Streamlined Constraint Reasoning*

Carla Gomes and Meinolf Sellmann

Cornell University
Department of Computer Science
4130 Upson Hall
Ithaca, NY 14853
{gomes,sello}@cs.cornell.edu

Abstract. We introduce a new approach for focusing constraint reasoning using so-called *streamlining constraints*. Such constraints partition the solution space to drive the search first towards a small and structured combinatorial subspace. The streamlining constraints capture regularities observed in a subset of the solutions to smaller problem instances. We demonstrate the effectiveness of our approach by solving a number of hard combinatorial design problems. Our experiments show that streamlining scales significantly beyond previous approaches.

Keywords: constraint reasoning, search, branching strategies

1 Introduction

In recent years there has been tremendous progress in the design of ever more efficient constraint-based reasoning methods. Backtrack search, the underlying solution method of complete solvers, has been enhanced with a range of techniques. Current state-of-the-art complete solvers use a combination of strong search heuristics, highly specialized constraint propagators, symmetry breaking strategies, non-chronological backtracking, nogood learning, and randomization and restarts.

In this paper, we propose a novel reasoning strategy based on *streamlining constraints*. This work has been inspired by our desire to solve very hard problems from combinatorial design. Combinatorial design is a large and active research area where one studies combinatorial objects with a series of sophisticated global properties [5]. For many combinatorial design problems it is not known whether the combinatorial objects under consideration even exist. As a concrete example, we have been studying an application in statistical experimental design. The application involves the design of experiments in such a way that the outcomes are minimally correlated. Mathematically, the task can be formulated as the problem of generating totally spatially balanced Latin squares. To give the reader a feel for the intricacy of the global constraints of this problem, we first give a brief description.

* This work was supported in part by the Intelligent Information Systems Institute, Cornell University (AFOSR grant F49620-01-1-0076).

A Latin square of order N is an N by N matrix, where each cell has one of N symbols, such that each symbol occurs exactly once in each row and column. (Each symbol corresponds to a "treatment" in experimental design.) In a given row, we define the distance between two symbols by the difference between the column indices of the cells in which the symbols occur. Note that each symbol occurs exactly once in each row. So, in a given row, all possible pairs of symbols occur, and for each pair of symbols, we have a well-defined distance between the symbols. For each pair of symbols, we define the total distance over the Latin square as the sum of the distances between the symbols over all rows. A totally spatially balanced Latin square is now defined as a Latin square in which all pairs of symbols have the same total distance [9]. Given the complexity of the total balancing constraint, finding spatially balanced Latin squares is computationally very hard, and, in fact, the very existence of spatially-balanced Latin squares for all but the smallest values of N is an open question in combinatorics.

We performed a series of experiments with both a highly tailored local search method and a state-of-the-art constraint programming approach (ILOG implementation using AllDiff constraints, symmetry breaking, randomization, restarts, and other enhancements). Both our local search method and our CSP-based method could find solutions only up to size $N = 9$. Using the approach introduced in this paper, called streamlining, we can solve considerably larger instances, namely up to size $N = 18$. This is a significant advance given the size of the search space and the complexity of the constraints. To the best of our knowledge, no other method can solve instances of this size.

We will discuss a similar advance in terms of generating diagonally ordered Magic Squares [11, 18]. A Magic Square of order N is an N by N matrix with entries from 1 to N^2, such that the sum of the entries in each column, row, and the main diagonals is the same. In a diagonally ordered Magic Square the entries on the main diagonals are strictly ordered. By using streamlined constraints, we could boost the performance of a CSP-based method from $N = 8$ to order $N = 19$. Again, we do not know of any other CSP approach that can handle instances of this size. It is worth noting though, that local search approaches perform very well for the magic square problem [4].

The key idea underlying "streamlining constraints" is to dramatically boost the effectiveness of the propagation mechanisms. In the standard use of propagation methods there is a hard limit placed on their effectiveness because it is required that no feasible solutions are eliminated during propagation: Streamlining is a somewhat radical departure from this idea, since we explicitly partition the solution and search space into sections with different global properties. For example, in searching for totally balanced Latin squares, one can start by searching for solutions that are symmetric. We will extend this idea much further by introducing a range of more sophisticated streamlining properties — such as composability — that are well-aligned with our propagation mechanisms, and allow us to scale up solutions dramatically.

Our approach was inspired by the observations that for certain combinatorial design problems there are constructive methods for generating a solution. Such solutions contain an incredible amount of structure.[1] Our conjecture is that for many other intricate combinatorial problems — if solutions exists — there will often also be highly regular ones. Of course, we do not know the full details of the regularities in advance. The properties we introduce using the streamlining constraints capture regularities at a high-level. In effect, search and propagation is used to fill in the remaining necessary detail.

