Journal of Heuristics manuscript No.
(will be inserted by the editor)

Generating Highly Balanced Sudoku Problems as Hard Problems

Carlos Ansótegui · Ramón Béjar · Cèsar Fernández · Carla Gomes · Carles Mateu

Received: date / Accepted: date

Abstract Sudoku problems are some of the most known and enjoyed pastimes, with a never diminishing popularity, but, for the last few years those problems have gone from an entertainment to an interesting research area, a twofold interesting area, in fact. On the one side Sudoku problems, being a variant of Gerechte Designs and Latin Squares, are being actively used for experimental design, as in [8,44,39,9]. On the other hand, Sudoku problems, as simple as they seem, are really hard structured combinatorial search problems, and thanks to their characteristics and behavior, they can be used as benchmark problems for refining and testing solving algorithms and approaches. Also, thanks to their high inner structure, their study can contribute more than studies of random problems to our goal of solving real-world problems and applications and understanding problem characteristics that make them hard to solve. In this work we use two techniques for solving and modeling Sudoku problems, namely, Constraint Satisfaction Problem (CSP) and Satisfiability Problem (SAT) approaches. To this effect we define the Generalized Sudoku Problem (GSP), where regions can be of rectangular shape, problems can be of any order, and solution existence is not guaranteed. With respect to the worst-case complexity, we prove that GSP with block regions of m rows and n columns with $m \neq n$ is NP-complete. For studying the empirical hardness of GSP, we define a series of instance generators, that differ in the balancing level they guarantee between the constraints of the problem, by finely controlling how the holes are distributed in the cells of the GSP. Experimentally, we show that the more balanced are the constraints, the higher the complexity of solving the GSP instances, and that

Some of the results of this paper were presented in the *Twenty-First Conference on Artificial Intelligence* (AAAI-06) [2].

C. Ansótegui, R. Béjar, C. Fernández, C. Mateu
Departament d'Informàtica i Enginyeria Industrial
Universitat de Lleida
Jaume II, 69, E-25001 Lleida, Spain.
E-mail: {carlos,ramon,cesar,carlesm}@diei.udl.cat

C. Gomes
Department of Computer Science
Cornell University
Ithaca, NY 14853, USA.
E-mail: gomes@cs.cornell.edu

GSP is harder than the Quasigroup Completion Problem (QCP), a problem generalized by GSP. Finally, we provide a study of the correlation between backbone variables – variables with the same value in all the solutions of an instance– and hardness of GSP.

1 Introduction

Research on typical case complexity and worst-case complexity in structured problem domains – see [22,23,32,35,7] – or understanding how the structure of the problems affects the complexity to solve them – as in [43,46,26,27,25,13,20,48,47] – has always attracted, and continues to, a well deserved attention. As real world and industrial problems usually present some kind of structure, whereas random problems generally do not have inner structure, Sudoku puzzles have drawn the attention of researchers [2,42,36,16,19,45,33] because they show more structure than similar problems as the Latin Square Completion Problem –also known as Quasigroup Completion Problem (QCP)– while being, for similar problem sizes, harder to solve. Moreover, Sudoku has been recently used to help on experimental design [37].

In order to deepen our understanding of when structured problems are harder to solve, we define a generalized Sudoku problem, then we conduct a deep experimental study of typical case and worst case hardness of Sudoku problems, identifying the factors influencing hardness and proposing methods and algorithms to generate Sudoku problems of different hardness.

We propose a generalization of the Sudoku problem, that in turn, is a QCP with additional constraints, but with particular block region constraints that subsumes QCP. The proposed generalization extends Sudoku in several directions. First, as opposed to regular Sudokus, a Generalized Sudoku Problem (GSP) may adopt any arbitrary size. Second, block regions do not need to be square shaped[1].

GSP also has a good scaling behaviour by allowing rectangular block regions. When using only square block regions, after a 5x5 Sudoku (25 columns across and 25 rows high), the next problem available is a 6x6 Sudoku, that is, 36 rows by 36 columns. The jump in hardness between a 5x5 and a 6x6 is huge, thus leaving researchers with either easy problems or very hard problems. Rectangular shaped problems hardness, as can be seen in our empirical results, fits in between those square shaped sizes. For benchmarking purposes they provide a handful of size and options to choose from, offering more possibilities on hardness.

