Searching for Doubly Self-orthogonal Latin Squares[⋆]

Runming Lu[1,2], Sheng Liu[1,2], and Jian Zhang[1]

[1] State Key Laboratory of Computer Science,
Institute of Software, Chinese Academy of Sciences
{lurm,lius,zj}@ios.ac.cn
[2] Graduate University, Chinese Academy of Sciences

Abstract. A Doubly Self Orthogonal Latin Square (DSOLS) is a Latin square which is orthogonal to its transpose to the diagonal and its transpose to the back diagonal. It is challenging to find a non-trivial DSOLS. For the orders $n = 2$ (mod 4), the existence of DSOLS(n) is unknown except for $n = 2, 6$. We propose an efficient approach and data structure based on a set system and exact cover, with which we obtained a new result, i.e., the non-existence of DSOLS(10).

1 Introduction

Latin squares (quasigroups) are very interesting combinatorial objects. Some of them have special properties. It can be quite challenging to know, for which positive integer n, a latin square of size n (with certain properties) exists. Mathematicians have proposed several construction methods to generate bigger Latin squares from smaller ones. For the small Latin squares, some can be found or constructed by hand easily, but the others are very hard to generate. Computer search methods can be helpful here. In fact, many open cases in combinatorics have been solved by various programs [2,8].

In this paper, we study a special kind of Latin square named *doubly self-orthogonal Latin square* (DSOLS) which is related to the so-called perfect Latin squares [4]. In [1], Du and Zhu proved the existence of DSOLS(n) for $n = 0, 1, 3$ (mod 4), except for $n = 3$. For the orders $n = 2$ (mod 4), the existence of DSOLS(n) is unknown except for $n = 2, 6$. So the existence of DSOLS(10) is the smallest open case.

2 Preliminaries and Notations

A Latin square L of order n is an $n \times n$ table where each integer $0, 1, \ldots, n-1$ appears exactly once in each row and each column. We call each of the n^2 positions of the table a cell. For instance, the position at row x column y is called cell (x, y) and the value of cell (x, y) is denoted as $L(x, y)$, where $x, y, L(x, y)$ all take values from $[0, n-1]$. If $\forall i \in [0, n-1], L(i, i) = i$, L is called an idempotent Latin square.

Two Latin squares L_1 and L_2 are orthogonal if each pair of elements from the two squares occurs exactly once, or alternatively,

$$L_1(x_1, y_1) = L_1(x_2, y_2) \wedge L_2(x_1, y_1) = L_2(x_2, y_2) \rightarrow x_1 = x_2 \wedge y_1 = y_2$$

[⋆] This work is partially supported by the National Natural Science Foundation of China (NSFC) under grant No. 60673044. Corresponding author: Jian Zhang. We are grateful to the anonymous reviewers for their comments, to Lie Zhu and Feifei Ma for their help in this research.

Definition 1. *A DSOLS of order n, denoted as DSOLS(n), is a Latin square A which is orthogonal to both its transpose to the diagonal A^T and its transpose to the back diagonal A^*.*

A DSOLS A of order n can also be characterized using first order logic formulas:

$$A(x, y) = A(x, z) \rightarrow y = z \tag{1}$$

$$A(x, y) = A(z, y) \rightarrow x = z \tag{2}$$

$$A(x_1, y_1) = A(x_2, y_2) \wedge A(y_1, x_1) = A(y_2, x_2) \rightarrow x_1 = x_2 \wedge y_1 = y_2 \tag{3}$$

$$A(x_1, y_1) = A(x_2, y_2) \wedge A(n-1-y_1, n-1-x_1) = A(n-1-y_2, n-1-x_2)$$

$$\rightarrow x_1 = x_2 \wedge y_1 = y_2 \tag{4}$$

Table 1 gives an example of DSOLS(4).

Table 1. A DSOLS(4) and Its two Transposes

0	2	3	1
3	1	0	2
1	3	2	0
2	0	1	3

A

0	3	1	2
2	1	3	0
3	0	2	1
1	2	0	3

A^T

3	0	2	1
1	2	0	3
0	3	1	2
2	1	3	0

A^*

A closely related concept is *doubly diagonal orthogonal latin squares* (DDOLS) [3], which refers to a pair of orthogonal latin squares with the property that each square has distinct elements on the main diagonal as well as on the back diagonal. A DSOLS can be viewed as a special kind of a DDOLS. The existence problem for DDOLS has been solved completely later on.