By imposing additional structural properties in advance we are steering the search first towards a small and highly structured area of the search space. Since our streamlining constraints express properties consistent with solutions of smaller problem instances, we expect these subspaces to still contain solutions. Moreover, by selecting properties that match well with advanced propagation techniques, while allowing for compact representations, our propagation becomes much more effective with the added constraints. In fact, there appears to be a clear limit to the effectiveness of constraint propagation when one insists on maintaining all solutions. The reason for this is that the set of all solutions of a combinatorial problem often does not have a compact representation, neither in theory nor in practice. For example, in the area of Boolean satisfiability testing, it has been found that binary decision diagrams (BDDs), which, in effect, provide an encoding of all satisfying assignments, often grow exponentially large even on moderate size problem instances. So, there may be a hard practical limit on the effectiveness of constraint propagation methods, if one insists on maintaining the full solution set. Streamlining constraints provide an effective mechanism to circumvent this problem.

The paper is organized as follows: In the next section, we introduce the idea of streamlining constraints. In section 3, we present our streamlining results for the diagonally ordered magic squares. In section 4, we discuss a streamlining approach for totally spatially balanced Latin squares. Finally, we conclude in Section 5.

2 Streamlining

In *streamlining*, we partition the search space of a problem P into disjoint sub-problems with respect to a specific set ($\mathcal{S}$) of solution properties that are computationally interesting: Sub-problem P_1 that corresponds to problem P with the additional constraint that $\mathcal{S}$ holds; and sub-problem P_2, the complement of the sub-problem P_1, with respect to the property $\mathcal{S}$. Since the addition of the constraints enforcing S shapes the problem in a favorable way, we name them

[1] Note that such constructions are far from "straightforward". For example, the great Euler extensively studied Latin squares and considered a special case, so-called orthogonal Latin squares (OLS). He conjectured that there did not exist solution for orthogonal Latin squares for an infinite number of orders. It took more than a century until a construction for all orders (except 2 and 6 for which it was proven that no solution exists [14]) was developed [3]. Until today, for the mutually orthogonal Latin square problem (MOLS), which is a generalization of OLS, no construction is known — despite the valiant efforts undertaken by mathematicians in combinatorial design theory.

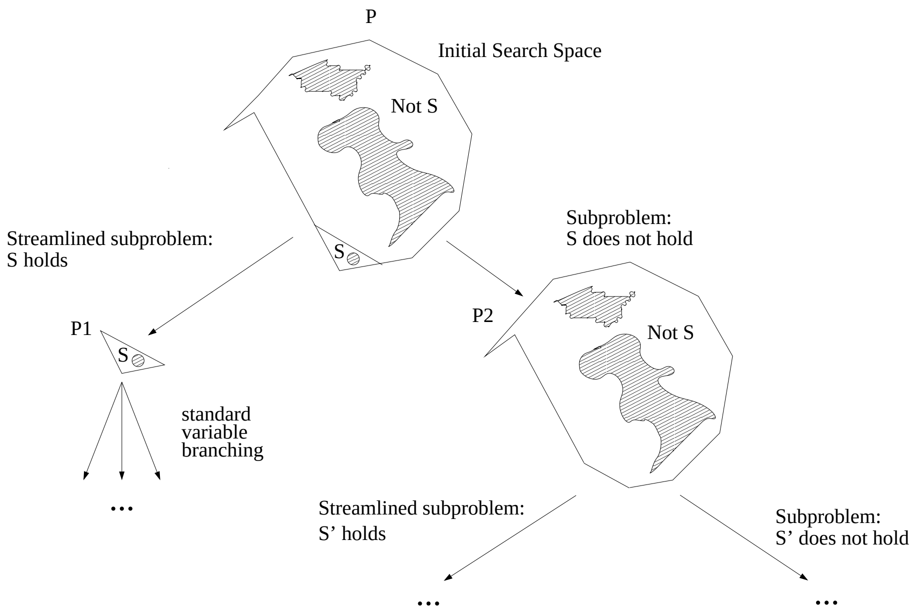

Fig. 1. Streamlining to focus the search on a small, structured part of the search space with hopefully high solution density.

streamlining constraints (or *streamliners*) and the resulting sub-problem P_1 the *streamlined sub-problem*. We can further streamline the resulting sub-problems by considering additional solution properties (see Fig.1).

In general, the search space for problem P_1 will be substantially smaller than that of its complement P_2. However, more importantly, we look for sub-spaces where our constraint propagation methods are highly effective.

Instead of a description in terms of a partitioning of the solution space, one can also describe the effect of streamlining constraints directly in terms of search. For backtrack search, streamlining constraints can be viewed as a strong branching mechanism, used at high levels of the search tree: Streamlining constraints steer the search toward portions of the search space for which streamlined properties hold. As a general global search algorithm, the process can be described as a tree search in which a node represents a portion of the search space and an arc represents the splitting of the search space by enforcing a given property. The top node of the tree represents the entire search space, containing all the solutions to the given problem instance. At the top level(s) of the search tree, we split the search space into sub-sets by enforcing streamlining constraints. For a given node, the branching based on a streamlining constraint originates two nodes: the left node, the first to be explored by the search procedure, corresponds to a streamlined sub-problem for which the streamlining constraint is enforced; the right node corresponds to the complement of the left node with respect to the parent node and the property enforced by the streamlining constraint. Once the search reaches a streamlined sub-problem, further branching decisions may be done based on additional streamlining constraints or using standard branching strategies. If the portion of the search space that corresponds to the streamlined sub-problem P_1 (resulting from enforcing property $\mathcal{S}$) is searched exhaustively without finding a solution, backtracking occurs to the node that is the complement of the streamlined sub-problem, *i.e.*, P_2, for which property $\mathcal{S}$ does not hold.