As this problem is an extension of QCP we have plenty of opportunity to introduce new balancing methods and, as has been previously shown [30,2], balancing has a significant impact on the hardness of the resulting problem instances. Due to the more complex underlying structure, as opposed to previously studied more random constraint satisfaction problems [3,4], introducing balance in all the groups of constraints is more challenging. Previous work has also considered how to produce harder instances of graph problems with methods that use balance in some way. For example, in [11] the authors give an algorithm that hides an independent set I on a graph and forces the number of edges from vertices outside of I to vertices inside I to be nearly equal, and they show that it is hard for heuristics to find such independent set. Finally, in [15] the authors show how to hide a $k-$coloring on a graph in such a way that it is hard to be

[1] Note that when the GSP size is a prime number (p), the only possible decomposition for blocks size is $1 \times p$, i.e., a QCP.

found by heuristics, thanks to balancing the degree of the vertices and edge distribution between pairs of vertex sets from an initial partition of the vertices in k sets.

In this work we also provide an empirical study on several facets of the GSP. One feature we measure experimentally is the effect of varying block region shape and size on problem hardness, showing that the more square shaped block regions are, the harder the problem is. Another studied facet is the effect that different degrees of balance in the hole pattern have on problem hardness, showing a step increase in hardness as hole patterns are more balanced. So, we define three different balancing patterns, called: singly, doubly, and fully balanced, introducing in each step one additional balancing rule, and we provide algorithms for generating problems with such patterns.

The article is structured as follows. Section 2 defines the Generalized Sudoku Problem. Section 3 presents a method to generate Generalized Sudokus, extending the Markov chain based algorithm for generation of Latin Squares [28] that was used for the generation of quasigroup problems with guaranteed solution [1]. Section 4 shows how to generate hole patterns to be applied to the already generated Sudokus. This section includes three methods for balancing hole patterns, with different degrees of balance. Section 5 studies the worst-case complexity of Sudoku problems for the case of non-square region shape. Section 6 is devoted to encode the GSP to SAT and CSP problems. Section 7 is a detailed empirical study of typical case complexity of Sudoku problems with any of the three balancing methods previously introduced, as well as, a study on the impact of balancing on hardness. Finally, before concluding, Section 8 studies the correlation of the backbone and the complexity of Sudoku instances.

2 Sudoku Problems

A Latin Square (LS), or Quasigroup, of order s, is an $s \times s$ matrix, with each of its s^2 cells filled with one of s symbols, such that no symbol is repeated in a row or column. A valid complete Generalized Sudoku (GS) of order s on s symbols, is a LS of order s with the additional restriction that each symbol occurs exactly once in each block region. A block region is a contiguous set of s pre-defined cells; block regions do not overlap, and there are exactly s block regions in a GS of order s. In the case of square block regions, each block region is an $\sqrt{s} \times \sqrt{s}$ matrix (s has to be a square number); in the case of rectangular block regions, each block region is an $m \times n$ matrix (m rows and n columns) with $m \times n = s$. Then, a GS with $m \times n$ block regions will have n region rows and m region columns (as an illustrating example, Fig. 1(c) shows a GS structure with $m = 2$ and $n = 3$).

We can trivially generate a GS of arbitrary order, with rectangular or square block regions, using the following method: let S denote the set of symbols of the GS,

$$S = (S_{i,j}^{k,l}), 0 \leq i, l \leq n-1, 0 \leq j, k \leq m-1, \tag{1}$$

where $S_{i,j}^{k,l}$ corresponds to the symbol located at i-th region row, j-th region column, and inside such a region it is on the k-th row and l-th column (see Fig. 1(a)). Then, these symbols can be represented as ordered pairs (see Fig. 1(b)) defined as

$$S_{i,j}^{k,l} = (s_1, s_0) = (k + j \,(\mathrm{mod}\ m), i + l \,(\mathrm{mod}\ n)). \tag{2}$$

Note that one can obtain the symbol value as $s_1 \cdot n + s_0$ (see Fig. 1(c)).

