$ ($MODLS\_n\_k$). For $n = 8, 9$ we used $p = 3$, $q = 3$ for product encoding and $m = 3$ for bimander and comander encodings, and for $n = 10$ we used $p = 3$, $q = 4$, $m = 3$.

Table 1. Size of SAT encodings for finding MOLS constructed using different AMO encoding schemes, Kb.

Problem	NAIVE	PW	BN	CM	PR	SQ	BM
MOLS_8_2	2 371	2 233	691	660	508	512	727
MOLS_8_3	7 326	6 757	1 952	1 857	1 383	1 420	2 059
MOLS_9_2	5 164	4 415	1 224	1 126	836	864	1330
MOLS_9_3	15 752	13 877	3 551	3 234	2 292	2 336	3 889
MOLS_10_2	10 198	8 343	1 930	1 856	1 328	1 363	2 169
MOLS_10_3	30 893	26 185	5 515	5 269	3 566	3 618	6 287

3 Parametrization of SAT solving algorithms

State-of-the-art SAT solvers are programs with a large number of various parameters. Meticulous tuning of these parameters for a particular class of problems

may significantly improve average effectiveness of the solver: sometimes it is even possible to achieve more than 100 times better performance.

There are some generally accepted techniques for parametrization of SAT solvers. According to papers [8, 7, 1] one should select from the considered set of SAT instances some subset called training set. On this training set the SAT solver with different combinations of parameter values is launched. For each combination of parameter values a special objective function is computed to measure the effectiveness of SAT solving with these values. The parametrization process itself is essentially a local search in a Cartesian product of domains containing all possible values of considered parameters. So all the solver parameters must be discrete (each non-discrete parameter should be discretized). To jump from local optimums sometimes various metaheuristic procedures are used. When the local search scheme finishes its work either due to reaching the optimality condition or because time limit was exceeded, the resulting combination of parameters is applied to solve all problems from the so-called test set. By test set it is usually meant the set of test instances used to measure the effectiveness of the parametrization procedure. Usually it is assumed that test set and training set do not intersect. In practice, given some test set one employs reasonable heuristics to form training set using problems of the same nature and similar dimension as in test set.

There are three state-of-the-art tools for tuning the parameters of SAT solvers: ParamILS (Iterated Local Search in Parameter Configuration Space [8]), GGA (Gender-based Genetic Algorithm [1]) and SMAC (Sequential Model-based Algorithm Configuration [7]). All mentioned tools are based on some local search metaheuristics. GGA uses the genetic algorithm, ParamILS and SMAC are based on the iterative hill climbing algorithm. Using these tools one can speed up both local search and tree search SAT algorithms. With the help of these tools two Configurable SAT Solver Challenges (CSSC) in 2013 and 2014 were held. According to [7] SMAC shows better efficiency compared to ParamILS and GGA. That is why in our computational experiments we used SMAC.

One of the last stages in the development of a SAT solver consists in fixing default values of its parameters. Usually to find such values developers use one of the mentioned tools. During this process the families of SAT instances from the latest SAT competitions are usually used as a training set. One of the reasons of such choice is that these families are formed by SAT encodings of problems from various areas of science. The set of values that has showed the best performance on the training set is finally utilized in the role of the default set of the parameters values. As a result the tuned solver shows good performance in application to the wide class of problems. Meanwhile, in practice one often faces the situation when it is necessary to solve large amount of SAT instances which encode almost identical combinatorial problems. Often such combinatorial problems are obtained by decomposing an original combinatorial problem. In the next section this situation will be described in application to finding systems of MOLS and MODLS.

We chose the LINGELING SAT solver [3] for tuning, because at CSSC 2014 it won the gold medals in all categories (Random SAT, Random SAT+UNSAT, Industrial SAT+UNSAT, Crafted SAT+UNSAT). There are 323 parameters in LINGELING available for tuning. In the next section we will show how this solver works with the default values of parameters on SAT instances built according to the SAT encodings described in the previous section. Then we will describe the results of its tuning in application to some problems which turned out to be very hard.

4 Computational experiments

To compare the effectiveness of SAT solvers on the SAT encodings described in Section 2 we considered the following combinatorial problems:

- MOLS_8_2;
- MOLS_9_2;
- MOLS_10_2;
- MODLS_8_2;
- MODLS_8_3;
- MODLS_10_2;
- MODLS_10_3.