The selection of streamlining constraints is based on the identification of properties of the solution space. Note that we can view redundant constraints as a particular case of streamlining constraints, in which the streamlining property considered holds for all the solutions. However, redundant constraints, while effective since they eliminate portions of the search space that do not contain any solution, are a very conservative example of streamlining constraint reasoning: they do not partition the solution space. In general, streamlining constraints do partition the solution space into disjoints subsets.

The goal of streamlined reasoning is to focus the search on a subspace that is relatively small, highly structured, and that has a good probability of containing a solution. In our current approach, we manually analyze the set of solutions of small problem instances and try to identify properties that hold for a good part of the solutions — without necessarily holding for all the solutions — but that will structure and reduce the remaining search space in this branch considerably. For example, in the case of the problem of generating totally balanced Latin squares, we observed that for instances of small orders, several of the solutions were highly symmetric. So, our streamlining constraint steers the search toward a region of symmetric totally balanced Latin squares. An exciting direction for future work is to develop statistical methods to automatically generate potential streamlining constraints from the solutions of small example instances.

In the selection of streamlining properties there are several tradeoffs to consider that emerge from the conflicting goals of increasing solution density, reducing the search space drastically, and enforcing a structure that can easily be checked: On the one hand, it would be desirable to select properties that hold for most of the solutions (or even for all the solutions, as in redundant constraints) to ensure that many solutions are in the streamlined subproblem. However, the aim to preserve a larger part of the solutions will usually conflict with the goal of structuring and significantly reducing the remaining search space. As we will see later in the paper, more aggressive streamlining strategies, based on selecting properties that hold only for a very small subset of solutions, can in fact be much more effective in steering the search toward a solution in a drastically reduced search space. Another aspect that is important to consider is how well a streamlining constraint propagates: good candidates for streamlining constraints should propagate easily. Finally, we consider the power of a streamlining constraints based on the compactness of the representation of the induced streamlined subproblem. For example, for the problem of totally balanced Latin squares, by choosing a very strong streamliner, we go from a representation that requires N^2 variables to a much more compact representation that requires only N variables.

3 Streamlining Diagonally Ordered Magic Squares

To provide a first illustration of our streamlining approach, we consider the problem of constructing a special class of magic squares. In the next section, we consider the more practically relevant, but also more complex, case of constructing spatially balanced Latin squares.

Definition 1. Diagonally Ordered Magic Squares (DOMS)
Given a natural number n*, let us set* $S = \frac{n(n^2+1)}{2}$.

- *A square* $M = (m_{ij})_{1 \le i,j \le n}$ *with* n^2 *pairwise disjoint entries* $1 \le m_{ij} \le n^2$ *is called a* magic square of order n *iff*
  1. *For all rows, the sum of all entries in that row adds up to* S*, i.e.* $\sum_j m_{ij} = S$ *for all* $1 \le i \le n$.
  2. *For all columns, the sum of all entries in that column adds up to* S*, i.e.* $\sum_i m_{ij} = S$ *for all* $1 \le j \le n$.
  3. *The sum of all entries in each of the two main diagonals adds up to* S*, i.e.* $\sum_i m_{ii} = S$ *and* $\sum_i m_{n+1-i,i} = S$.
- *A magic square is called* diagonally ordered *iff both main diagonals, when traversed from left to right, have strictly increasing values, i.e.* $m_{ii} < m_{i+1,i+1}$ *and* $m_{n+1-i,i} < m_{n-i,i+1}$ *for all* $1 \le i < n$.
- *Given a natural number* n*, the* Diagonally Ordered Magic Squares Problem (DOMS) *consists in the construction of a diagonally ordered magic square.*

While there exist polynomial-time construction methods for standard magic squares, i.e. without the diagonality constraints, no such construction is know for DOMS. We start out by using a standard constraint programming model, similar to the one used [10]. For details, see the experimental section below. Our experiments show that even small instances are already quite difficult. In particular, we did not find any solutions for order nine or higher. By studying solutions of small order, we found that, for a large number of solutions, the largest entries $(n^2 - n, \ldots, n^2)$ and the smallest entries $(1, \ldots, n)$ are fairly evenly distributed over the square — which makes perfect sense, since one cannot have too many large or too many small numbers clustered together in one row or column (see Fig.3). The question arises: how could one formalize this observation?

(M)

6	9	3	16
12	7	13	2
15	4	10	5
1	14	8	11

(D)

			*
		*	
*			
	*		

(L)

2	3	1	4
3	2	4	1
4	1	3	2
1	4	2	3

Fig. 2. Structure of solutions: Square M shows a typical diagonally ordered magic square of order 4. In D, we can see how the n largest numbers are distributed very nicely over the square. By associating symbol 1 with numbers $1, \ldots, 4$, symbol 2 with numbers $5, \ldots, 8$, symbol 3 with numbers $9, \ldots, 12$ and finally symbol 4 with numbers $13, \ldots, 16$, we get the square L. We observe: L is a Latin square!