(a)

S_{00}^{00}	S_{00}^{01}	S_{00}^{02}	S_{01}^{00}	S_{01}^{01}	S_{01}^{02}
S_{00}^{10}	S_{00}^{11}	S_{00}^{12}	S_{01}^{10}	S_{01}^{11}	S_{01}^{12}
S_{10}^{00}	S_{10}^{01}	S_{10}^{02}	S_{11}^{00}	S_{11}^{01}	S_{11}^{02}
S_{10}^{10}	S_{10}^{11}	S_{10}^{12}	S_{11}^{10}	S_{11}^{11}	S_{11}^{12}
S_{20}^{00}	S_{20}^{01}	S_{20}^{02}	S_{21}^{00}	S_{21}^{01}	S_{21}^{02}
S_{20}^{10}	S_{20}^{11}	S_{20}^{12}	S_{21}^{10}	S_{21}^{11}	S_{21}^{12}

(b)

0,0	0,1	0,2	1,0	1,1	1,2
1,0	1,1	1,2	0,0	0,1	0,2
0,1	0,2	0,0	1,1	1,2	1,0
1,1	1,2	1,0	0,1	0,2	0,0
0,2	0,0	0,1	1,2	1,0	1,1
1,2	1,0	1,1	0,2	0,0	0,1

(c)

0	1	2	3	4	5
3	4	5	0	1	2
1	2	0	4	5	3
4	5	3	1	2	0
2	0	1	5	3	4
5	3	4	2	0	1

Fig. 1 GS construction example for $m = 2$ and $n = 3$.

These generalizations provide us with a range of interesting problems, from the Generalized Sudoku with square ($m = n$) or rectangular ($m \neq n$) block regions to the QCP [24] ($m = 1$).

We define the Generalized Sudoku Problem (GSP) as follows: given a partially filled Generalized Sudoku instance of order s, can we fill the remaining cells of the $s \times s$ matrix such that we obtain a valid complete Generalized Sudoku instance?

We define the Generalized Sudoku Problem (GSP) as follows: given a partially filled Generalized Sudoku instance of order s, can we fill the remaining cells of the $s \times s$ matrix such that we obtain a valid complete Generalized Sudoku instance?[2]

Also, to follow the research trend focusing on solvable problems, and coherently to the conventions for QWHP (Quasigroup With Holes Problem) [1], we define the Generalized Sudoku With Holes Problem (GSWHP) as those instances of GSP built from an existing GS where some cells have been emptied. This ensures that, at least, a solution exists. During this work, unless stated otherwise, we will deal with GSWHP, that is, problems with guaranteed solution.

3 Generating Complete Generalized Sudokus

From the previous definitions in section 2 is easily seen that in order to create a GSWHP instance, first we need to obtain a (valid and complete) GS. Then, to generate such GS, we follow the approach used in [28] of building a Markov chain whose set of states includes, as a subset, the set of complete Generalized Sudokus.

The procedure described in [28] allows, from an initial $s \times s$ Latin square, visit all the space of $s \times s$ Latin squares, following an uniform distribution, by defining some kind of random *moves* between the visited LS. These moves or *perturbations* defines a chain of states, where each state corresponds to a particular LS, and where the transition probability is state independent, so being a Markov chain.

One can think on different types of moves between LS. A type of moves could be row or column permutation, that always define moves to another LS (proper LS).

[2] Notice that we are mainly interested in problem benchmarking, so we do not consider the solution uniqueness of a well posed Sudoku problem. Nevertheless, we have oberved that the number of solutions ranges from one to many along the phase transition. In particular, the hardest point of the transition curve coincides with the point where the number of solutions abruptly increases from a few (or one) to many.