3 Finding a DSOLS Using SAT and CSP

3.1 SAT Encoding of the Problem

We first encode the problem of finding DSOLS(n) into a SAT instance. Since each cell of Latin square can take one and only one value from the domain $[0, n-1]$, we introduce one boolean variable for each possible value of each cell. For each row $i \in [0, n-1]$, each column $j \in [0, n-1]$ and each candidate value $k \in [0, n-1]$, a boolean variable V_{ijk} is introduced. The variables V_{ijk} should satisfy some inherent constraints. For instance, each cell should take a value from $[0, n-1]$, so we have:

$$\forall i, j \in [0, n-1], V_{ij0} \vee V_{ij1} \vee \cdots \vee V_{ij(n-1)}$$

But each cell should not take more than one values from $[0, n-1]$ at the same time, so $\forall i, j \in [0, n-1]$, we have formulas like the following:

$$\neg V_{ij0} \vee \neg V_{ij1} \quad \neg V_{ij0} \vee \neg V_{ij2} \quad \cdots \quad \neg V_{ij(n-2)} \vee \neg V_{ij(n-1)}$$

Besides these constraints, we also have to encode the problem-specific constraints in formulas 1, 2, 3, 4 into SAT clauses. There can also be other constraints. For instance, we may add unit clauses $V_{iii}(i \in [0, n-1])$ to force the cell (i, i) to take the value i.

Then we send the SAT instance to the state-of-the-art SAT solver MiniSAT. The tool is very efficient. But we still can not solve DSOLS(10) in one week.

3.2 Classical Constraint Satisfaction Problem (CSP)

In [2], Dubois and Dequen employ the CSP model and develop a specific quasigroup generator *qgs* for QG2, another well known combinatorial problem which is quite difficult. With *qgs*, they first proved the non-existence of QG2(10) in 140 days of sequential computation. Due to the similarity of QG2 and DSOLS, we tried to use the technique of *qgs* to find DSOLS(10).

From the CSP's viewpoint, DSOLS can be formulated as a series of overlapping `Alldifferent` constraints in which no two variables involved are allowed to take the same value. For each cell of DSOLS(n), $\forall i, j \in [0, n-1]$, we associate a variable $A(i, j)$ with a domain $D_n = [0, n-1]$. The constraints of DSOLS can be formulated as follows:

$$\text{Alldifferent}\{A(i,j) | i \in [0, n-1]\}, \forall j \in [0, n-1] \tag{5}$$

$$\text{Alldifferent}\{A(i,j) | j \in [0, n-1]\}, \forall i \in [0, n-1] \tag{6}$$

$$\text{Alldifferent}\{A(j,i) | A(i,j) = v\}, \forall v \in [0, n-1] \tag{7}$$

$$\text{Alldifferent}\{A(n-1-j, n-1-i) | A(i,j) = v\}, \forall v \in [0, n-1] \tag{8}$$

When a variable $A(a, b)$ is assigned a value v_0 in the domain D_n, the constraint propagation procedure would be enabled. For example, in every `Alldifferent` constraint in which $A(a, b)$ appears, the value v_0 will be deleted from the domains of the other variables of the constraint. Some `Alldifferent` constraints will be changed dynamically as well based on constraint 7 and 8. For example, the assignment of v_0 to $A(a, b)$ will affect the variables $A(b, a)$ and $A(n-1-b, n-1-a)$ involved in the following constraints

$$\text{Alldifferent}\{A(j,i) | A(i,j) = v_0\}$$

$$\text{Alldifferent}\{A(n-1-j, n-1-i) | A(i,j) = v_0\}$$

Once enabled, the constraint propagation procedure will be carried out recursively until no propagation can be done.

We developed a solver based on the classical CSP model. The basic idea of the solver was similar to the method by Dubois and Dequen [2]. In the implementation, arc-consistency was enforced for the `Alldifferent` constraints, but the propagation method was light-weight and we did not use any novel techniques for constraint propagation. But for the `Alldifferent` constraints 7 and 8, since the variables in the constraints dynamically depend on the values of other cells, we designed a specific propagator to maintain the constraints instead of using a general-purpose CSP solver. We also used some simple techniques like fixing all the cells on the diagonal to break symmetry and tried several common heuristics for selecting variables and for assigning values. With the tool, we still failed to solve the DSOLS(10) problem in one week.

4 An Approach Based on Set System and Exact Cover

Combinatorial objects can also be viewed as set systems [6]. A set system is defined as a collection of subsets of a given set X, $S = \{S_1, S_2, ..., S_m\}$ ($S_i \subseteq X$), which has some additional properties. From the set system's point of view, some combinatorial object searching problems can be formulated as the clique problem [6]. Thus the solution to a set system can be constructed via cliques in certain problem-specific graphs. In the set system representation, lots of inherent symmetries like value symmetry automatically do not exist any more, as compared with other representations. So in some cases, the set system representation may induce surprisingly efficient performance on some combinatorial object searching problem. Based on this observation, Vasquez [7] developed an efficient algorithm for the queen graph coloring problem and obtained some new results. Motivated by this idea, we treat the constraints of DSOLS as a set system.