Given a DOMS M of order n, let us define $L = (l_{ij})$ with $l_{ij} = \lfloor (m_{i,j} - 1)/n \rfloor + 1$. In effect, we associate the numbers $kn+1, \ldots, (k+1)n$ with symbol $k \in [1, \ldots, n]$. If no two numbers of any interval $[kn+1, \ldots, (k+1)n]$ occur in the same row or column of M, one can expect that the numbers are quite evenly distributed over the square, which is exactly what we need to do in a magic square.[2] And if this is the case, then L is a Latin square. Now, in order to boost our search, we use "Latin squareness" as a streamliner: By adding the corresponding constraints on the first level of our backtrack search, we focus on squares that are hiding a Latin square structure.[3]

Although we observed Latin square structure in many solutions of diagonally ordered magic squares of small orders, it is of course not guaranteed that there exist such magic squares for all orders. However, given that the number of magic squares grows quite rapidly with the order, it appears likely that magic squares with hidden Latin square structure exist for all orders. Moreover, we also do not know in advance whether the additional constraints will actually facilitate the search for magic squares. However, as we will see below, our empirical results show that the constraints dramatically boost our ability to find higher order magic squares.

3.1 Experimental Results

For our base constraint programming model, we used an approach similar to the one described in [10]. Each entry of the square is represented by a variable taking values in $\{1, \ldots, n^2\}$. We add an All-Different constraint [12] on all variables and also add the sum-constraints and diagonal constraints as given by Definition 1. For the branching variable selection, we use a randomized min-domain strategy with restarts.

In the streamlined approach, we also need to provide a structure for the hidden Latin square structure. We add the usual variables and constraints, combined with a dual model defined by the column conjugate.[4] Channeling constraints between the Latin square structure and the magic square are of the form:

$$\begin{array}{lcll} m_{ij} > (k-1)n & \Leftrightarrow & l_{ij} \geq k & \forall\, 1 \leq k \leq n \\ m_{ij} \leq kn & \Leftrightarrow & l_{ij} \leq k & \forall\, 1 \leq k \leq n \end{array}$$

The Latin square streamliner comes with additional variables, so we have to take them into account in the subsequent tree search. In the branch of the tree where we impose the streamliner, we first assign the Latin square variables and only then search over the original variables. We follow a randomized min-domain strategy for both types of variables with restarts.

[2] Note that other, less balanced, solutions do exists. However, the whole idea of streamlining is to focus in on a well-structured subset of the solution space.

[3] Since these magic squares have such beautiful hidden structure, we informally call these *Dumbledore Squares*, in reference of the idea of hiding one magic item within another, as the Sorcerer's Stone in the Mirror of Erised [13].

[4] The notion of row conjugate is defined in our description of spatially balanced Latin squares (Section 4).

order	3	4	5	6	7	8	9	10
pure	0.01	0.01	0.03	1.19	5.05	5391	—	—
streamlined	0.03	0.03	0.03	0.11	0.12	0.42	0.55	0.72

Table 1. Solution times in seconds for finding the first diagonally ordered Magic square using the pure and the streamlined CP approach (— means not solved after 10 hours).

order	11	12	13	14	15	16	17
time	37.67	112	140	3432	7419	28.6K	61K
last fails	1430	2380	5017	6879	8494	1162	11.6K
total cps	24K	47K	59K	726K	1.9M	4.2M	9.1M
restarts	5	9	10	106	222	399	663

Table 2. Detailed results for higher order streamlined Magic squares. We give the CPU times in seconds that are required to find the first solution, the number of fails on the final restart, the total number of choice points, and the number of restarts.

Tables 1 and 2 summarize the results of our experimentation. All experiments in this paper were implemented using ILOG Solver 5.1 and the gnu g++ compiler version 2.91 and run on an Intel Pentium III 550 MHz CPU and 4.0 GB RAM.

Table 1 shows how the streamlining constraint leads to much smaller solution times compared to running on only the original set of constraints (two or more orders of magnitude speedup). We also solve a number instances that were out of reach before. Table 2 summarizes the statistics on those runs. We can see clearly that streamlining significantly boosts performance. Since it took a couple of days to compute them, we do not report details on our experiments with orders 18 and 19 here, but we still want to mention that we were able to compute Dumbledore Squares (i.e. diagonally ordered magic squares with a hidden Latin square structure) of those sizes.

4 Streamlining Spatially Balanced Experiment Design

We now discuss an example of streamlining on a combinatorial design problem of significantly higher practical value. In particular, we consider computing spatially balanced Latin squares. As mention in the introduction, these special Latin squares are used in the design of practical agronomics and other treatment experiments where it is important to minimize overall correlations.