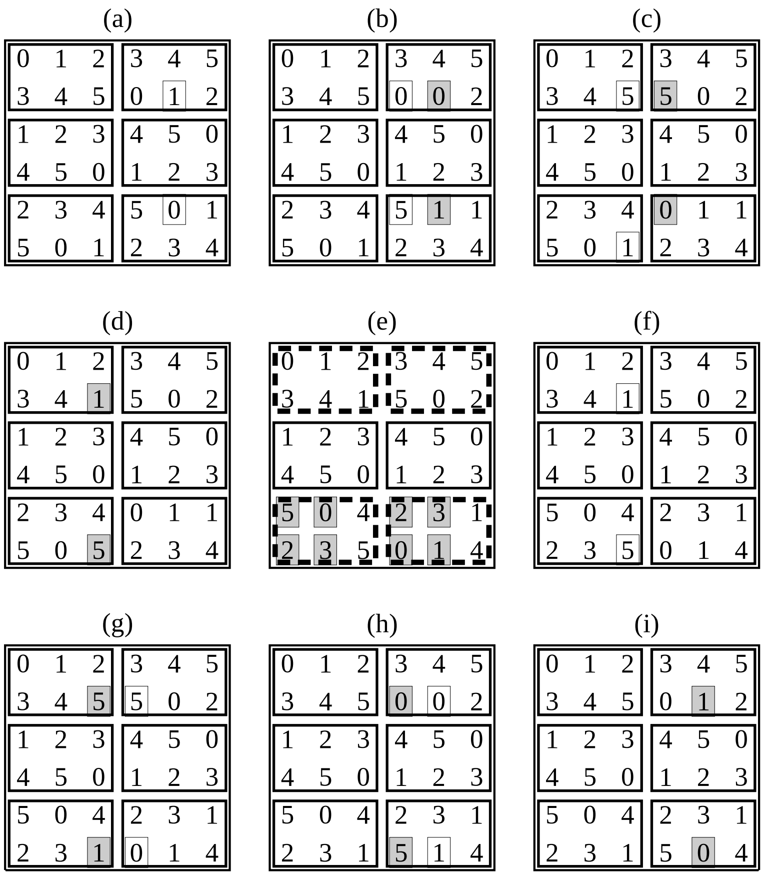

Fig. 2 GS generation example.

Another type of *finer* moves could be symbol swapping, that obviously could lead to something that is not a LS (improper LS). In order to ensure that all the LS space is visited, we will require of those improper LS as pivots toward a proper LS, because row and column permutations do not guarantee *per se* visiting all the space. In order to define more precisely these finer moves, an $s \times s \times s$ incidence binary matrix is defined, where each component indicates if a given symbol is at a given column and row (1) or not (0). Finer moves are defined as operations over the incidence matrix, and it is proved that there exists a maximum number of movements that lead from an improper LS to a proper one. If we use the Markov chain that considers only *proper* LS as states and using improper LS as pivots, and because any GS is also a LS of the same order, this chain obviously includes a subchain with all the possible GSs. So, for any pair of complete GSs, there exists a sequence of proper moves, of the type mentioned in Theorem 6 of [28], that transforms one into the other.

However, if we simply use the Markov chain by making proper moves uniformly at random, starting from an initial complete GS, most of the time we will reach LS that are not valid GSs. To cope with this problem, we select the move that minimizes the number of *violated* cells, i.e. cells with a symbol that appears at least twice in the same block region. To minimize those violated cells we proceed using the method defined in [28], that moves from one LS to another by choosing initially 4 random parameters $(r1, r2, t, i)$ (two rows, $r1$,$r2$, a symbol, t, and a number of iterations, i). If through our search for a GS we stop in a LS that is not a GS, instead of starting a new search by choosing those above mentioned parameters at random, we choose the initial rows at random but only among those that contain violated cells, and the same is applied to the initial symbol, which is one of the symbols that violated the GS condition in a block.

To escape local minima, if after a certain number of moves[3] we do not get a GS, we restart the move from the previous GS, and we perform the moves described above until we have traversed a certain number of valid GSs. Observe that this method does not necessarily generate a GS instance uniformly at random, because we do not always select the next move uniformly at random. However, as we will see in the experimental results, this method provides us with very hard computational instances of the GSWHP, once we punch holes in the appropriate manner. It is worth to mention that even if we were able to use a uniform generation method for GS, because we produce the final GSWHP instance by punching the holes in a second step independently of the first one, the distribution of GSWHP instances obtained will not be uniform. The reasons are the same as the ones discussed in [30] for explaining why a two-step generation process for satisfiable latin-square completion problems will be biased proportionally towards instances with more solutions. That is, the more solutions a GSWHP instance has, the more different GSs can be obtained in the first step with our Markov chain that can lead towards the same final GSWHP instance.