The set X consists of all cells of the Latin square and $S = \{S_1, S_2, ..., S_{2^{|X|}-1}\}$ is the powerset of X where $S_i \subseteq X(i \in [1, 2^{|X|} - 1])$ (the empty set is not included). For all i, define $S_i^T = \{(y, x)|(x, y) \in S_i\}$ and $S_i^* = \{(n-1-y, n-1-x)|(x, y) \in S_i\}$.

Definition 2 (admissible). *S_i is admissible iff S_i satisfies the following constraints:*

(1) $|S_i| = n$
(2) $\forall (x_1, y_1), (x_2, y_2) \in S_i$, *if* $(x_1, y_1) \neq (x_2, y_2)$, *then* $x_1 \neq x_2$ *and* $y_1 \neq y_2$
(3) *if* $(x, y) \in S_i$ *and* $x \neq y$ *then* $(y, x) \notin S_i$
(4) *if* $(x, y) \in S_i$ *and* $(x, y) \neq (n-1-y, n-1-x)$ *then* $(n-1-y, n-1-x) \notin S_i$

Definition 3 (compatible). *Two elements of S (S_i and S_j, $i \neq j$) are compatible iff*

(1) $S_i \cap S_j = \varnothing$
(2) $|S_i \cap S_j^T| = 1$
(3) $|S_i \cap S_j^*| = 1$

Theorem 1. *There exists a solution to DSOLS(n) if and only if there is a collection of n admissible subsets of X each pair of which are compatible.*

It is easy to prove that the theorem holds. On the one hand, if there exists a DSOLS(n), we can partition the n^2 cells into n sets such that cells are in the same set if and only if they have the same value. According to the definition of DSOLS, it is obvious that these n sets are admissible subsets of X and they are mutually compatible. On the other hand, suppose there are a collection of n admissible subsets of X and they are compatible. The cells in the same set can be assigned with the same value while the cells in different sets can be assigned with different values. The Latin square constructed by these n^2 cells is a DSOLS(n).

A set system of DSOLS(n) can be represented as a graph whose vertices correspond to the admissible sets and edges correspond to the compatible set pairs. Any clique of this graph is also a set system and a clique with n vertices corresponds to a solution of DSOLS(n). If the number of vertices in each maximum clique of the graph is less than n, we can conclude the non-existence of DSOLS(n); otherwise we can conclude the existence of a DSOLS(n).

Algorithm Framework and Data Structure

It is straightforward to use clique algorithms to construct a (partial) solution of a given combinatorial problem which is represented as a set system. If the solution of the combinatorial problem corresponds to the exact cover of the set system, a substantially more efficient algorithm can be utilized because of this property.

Definition 4 (exact cover). *Given a set X and a collection S of subsets of X, an exact cover is a subcollection T of S such that each element in X falls into exactly one element of T.*

T is actually a partition of X. If the element $e \in X$ is contained in the set $t \in T$, then the element e is said to be covered by the set t.

Knuth uses an efficient data structure DLX to index the subsets in the exact cover problem, and develops Algorithm X to solve the classical exact cover problem [5]. In DSOLS(n), all elements in X should also be covered because each cell should be assigned a value. However, a set system of DSOLS(n) is not the general exact cover problem since the compatibility of a pair of subsets requires additional constraints like $|S_i \cap S_j^T| = 1$ and $|S_i \cap S_j^*| = 1$ besides $S_i \cap S_j = \varnothing$. So the constraints of the set system of DSOLS(n) are tighter than those of the general exact cover problem.

Our algorithm for DSOLS(n) includes two steps. First, we enumerate all admissible sets of DSOLS(n), then from these admissible sets, we try to find n sets such that each pair of them are compatible. The framework of our algorithm is given below.

```
 1: if all cells have been labeled then
 2:     A solution is found;
 3:     return ;
 4: end if
 5: Select an unlabeled cell $c$ from the set $X$ and mark $c$ as labeled;
 6: for each subset $S_i$ containing $c$ do
 7:     $S_* = S$;
 8:     $S_* = S_* - \{S_j | S_i \cap S_j \neq \varnothing\}$;
 9:     $S_* = S_* - \{S_j || S_j \cap S_i^T| \neq 1$ OR $|S_j \cap S_i^*| \neq 1\}$;
10:     Search($S_*$);
11: end for
12: restore $c$ as unlabeled;
13: return ;
```

Algorithm 1. The Algorithm Search(S)

It is an exhaustive search algorithm with a filtering scheme to reduce the search space. When the algorithm chooses a subset S_i to constitute the solution, it applies a propagation procedure which removes the remaining subsets not compatible with S_i.