Definition 2. **[Latin square and conjugates]** *Given a natural number $n \in \mathbb{N}$, a Latin square L on n symbols is an $n \times n$ matrix in which each of the n symbols occurs exactly once in each row and in each column. We denote each element of L by l_{ij}, $i, j \in \{1, 2, \cdots, n\}$. n is the order of the Latin square.*
[Row (column) conjugate of a given Latin square] *Given a Latin square L of order n, its* row (column) conjugate *R (C) is also a Latin square of order n, with symbols, $1, 2, \cdots, n$. Each element r_{ij} (c_{ij}) of R (C) corresponds to the row (column) index of L in which the symbol j occurs in column (row) i.*
[Row distance of a pair of symbols] *Given a Latin square L, the distance of a pair of symbols (k, l) in row i, denoted by $\mathrm{d}_i(k, l)$, is the absolute difference of the column indices in which the symbols k and l appear in row i.*

[**Average distance of a pair of symbols in a Latin square**] *Given a Latin square L, the average distance of a pair of symbols (k, l) in L is* $\bar{\mathrm{d}}(k,l) = \sum_{i=1}^{n} \mathrm{d}_i(k,l)/n$.

It can be shown that for a given Latin square L of order $n \in \mathbb{N}$, the expected distance of any pair in any row is $\frac{n+1}{3}$ [16]. Therefore, a square is totally spatially balanced iff every pair of symbols $1 \leq k < l \leq n$ has an average distance $\bar{\mathrm{d}}(k,l) = \frac{n+1}{3}$. (Note that in the introduction we slightly simplified our description by considering the total row distance for each pair without dividing by n, the number of rows.) We define:

Definition 3. [**Totally spatially balanced Latin square**] *Given a natural number $n \in \mathbb{N}$, a* totally spatially balanced Latin square *(TBLS) is a Latin square of order n in which* $\bar{\mathrm{d}}(k,l) = \frac{n+1}{3} \quad \forall\, 1 \leq k < l \leq n$.

Fig.3 provides an example illustrating our definitions. In [9] we developed two different approaches for TBLS, one based on local search, the other based on constraint programming. Neither of the two pure approaches is able to compute solutions for orders larger than 9, and only by using a specialized composition technique that works for orders that are multiples of 6, we were able to compute a solution for orders 12 and 18. We will describe this technique in Section 4.3 as another example of streamlining.

(L)	(C)	(R)
1 2 3 4 5	1 2 3 4 5	1 5 4 3 2
5 1 2 3 4	2 3 4 5 1	2 1 5 4 3
4 5 1 2 3	3 4 5 1 2	3 2 1 5 4
3 4 5 1 2	4 5 1 2 3	4 3 2 1 5
2 3 4 5 1	5 1 2 3 4	5 4 3 2 1

Fig. 3. A Latin square L, its row conjugate R, and its column conjugate C. The distance of pair $(1,5)$ in row 1 is $\mathrm{d}_1(1,5) = 4$, in row 2 it is $\mathrm{d}_2(1,5) = 1$, and the average distance for this pair is $\bar{\mathrm{d}}(1,5) = \frac{8}{5}$.

When working on the CP approach, we first tried to use symmetry breaking by dominance detection (SBDD [6,7]) to avoid that equivalent search regions are investigated multiple times. We were surprised to find that a partially symmetry breaking initialization of the first row and the first column yielded far better computation times than our SBDD approach.

We investigated the matter by analyzing the solutions that are found after the initial setting of the first row and column. We found that most solutions exhibited a diagonal symmetry. That is, for a large number of solutions (around 50% when fixing the first column and row) it is the case that $l_{ij} = l_{ji}$. This gave rise to the idea to "streamline" our problem by adding the additional constraint that the solutions we look for should be diagonally symmetric, and then to analyze the set of solutions found in that manner again.

First, we observed that the computation time in the presence of the additional constraint went down considerably. When we then analyzed the newly found set of solutions, we found that all solutions computed were unique in the following sense: not a single pair of solutions to the diagonally symmetric TBLS was symmetric to one another. The question arises: Could we use this fact to streamline our search further?

4.1 Analyzing Solutions to Small TBLS Instances

Before we can formulate the streamlining constraints that evolved out of the observation that all observed solutions to the diagonally symmetric TBLS are unique, let us discuss the inherent symmetries of the problem.

Clearly, any permutation of the rows has no effect on the "Latin squareness" nor does it affect the spatial balance of the square. The same holds when renaming the symbols. Thus, applying any combination of row and symbol permutations to a feasible solution yields a (possibly new) square that we call "symmetric" to the original one.

Now, by enforcing diagonal symmetry of the squares and initializing both the first row and the first column with $1, \ldots, n$, we found that all solutions computed were *self-symmetric*. (Note that this is an empirical observations based on small order instances.) With the term self-symmetric we denote those squares that are only symmetric to themselves when applying any combination of row and symbol permutations that preserves the initial setting of the first row and column. In some sense, one may want to think of these solutions as located on the "symmetry axis".