Fig. 2 shows an example of an execution of our Markov chain algorithm. From the canonical GS (a), we perform 2 moves ([a-e] and [f-i]) of the second method detailed in [28] until we obtain a valid GS (i). Each move is started by randomly selecting 2 rows, a symbol and the number of iterations. For the first move (a-e), the initial random values are $(1, 4, 1, 3)$. At the first iteration (a), symbol 1 points to the corresponding symbol to be switched by intersecting its column with the second row (symbol 0). Symbols to be switched are framed, becoming framed/shadowed once the change is produced. Previously switched symbol 0 is taken in the second iteration to designate its corresponding counterpart to be switched (symbol 5 in this case)(b). The same procedure applies up to the last iteration (c). At this point, the last selected symbol (5), is switched with the initial random selected symbol (1) at its corresponding row, giving (d). Once arrived at this point, if the resulting LS is improper (as it is in our example), switches between symbols of the second initial row and the row that contains the last selected symbol (5) are applied until getting a proper LS (e). Dashed blocks show which blocks violate the GS condition, so a second move will be needed. On the second move, the initial random values are $(1, 5, 1, 3)$. Note that in this case, rows 1 and 5 as well as symbol 1 are chosen at random from those that violate the GS condition, i.e. from dashed blocks on (e). The same procedure as before applies from (e) to (i), getting at this last step (i) a proper GS, so no additional shuffles are required unlike in (e).

[3] In our experiments, this number has been fixed to 20, because it works reasonably well with all the orders of GSs that we have tested.

The source code of the Markov chain algorithm, as well as the source code of all the hole pattern generators we present in the next section, can be downloaded from our web page at `http://ia.udl.cat/sudoku/`.

4 Balanced Hole Patterns

Once a valid GS is generated, to create a GSWHP we must punch holes to be filled. The simplest method to remove values from the GS is to choose which cells will be removed randomly. Actually, this was the method used in [33] to create Sudoku instances. This creates problems that will be, usually, easier to solve than if we choose these holes following a pattern. This is true especially when such a pattern is balanced, that is, that the number of holes in every row, column or block region is the same (or very similar). We will present here three methods to punch holes, each one progressively providing a more refined pattern, and we will see later, in the experimental results, that this increasingly refined balance heavily influences problem hardness.

4.1 Singly Balanced

First, we consider the balanced pattern used in [30] for QWHP instances, that we call here *singly balanced*. In a singly balanced pattern we have, when possible, the same number of holes in every row and column of the Sudoku. Given the total number of holes h, we can distribute $q = h/s$ holes in each row and column of the Sudoku using an algorithm for regular bipartite graph generation, based on a Markov Chain algorithm [29]. Observe that a hole pattern with q holes in every row and every column is equivalent to a $q-$regular bipartite graph $(R \cup C, E)$, with R the set of rows of the Sudoku and C its set of columns and $(r, c) \in E$ indicates that there is a hole in position (r, c) of the Sudoku. We move along the Markov chain, where every state is a pattern that satisfies that the number of holes in each row and column is the same, using a "switch" [29]. A switch is defined by two entries $(i, j), (i', j')$ of the GS instance, such that there is a hole in (i, j) and (i', j') but not in (i, j') and (i', j). If we change both holes to positions (i, j') and (i', j) the row and column hole number does not change. When $q = h/s$ is not an integer, we can still generate, with the same algorithm, an almost balanced pattern with a bipartite graph where the degree of the vertices is either $\lfloor q \rfloor$ or $\lfloor q \rfloor + 1$. In this case, we create the initial hole pattern putting $\lfloor q \rfloor$ holes in every row and column of the GS. Then we distribute the remaining $(h \ mod \ s)$ holes by randomly selecting $(h \ mod \ s)$ additional cells with no rows or columns in common.

4.2 Doubly Balanced

Because the distribution of holes between different blocks can make a difference in the difficulty of the problem, we propose a new method that ensures that the number of holes in every row, column, and block region will be the same. Our new *doubly balanced* method is based on the Markov chain of section 4.1, but now every state is a hole pattern that also satisfies that the number of holes is the same in all blocks. So, we use this Markov chain, but we restrict the moves to those moves that also maintain the number of holes present in each block, i.e., moves such that the blocks of (i, j) and

(i', j') are either in the same block row or the same block column, or both. Such moves always exist, even for a hole pattern with only one hole in each block. With this, we have the code detailed in Algorithm 1 for generating a hole pattern H with $q = h/s$ holes in each row, column and block, using a GS $S(i,j)$ (with symbols $\{1,\dots,s\}$) to create the initial pattern considering each symbol as a hole, and then performing t moves trough the Markov chain in order to sample from the set of possible doubly balanced hole patterns.