For computational reason, $\{S_j | S_i \cap S_j \neq \varnothing\}$ is removed first since these subsets can be indexed conveniently while removing $\{S_j || S_j \cap S_i^T| \neq 1 \text{ OR } |S_j \cap S_i^*| \neq 1\}$ needs to scan the whole remaining subsets to select the right ones to be removed. Suppose that a subset S_i is chosen to constitute the solution, m subsets remain and r subsets violate the constraint $S_i \cap S_j = \varnothing$, it takes $r \times n$ operations to remove the r subsets which are not compatible with S_i, $(m - r) \times n$ operations to compute the remaining subsets not compatible with S_i and at most $(m - r) \times n$ operations to remove them.

It seems that the computational complexity is very high since the number of admissible sets is huge. However, after selecting a subset S_i, the number of the remaining subsets which are compatible with S_i decreases dramatically. Table 2 shows the average number of admissible subsets at different levels of the backtracking search tree.

Table 2. Average Number of Admissible Subsets in Different Search Levels

	Level									
n	0	1	2	3	4	5	6	7	8	9
7	208.0	12.0	7.0	7.0	7.0	7.0	7.0	-	-	-
8	2784.0	124.8	11.5	8.4	8.2	8.0	8.0	8.0	-	-
9	20556.0	870.8	52.0	9.8	9.1	9.1	9.0	9.0	9.0	-
10	200960.0	8750.3	423.4	40.0	16.0	0	0	0	0	0

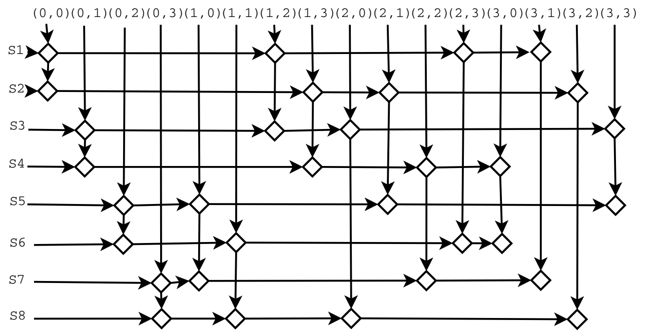

Fig. 1. Data Structure for DSOLS(4)

It does not take much time to enumerate all admissible sets of DSOLS(n) when n is not bigger than 10. Even using a naive method such as enumerating all permutation of n, the computation is only 10!=3628800 steps when $n = 10$. The first column of Table 2 shows the number of admissible sets of DSOLS(n) ($n \leqslant 10$). It is even impossible to store all pairs of these compatible sets in a desktop computer, not to speak of using general clique algorithms including stochastic algorithms to handle such a big graph.

For enumerating all compatible collections of size n in the set system of DSOLS(n), we modify the algorithm for exact cover to take into account the two extra compatibility constraints, $|S_i \cap S_j^T| = 1$ and $|S_i \cap S_j^*| = 1$. A big sparse table is used to index the subsets so that given a cell c it is convenient to access all subsets containing c and given a subset S_i it is convenient to access all cells in S_i. Each cell has a CELL list that contains all subsets covering this cell. Each subset T keeps a list recording the positions of T in the CELL lists besides the cells it covers. For example, DSOLS(4) has 8 admissible subsets ($S_1, S_2, \ldots, S_8$) which are represented in the way shown in Fig. 1.

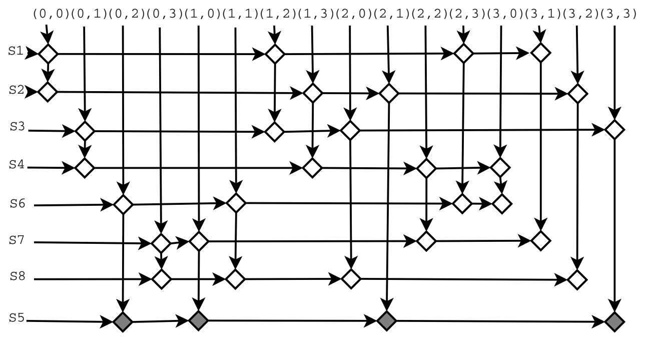

Fig. 2. Data Structure for DSOLS(4) after removing S_5

When removing a subset S_i, S_i in the subsets list and the nodes in the CELL lists representing S_i need to be removed. Instead of deleting these nodes from the lists directly, these nodes are moved to the end of the lists and the length of each list is decreased by 1. For example, after removing S_5 from Figure 1, the updated structure is shown in Figure 2. One benefit of this method is that, when the search needs to restore the subsets during the backtracking, the only operation needed is to recover the length of CELL lists no matter how many subsets had been removed.