For each of these problems we made all 7 considered SAT encodings. In our experiments besides LINGELING, we used PLINGELING and TREENGELING SAT solvers [3]. We chose them because they won several prizes on the latest SAT competition and SAT Race (they were held in 2014 and 2015 respectively). In particular, we used the versions from SAT Race 2015 for all solvers. Note that PLINGELING and TREENGELING are multi-threaded programs, while LINGELING is a sequential program. We utilized the experimental setup consisting of two 16-core AMD 6276 processors coupled with 64 Gb RAM. Thus, each multi-threaded solver was launched on 32 cores.

On the first stage we launched each of the mentioned solvers with the default values of parameters on every SAT instance (49 SAT instances in total). In this experiment we used the time limit of 5000 seconds for every launch. This limit corresponds to the wall time, so for a sequential solver it is almost identical to CPU time, but in the case of multi-threaded solvers the CPU time can be much greater. It should be noted that this is a standard wall time limit used at SAT competitions. In Tables 2, 3 and 4 the results of the described experiment are shown. Each value in these tables corresopnds to the CPU time in seconds. Here "-" stands for "interrupted due to exceeding the time limit". The best value for each pair (problem, type of the SAT encoding) is marked with bold.

Let us discuss the obtained results. In every case when a solver could cope with a SAT instance, the answer was SATISFIABLE. Problems MODLS_10_2 and MODLS_10_3 turned out to be very hard — no pair (SAT solver, SAT encoding) could cope with them. On the considered test instances two variants of SAT encodings (BM and PW) turned out to be quite mediocre. Interesting

Table 2. CPU time for LINGELING with the default parameters values, seconds.

Problem	NAIVE	PW	BN	CM	PR	SQ	BM
MOLS_8_2	**50**	151	2489	674	1718	158	276
MOLS_9_2	-	-	-	-	-	-	-
MOLS_10_2	**2404**	-	-	-	-	-	-
MODLS_8_2	**50**	200	957	1890	290	847	248
MODLS_8_3	-	-	-	-	-	-	-
MODLS_10_2	-	-	-	-	-	-	-
MODLS_10_3	-	-	-	-	-	-	-

Table 3. CPU time for PLINGELING, seconds.

Problem	NAIVE	PW	BN	CM	PR	SQ	BM
MOLS_8_2	13.7	0.4	25.5	0.2	**0.1**	0.2	620.6
MOLS_9_2	5581	20135	142346	40396	**4516**	15141	-
MOLS_10_2	**9169**	73653	-	-	-	-	38935
MODLS_8_2	1400	512	1272	2850	**91**	851	3617
MODLS_8_3	-	-	-	-	-	-	-
MODLS_10_2	-	-	-	-	-	-	-
MODLS_10_3	-	-	-	-	-	-	-

Table 4. CPU time for TREENGELING, seconds.

Problem	NAIVE	PW	BN	CM	PR	SQ	BM
MOLS_8_2	19	779	4976	536	**9**	643	31
MOLS_9_2	32569	29626	153762	26940	32175	**9290**	53727
MOLS_10_2	18710	102101	-	**6111**	-	-	-
MODLS_8_2	437	739	**391**	1567	1269	2574	698
MODLS_8_3	-	-	-	-	-	**44986**	145142
MODLS_10_2	-	-	-	-	-	-	-
MODLS_10_3	-	-	-	-	-	-	-

Table 5. CPU time of parametrized LINGELING on test sets, seconds.

Problem	NAIVE	PW	BN	CM	PR	SQ	BM
MODLS_10_2, 6 rows known	**5.73**	9.28	9.78	13.30	8.80	11.79	10.83
MODLS_10_3, 6 rows known	**13.52**	29.47	32.02	36.17	35.45	44.34	32.74

fact is that PLINGELING showed superlinear speed up (compared to LINGELING) on MOLS_8_2. On LINGELING in all cases the best results were obtained using NAIVE SAT encoding, however PLINGELING and TREENGELING in almost all cases displayed the best results with other encodings. Finally, the best CPU times for MOLS_8_2 and MOLS_9_2 were obtained by PLINGELING on PR encoding; for MOLS_10_2 and MODLS_8_2 — by LINGELING on NAIVE SAT encoding; for MODLS_8_3 — by TREENGELING on SQ SAT encoding.