Algorithm 1: Algorithm to create Doubly Balanced hole patterns in a given GS

```
input  : $s, h, t$
output: a doubly balanced hole pattern $H$ of order $s$ with $h$ holes
$H = \{\ (i,j) \mid S(i,j) \in [1, \lfloor h/s \rfloor]\ \}$
for $1 \dots t$ do
    $T = \{\ switch((i,j),(i',j'))\ of\ H \mid \lfloor i/l \rfloor = \lfloor i'/l \rfloor \ \vee\ \lfloor j/n \rfloor = \lfloor j'/n \rfloor\ \}$
    pick a uniformly random $switch((i,j),(i',j'))$ from $T$
    $H = (H - \{(i,j),(i',j')\}) \ \cup\ \{(i,j'),(i',j)\}$
```

Observe that the Sudoku S considered to create the initial hole pattern can be any arbitrary Sudoku, and has no relation with the Sudoku to which we want to apply the hole pattern. Building such an arbitrary Sudoku can be done using Equations 1 and 2.

This code, obviously, only generates a perfect doubly balanced pattern when h/s is an integer. If this is not the case, we generate a hole pattern that is almost a doubly balanced one. That is, a hole pattern H that contains $h\ mod\ s$ rows, columns and blocks with $\lfloor h/s \rfloor + 1$ holes each and the remaining $s - (h\ mod\ s)$ rows, columns and blocks with $\lfloor h/s \rfloor$ holes. To get this hole pattern, we generate the initial pattern as in Algorithm 1, but also select a random subset M (with $|M| = (h\ mod\ s)$) of the positions (i,j) of the Sudoku with $S(i,j) = \lfloor h/s \rfloor + 1$ to select the positions of the additional $(h\ mod\ s)$ holes. So, the initial pattern will be:

$$\begin{aligned} H = \{\ (i,j) \mid S(i,j) \in [1, \lfloor h/s \rfloor]\ \} \cup \\ \{\ (i,j) \mid S(i,j) = \lfloor h/s \rfloor + 1 \wedge (i,j) \in M\} \end{aligned}$$

4.3 Fully Balanced

Our last balanced method, which we call *fully balanced*, is a generalization of the previous one. Here, we also force the number of holes in each row and column inside each block to be the same, when possible. So, this method produces a fully balanced hole pattern if the block regions are square ($m = n$), if the total number of holes h satisfies that $q_1 = h/s$ is an integer (the number of holes in each block, row and column of the Sudoku) and if $q_2 = q_1/n$ is also an integer (the number of holes in each row and column inside any block region). For that reason, in this model we restrict regions to be square (so $n = \sqrt{s}$), although we do not restrict the number of holes, so in general we will not always get a fully balanced pattern. If all these conditions are met, because what we indeed need in every block is a hole pattern that is singly balanced inside the block, the following simple code generates a fully balanced Sudoku:

Algorithm 2: Algorithm to create Fully Balanced hole patterns in a given GS

```
input  : $s, h$
output: a fully balanced hole pattern $H$ of order $s$ with $h$ holes
for $i \in 1 \ldots \sqrt{s}$ do
    for $j \in 1 \ldots \sqrt{s}$ do
        $H'$ := Singly balanced hole pattern of order $\sqrt{s}$ with $\lfloor h/s \rfloor$ holes
        set the hole pattern of block region $(i, j)$ in $H$ equal to $H'$
```

▨ 0	▨ 1	2
▨ 1	2	▨ 0
2	▨ 0	1

(a)

▨ 0	1	2	▨ 0	1	2	▨ 0	1	2
1	2	▨ 0	1	2	▨ 0	1	2	▨ 0
2	▨ 0	1	2	▨ 0	1	2	0	1
▨ 0	1	2	▨ 0	1	2	▨ 0	1	2
1	2	▨ 0	1	2	0	1	2	▨ 0
2	▨ 0	1	2	▨ 0	1	2	▨ 0	1
0	1	2	▨ 0	1	2	0	1	2
1	2	▨ 0	1	2	▨ 0	1	2	▨ 0
2	▨ 0	1	2	▨ 0	1	2	▨ 0	1

(b)

Fig. 3 Fully balanced GS hole poking example. (a) LS of order $\sqrt{s}$, every cell is associated with a region block on the resulting GS instance. (b) Resulting hole pattern where grayed cells will be holes in the GS instance.