(A)		1 2 3 4 5	*(B)*	*(C)*	*(D)*
1 2 3 4 5			3 5 2 1 4	1 2 3 4 5	1 2 3 4 5
2 4 5 3 1	ρ_1	5 4 1 2 3	4 3 1 5 2	5 1 4 2 3	2 4 5 3 1
3 5 2 1 4	σ	4 3 1 5 2	5 1 4 2 3	2 4 5 3 1	3 5 2 1 4
4 3 1 5 2	ρ_2	1 5 2 3 4	2 4 5 3 1	3 5 2 1 4	4 3 1 5 2
5 1 4 2 3			1 2 3 4 5	4 3 1 5 2	5 1 4 2 3

Fig. 4. An example for a self-symmetric solution to diagonally symmetric TBLS. B denotes the square that evolves out of A by applying row permutation ρ_1. C shows the result after applying the symbol-permutation σ that re-establishes the initial setting of the first row. Finally, we get D after applying the row permutation ρ_2 that re-establishes the initial setting of the first column. For self-symmetric squares we observe: A=D, no matter how we choose ρ_1.

An example for a self-symmetric square is given in Figure 4. When we start with solution A and consider a permutation of the rows according to the given permutation ρ_1, we get solution B that is symmetric to A in the sense that B is totally balanced if and only if A is totally balanced. If we want to ensure that our solution preserves the pre-assignments in the first row and the first column, we need to apply unique permutations σ and ρ_2 of symbols and columns next. As a result of this operation, we get square D. For self-symmetric squares, we have that A=D, no matter how we choose the initial row permutation. Note that it is actually enough to consider the permutations ρ_1 and σ only. We chose to allow an additional row permutation ρ_2 so that ρ_1 can be chosen arbitrarily.

For the purpose of streamlining the search for solutions to TBLS, the question arises what constraints we could post to focus our search on self-symmetric squares first. For this purpose, we prove a theorem that allows us to judge easily whether a given square is self-symmetric or not. Let us denote with x_i the permutation of symbols that is defined by row i in some square X. Then:

Theorem 1. *A square A that solves the diagonally symmetric TBLS is self-symmetric iff all permutations defined by A commute with each other, i.e.*

$$a_i \circ a_j = a_j \circ a_i \qquad \forall\, 1 \leq i < j \leq n,$$

whereby a_i denotes the i-th row of A, n denotes the order of the square and $\circ$ denotes the composition of permutations.

For the proof of this theorem, we refer to Appendix A. The proof is based on concepts from permutation group theory. To briefly illustrate the theorem, consider the totally spatially balanced Latin square A in Fig. 4. Consider, for example, permutations defined by the second and the third row of A. We obtain $a_2 \circ a_3 = 5\ 1\ 4\ 2\ 3$ which equals $a_3 \circ a_2$.

4.2 Experimental Results

Our approach is based on constraint programming and can be sketched as follows: Every cell of our square is represented by a variable that takes the symbols as values. We use an All-Different constraint [12] over all cells in the same column as well as all cells in the same row to ensure the Latin square requirement. We also keep a dual model in form of the column conjugate that is connected to the primal model via channeling constraints that were developed for permutation problems [17]. This formulation is particularly advantageous given that by having the dual variables at hand it becomes easier to select a "good" branching variable. In order to enforce the balancedness of the Latin square, we introduce variables for the values $\bar{\mathrm{d}}(k,l)$ and enforce that they are equal to $\frac{(n+1)}{3}$.

With respect to the branching variable selection, for Latin square type problems it has been suggested to use a strategy that minimizes the options both in terms of the position as well as the value that is chosen. In our problem, however, we must also be careful that we can detect unbalancedness very early in the search. Therefore, we traverse the search space symbol by symbol by assigning a whole column in the column conjugate before moving on to the next symbol. For a given symbol, we then choose a row in which the chosen symbol has the fewest possible cells that it can still be assigned to. Finally, we first choose the cell in the chosen row that belongs to the column in which the symbol has the fewest possible cells left.

With Theorem 1, we can now easily streamline this approach by guiding our search towards solutions that are both diagonally symmetric and self-symmetric Latin squares. Based on our experience with smaller problem instances, we conjecture that such solutions will also exist for higher orders. So, following our streamlining approach, at the first two levels of our tree search, before choosing the branching variable as described above, we enforce as branching constraints:

1. $a_{ij} = a_{ji}$ for all $1 \le i < j \le n$, and
2. $a_i a_j = a_j a_i$ for all $1 \le i < j \le n$.

order	3	5	6	8	9	11	12	14
pure	0.01	0.02	0.06	16.14	241	—	—	—
streamlined	0.01	0.03	0.05	0.88	0.91	9.84	531	5434

Table 3. Solution times in seconds for finding the first totally spatially balanced Latin square.

Table 3 shows the effect of streamlining. We can see clearly how the new method dramatically reduces the runtime, and allows us to solve much larger instances compared to what was possible with the pure model without the streamlining constraints. It is worth noting here that, for all instances, we found diagonally symmetric, self-symmetric solutions, which provides empirical evidence that such solutions exist if the problem is solvable at all. An interesting challenge is to prove this formally. Note, however, that in order to apply streamlining constraints, one need not have such a guarantee.