5 Experimental Results

Using the algorithm and data structures described in the previous section, we developed a DSOLS generator named DSOLver. To compare the performance of different approaches, we carried out some experiments on a desktop computer (Fedora 10 with 1.5G memory, 2.13GHz CPU, Intel Core2 6400). Table 3 lists the time[1] and branch nodes of MiniSAT, light-weight CSP program (LCSP) and DSOLver on proving the existence of DSOLS(n) where n ranges from 6 to 9. Table 4 lists the time and branch nodes of LCSP and DSOLver on finding all idempotent solutions of DSOLS. Because classical MiniSAT only specializes in finding one solution but not in enumerating all solutions, we do not list its results in Table 4.

These two tables show that DSOLver is better on both CPU time and number of branches. Note that selecting one subset in DSOLver corresponds to assigning n different cells' values in CSP model. So one branch in DSOLver is equivalent to n branches in SAT and CSP. But even if the branch number of DSOLver is multiplied by n, the result is still much smaller than that of SAT and CSP. The reason probably is that each admissible subset has already summed up all the constraints of the n cells involved into one scheme. This kind of scheme can be reused as a package without considering the internal constraint structure again and again as compared with the classical CSP model.

[1] With a different SAT encoding, Hantao Zhang implemented a method which found the first solution of DSOLS(9) in 4 seconds (Private communication with Jian Zhang, July 2010).

Table 3. Comparison of MiniSAT, LCSP and DSOLver on finding one solution

DSOLS(n)		MiniSAT		LCSP		DSOLver	
n	exist?	time(second)	#branches	time(second)	#branches	time(second)	#branches
6	no	0.036	119	0.002	104	0	0
7	yes	0.125	47	0.005	1537	0	7
8	yes	0.716	6140	0.003	856	0.008	33
9	yes	48.201	137083	7.123	3140786	0.071	49

Table 4. Comparison of LCSP and DSOLver on finding all solutions of DSOLS

		LCSP		DSOLver	
n	#solutions	time(second)	#branches	time(second)	#branches
7	64	0.025	8966	0	116
8	1152	8.403	3421189	0.109	4710
9	28608	14515.26	1945836918	28.917	570365
10	0	-	-	83733	606458896

6 Conclusion

Finding a DSOLS is a hard combinatorial search problem. It can be a challenging benchmark for constraint solving and constraint programming. In this paper we discuss how to solve the problem using various techniques with different formulations. We propose an efficient exhaustive search algorithm based on the set system representation and exact cover, with which we obtained a new result, i.e., the non-existence of DSOLS(10). There remain some open cases on the existence of DSOLS(n). In the future, we plan to improve the current algorithm and design new data structures to handle even bigger open instances, e.g., DSOLS(14).

References

1. Du, B., Zhu, L.: Existence of doubly self-orthogonal Latin squares. Bulletin of the Institute of Combinatorics and its Applications (Bulletin of the ICA) 20, 13–18 (1997)
2. Dubois, O., Dequen, G.: The non-existence of (3, 1, 2)-conjugate orthogonal idempotent Latin square of order 10. In: Walsh, T. (ed.) CP 2001. LNCS, vol. 2239, pp. 108–120. Springer, Heidelberg (2001)
3. Heinrich, K., Hilton, A.J.W.: Doubly diagonal orthogonal latin squares. Discrete Mathematics 46(2), 173–182 (1983)
4. Kim, K., Prasanna-Kumar, V.K.: Perfect Latin squares and parallel array access. SIGARCH Comput. Archit. News 17(3), 372–379 (1989)
5. Knuth, D.E.: Dancing links. Arxiv preprint cs/0011047 (2000)
6. Östergård, P.R.J.: Constructing combinatorial objects via cliques. In: Webb, B.S. (ed.) Surveys in Combinatorics. London Mathematical Society Lecture Note Series, vol. 327, pp. 57–82. Cambridge University Press, Cambridge (2005)
7. Vasquez, M.: New results on the queens_n^2 graph coloring problem. J. of Heuristics 10(4), 407–413 (2004)
8. Zhang, H.: Combinatorial designs by SAT solvers. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, vol. 2. IOS Press, Amsterdam (2009)