In [13] we considered weakened problems for MODLS_10_3 in which the values of the first 45 cells of the first DLS were added to an original CNF via unit clauses. The corresponding SAT instances turned out to be quite hard. In the present paper we considered another type of weakened problems for MODLS_10_3. We constructed 100 SAT instances by adding to an original CNF the values of the first 6 rows of the first DLS via unit clauses. We also made 100 SAT instances for MODLS_10_2 in the similar way. Following [13] we took the corresponding values from the first 50 pairs of MODLS of order 10 found in the volunteer computing project SAT@home [12]. As a result we obtained 100 relatively simple (for LINGELING with the default parameters values) SAT instances for each of the considered problems. We used SMAC (see Section 3) to find good LINGELING parameters values for the obtained families of instances. In every case as the training set we used the first 10 SAT instances from a family, thus the remaining 90 SAT instances formed the test set. In every case SMAC was launched 16 times, each of them for 1 day (with different values of the start seeds). The final performance of the solver on test sets for each type of encoding is presented in Table 5 (the best one out of 16 SMAC results was chosen in every case).

Let us discuss the obtained results. For both considered weakened problems the NAIVE encoding turned out to be better than others. With the help of parametrization the speed-up of 49 % (40 %) was achieved for the weakened problem MODLS_10_2 (MODLS_10_3) on this encoding in comparison with the default parameters values.

5 Conclusion

The results of computational experiments clearly show that it is very important to choose the best pair (SAT solver, SAT encoding) for a particular combinatorial problem. The parametrization of the chosen SAT solver also can provide an additional speed-up. In future we plan to evaluate other types of SAT encodings for the considered problems, such as the ones based on representing Latin squares via orthogonal arrays, sets of transversals, etc. We also plan to utilize other tools for the SAT solvers parameters tuning.

Acknowledgments. The research was funded by Russian Science Foundation (project No. 16-11-10046).

References

1. Ansótegui, C., Sellmann, M., Tierney, K.: A gender-based genetic algorithm for the automatic configuration of algorithms. In: Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming. pp. 142–157. CP'09, Springer-Verlag, Berlin, Heidelberg (2009)
2. Balint, A., Belov, A., Järvisalo, M., Sinz, C.: Overview and analysis of the SAT challenge 2012 solver competition. Artif. Intell. 223, 120–155 (2015)
3. Biere, A.: Lingeling essentials, A tutorial on design and implementation aspects of the the SAT solver lingeling. In: Berre, D.L. (ed.) POS-14. Fifth Pragmatics of SAT workshop, a workshop of the SAT 2014 conference, part of FLoC 2014 during the Vienna Summer of Logic, July 13, 2014, Vienna, Austria. EPiC Series, vol. 27, p. 88. EasyChair (2014)
4. Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (February 2009)
5. Colbourn, C.J., Dinitz, J.H.: Handbook of Combinatorial Designs, Second Edition (Discrete Mathematics and Its Applications). Chapman & Hall/CRC (2006)
6. Cook, S.A.: The complexity of theorem-proving procedures. In: Harrison, M.A., Banerji, R.B., Ullman, J.D. (eds.) Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA. pp. 151–158. ACM (1971)
7. Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: Proc. of LION-5. p. 50723 (2011)
8. Hutter, F., Hoos, H.H., Stützle, T.: Automatic algorithm configuration based on local search. In: Proc. of the Twenty-Second Conference on Artifical Intelligence (AAAI '07). pp. 1152–1157 (2007)
9. Jacobson, M.T., Matthews, P.: Generating uniformly distributed random latin squares. Journal of Combinatorial Designs 4(6), 405–437 (1996)
10. Nguyen, V.H., Mai, S.T.: A new method to encode the at-most-one constraint into sat. In: Proceedings of the Sixth International Symposium on Information and Communication Technology. pp. 46–53. SoICT 2015, ACM, New York, NY, USA (2015)
11. Tseitin, G.S.: Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970, chap. On the Complexity of Derivation in Propositional Calculus, pp. 466–483. Springer Berlin Heidelberg, Berlin, Heidelberg (1983)
12. Zaikin, O., Kochemazov, S., Semenov, A.: SAT-based search for systems of diagonal latin squares in volunteer computing project SAT@home. In: Biljanovic, P., Butkovic, Z., Skala, K., Grbac, T.G., Cicin-Sain, M., Sruk, V., Ribaric, S., Gros, S., Vrdoljak, B., Mauher, M., Tijan, E., Lukman, D. (eds.) 39th International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2016, Opatija, Croatia, May 30 - June 3, 2016. pp. 277–281. IEEE (2016)
13. Zaikin, O., Zhuravlev, A., Kochemazov, S., Vatutin, E.: On the construction of triples of diagonal Latin squares of order 10. Electronic Notes in Discrete Mathematics 54, 307–312 (2016)
14. Zhang, H.: Combinatorial designs by SAT solvers. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 533–568. IOS Press (2009)