4.3 Balanced Square Composition

As mentioned earlier, we have also developed a composition technique that builds a balanced square of order $2n$ using as a building block a balanced square of order n. This method is suitable for orders $2n \mod 6 = 0$. Composition is an extreme (and elegant) case of streamlining. Due to space constraints, we describe the method only briefly.

order	6	12	18
time	0.02	14.36	107K
last fails	1	12K	43K
total cps	2	36K	100M
restarts	0	2	504

Table 4. Results using composition streamlining. We give CPU times in seconds for finding the first solution, the number of fails on the last restart, the total number of choice points, and the number of restarts.

The idea works as follows: Given a totally balanced square A of order n such that $2n \mod 6 = 0$, we denote the columns of A by $A_1, \dots, A_n$. Let us define a shifted copy of A by setting $B = (b_{ij})$ with $b_{ij} = a_{ij} + n$. We denote the columns of B by $B_1, \dots, B_n$. Now, we would like to compose a solution for order $2n$ out of two copies of each square A and B, whereby a column X_k in the new square $X = (x_{ij})_{1 \le i,j \le 2n}$ is forced to equal either $(A_l^T, B_l^T)^T$ or $(B_l^T, A_l^T)^T$ for some $1 \le l \le n$. The balancedness constraints are then expressed in terms of the column indices of A and B. Note how this streamliner reduces the size of our problem from n^2 variables to just n variables. See Fig.5 for an illustration.

Table 4 gives our computational results. We see that with streamlining based on composition, we find totally spatially balanced Latin squares of order 18.

A				B		
0	1	2		0*	1*	2*
1	2	0		1*	2*	0*
2	0	1		2*	0*	1*
A1	A2	A3		B1	B2	B3

C

0	1*	1	2	2*	0*
1	2*	2	0	0*	1*
2	0*	0	1	1*	2*
0*	1	1*	2*	2	0
1*	2	2*	0*	0	1
2*	0	0*	1*	1	2
A1	B2	A2	A3	B3	B1
B1	A2	B2	B3	A3	A1

Fig. 5. Illustration of streamlining by composition. A Latin square of order 6 is obtained using as building block a totally balanced Latin square of order 3 (A, left top corner). Latin square A is duplicated with its symbols relabeled, producing Latin square B. The composed Latin square C (order 6) is obtained by selecting entire columns of A and B, such that C is also a totally balanced Latin square of order 6 with the additional property that each of its columns is either of type $(A_i^T, B_i^T)^T$ or $(B_i^T, A_i^T)^T$($i = 1, 2,$ or 3). By using composition as a streamliner we obtain a much more compact representation for our streamlined problem, with only N variables. The domain of each variable is $\{A_1, A_2, A_3, B_1, B_2, B_3\}$, denoting the column that appears in the column i of the composed Latin square in the top part.

5 Conclusion

We have introduced a new framework for boosting constraint reasoning based on streamlining constraints. The idea consists in adding constraints that partition the set of solutions, thereby focusing the search and propagation mechanism on a small and structured part of the full combinatorial space. We used hard combinatorial design problems to demonstrate that this approach can be very effective. To develop good streamlining constraints, we studied regularities of solutions of small problem instances. A promising area for future work is to use statistical methods to try to discover useful solution regularities automatically.

Acknowledgment We would like to thank Bart Selman for interesting discussions and very helpful comments regarding the presentation of the streamlining idea.

References

1. R.K. Ahuja, T.L. Magnati, and J.B. Orlin. *Network Flows.* Prentice Hall, 1993.
2. K. R. Apt. The Rough Guide to Constraint Propagation. *5th International Conference on Principles and Practice of Constraint Programming (CP)*, LNCS 1713:1–23, 1999.
3. R.C. Bose, S.S. Shrikhande, and E.T. Parker. Further Results on the Construction of Mutually Orthogonal Latin Squares and the Falsity of Euler's Conjecture. *Canadian Journal of Mathematics*, 12:189–203, 1960.
4. P.Codognet and D.Diaz. An Efficient Library for Solving CSP with Local Search. *http://ws.ailab.sztaki.hu/Codognet_Diaz.pdf.*
5. C.J. Colbourn and J.H. Dinitz. (Eds.) *The CRC Handbook of Combinatorial Designs*, in CRC Press Series on Discrete Mathematics and its Applications, CRC Press, 1996.
6. T. Fahle, S. Schamberger, and M. Sellmann. Symmetry Breaking. *7th International Conference on Principles and Practice of Constraint Programming (CP)*, LNCS 2239:93–107, 2001.
7. F. Focacci and M. Milano. Global Cut Framework for Removing Symmetries. *7th International Conference on Principles and Practice of Constraint Programming (CP)*, LNCS 2239:77–92, 2001.
8. F. Focacci and P. Shaw. Pruning sub-optimal search branches using local search. CP-AI-OR, pp. 181–189, 2002.
9. C. Gomes, M. Sellmann, H. van Es, and C. van Es. The Challenge of Generating Spatially Balanced Scientific Experiment Designs. Poster CP-AI-OR, 2004. See also www.cs.cornell.edu/gomes/SBLS.htm.
10. ILOG Solver 5.1. User's manual:443–446, 2001.
11. J. Moran. *The Wonders of Magic Squares.* New York: Vintage, 1982.
12. J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. *12th National Conference on Artificial Intelligence*, AAAI, pp. 362–367, 1994.
13. J.K.Rowling. Harry Potter and the Sorcerer's Stone. *Scholastic*, 1998.
14. G. Tarry. Le problème des 36 officiers. *C.R.Assoc. France Av. Science*, 29(2):170–203, 1900.
15. W. Trump. How many magic squares are there? Found at *http://www.trump.de/-magic-squares/howmany.html*, 2003.
16. H. van Es and C. van Es. The spatial nature of randomization and its effects on outcome of field experiments. *Agron. J*, 85:420–428, 1993.
17. B. Hnich, B.M. Smith, and T. Walsh. Dual Modelling of Permutation and Injection Problems. *Journal of Artificial Intelligence Research*, 21:357–391, 2004
18. E.W. Weisstein. Magic Square. From Mathworld — A Wolfram Web Resource. *http://mathworld.worlfram.com/MagicSquare.html*, 1999.

Appendix A.

Proof (Thm. 1). We use standard permutation group theory for our proof. Denote with S_n the set of bijective functions $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$. We call the elements of S_n *permutations*. It is a well know fact that $(S_n, \circ)$ forms a group, whereby the operator $\circ : S_n \times S_n \to S_n$ denotes the standard composition of permutations, i.e. $(\pi_1 \circ \pi_2)(j) = \pi_1(\pi_2(j))$ for all $1 \le j \le n$ ($\pi_1 \circ \pi_2$ reads π_1 *after* π_2). For simplicity, when the notation is unambiguous, we leave out the $\circ$-operator sign and simply write $\pi_1\pi_2$ for the composition of two permutations. With $id \in S_n$ we denote the identity defined by $id(j) = j$ for all $1 \le j \le n$. Finally, for all $\pi \in S_n$, the unique function $\phi \in S_n$ with $\phi\pi = id = \pi\phi$ is denoted with π^{-1}. Note that $(S_n, \circ)$ is not abelian, i.e. in general the elements of the group do not commute.

Now, denote with A, B, C, D the subsequent squares that we get by applying permutations ρ_1, σ, and ρ_2. Further, let us set $s = \rho_1^{-1}(1)$ the index of the row that is permuted into the first position by ρ_1. Then, we can identify $\sigma = a_s^{-1}$, since by definition of σ it holds that $\sigma a_s = id$. Similar to σ, also ρ_2 is already determined by the choice of ρ_1. Since the first column of B, when read as a permutation, is equal to ρ_1^{-1}, it follows that $\rho_2 = \sigma\rho_1^{-1} = a_s^{-1}\rho_1^{-1}$, which implies $\rho_2^{-1} = \rho_1 a_s$. Now, let $1 \le k \le n$ denote some arbitrary row index. Then:

1. By definition of B, it holds $b_k = a_{\rho_1^{-1}(k)}$.
2. By definition of C, we have $c_k = \sigma b_k = a_s^{-1} a_{\rho_1^{-1}(k)}$.
3. For D, it holds that $d_k = c_{\rho_2^{-1}(k)} = c_{\rho_1 a_s(k)}$.

Equipped with these three facts, let us now prove the desired equivalence:

"$\Rightarrow$" By using our assumption that A is self-symmetric (i.e. A=D), we have:

$$a_k = d_k = c_{\rho_1 a_s(k)} = a_s^{-1} a_{\rho_1^{-1}\rho_1 a_s(k)} = a_s^{-1} a_{a_s(k)}. \tag{1}$$

Next, we exploit that A is diagonally symmetric (i.e. $a_{ij} = a_{ji}$, or, written as permutations, $a_i(j) = a_j(i)$):

$$a_s a_k(j) = a_{a_s(k)}(j) = a_j(a_s(k)) = a_j a_s(k) \quad \forall\, 1 \le j \le n. \tag{2}$$

And therefore

$$a_s a_j(k) = a_j a_s(k) \quad \forall\, 1 \le j \le n. \tag{3}$$

Note that the above must hold for all $1 \le s, j, k \le n$, since $A = D$ for arbitrary choices of ρ_1. Thus, we have

$$a_i a_j = a_j a_i \quad \forall\, 1 \le i < j \le n. \tag{4}$$

"$\Leftarrow$" Now let us assume a solution A to a diagonally symmetric TBLS has its first row and column fixed to id (this assumption is important since we will need to use our three facts again in the following). Let us assume further that A contains only commuting permutations. By exploiting the diagonal symmetry of A, we have:

$$a_s a_k(j) = a_s a_j(k) = a_j a_s(k) = a_j(a_s(k)) = a_{a_s(k)}(j) \quad \forall\, 1 \le k, j \le n. \tag{5}$$

And thus

$$a_k = a_s^{-1} a_{a_s(k)} = a_s^{-1} a_{\rho_1^{-1}\rho_1 a_s(k)} = c_{\rho_1 a_s(k)} = d_k \quad \forall\, 1 \le k \le n. \tag{6}$$

Consequently, A is a self-symmetric square.

□