In the case that $q_1 = h/s$ is not an integer, we have an additional set of $(h \ mod \ s)$ holes that we need to distribute as uniformly as possible between the s blocks. So, we will distribute one additional hole in every block region from a selected set of $(h \ mod \ s)$ block regions. To do it, consider a Latin Square $R(i, j)$ of order $\sqrt{s}$, with symbols $\{0, \ldots, \sqrt{s} - 1\}$, such that entry $R(i, j)$ is associated with the block region in block row i and block column j of our desired hole pattern. We use $R(i, j)$ to decide which blocks will contain one additional hole, trying to select the same number of block regions from every block row and every block column. Let $q = \lfloor((h \ mod \ s) \ / \ \sqrt{s})\rfloor$ and $r = ((h \ mod \ s) \ mod \ \sqrt{s})$. Then, the set of blocks (i, j) that will contain one additional hole will be:

$$\{(i, j) \mid R(i, j) < q\} \cup \{(i, j) \mid R(i, j) = q \ \wedge \ i < r\} \tag{3}$$

Observe that we try to distribute the remaining $(h \ mod \ s)$ holes giving almost the same number holes to every block row and block column (take into account that when $r > 0$, we cannot evenly distribute holes, some block rows and some block columns will necessarily have one more hole than the rest of block rows or block columns).

Finally, given a block (i, j) and the number of holes $h_{i,j}$ we have decided to distribute in the block, we basically use the Markov chain algorithm of section 4.1 in order to have the same number of holes in every subrow and subcolumn of the block, or almost the same number when $h_{i,j}/\sqrt{s}$ is not an integer.

This hole punching method can be easily seen in Fig. 3. In this example we poke 23 holes in a 9×9 sudoku puzzle, i.e. a 81 cell sudoku. As the number of holes is not a multiple of the number of block regions, we must punch 2 holes in every block region, and the remaining 5 holes will be punched using Equation 3. For this example, we have $h = 23$, $s = 9$, $q = 1$ and $r = 2$, thus Equation 3 results in

$$\{(i, j) \mid R(i, j) < 1\} \cup \{(i, j) \mid R(i, j) = 1 \ \wedge \ i < 2\}$$

this gives the following (i, j) values, referring to the following cells of LS in Fig. 3(a) that will have one additional hole: $\{(0,0),(1,2),(2,1)\} \cup \{(0,1),(1,0)\}$, shown in the figure as grayed cells. This leads to some unbalancing on some constraints, as are rows 3,5, and 7, that have less holes than the rest of the rows.

Once the number of holes to poke in every block region is set forth, we proceed as if the block region was a LS of $\sqrt{s}$ order and punch holes accordingly. The result can be seen on Fig. 3(b), where grayed cells represent holes, and, as it can be easily noticed, block regions with one additional hole (in the figure, block regions with 3 holes), correspond to grayed cells in 3(a).

5 Worst-case Complexity of GSP

It has been shown in [49], that GSP is NP-complete for the particular case of square regions (n columns and n rows). As the empirical complexity results of Section 7 will show, on average, GSP with rectangular regions is easier than with square regions (the complexity decreasing the larger the ratio n/m) the next natural question to answer was if NP-completeness still applied to the rectangular case. We show here that this is the case.

Theorem 1 *GSP with block regions with m rows, n columns and $n \neq m$ is NP-complete.*

Proof: The proof for this case is a generalization of the proof of [49], and shows a reduction from QCP (Quasigroup Completion Problem) to GSP (Generalized Sudoku Problem).

The following construction uses a GSP with $n > m$ but can be transformed to an isomorphic GSP with $m > n$, simply by a 90 degrees rotation.

Given an instance of the QCP of order m, the reduction follows by constructing an instance of GSP with region blocks with m rows and n columns such that the GSP instance has solution if and only if the QCP instance has solution. The symbols of the QCP instance are embedded into the first columns of the regions of the first region row.

Let L be the QCP instance and S the GSP instance, and their symbols be denoted by

$$L = (L_{i,j}),\ 0 \leq i,j \leq m-1;$$
$$S = (S^{k,l}_{i,j}),\ 0 \leq i,l \leq n-1, 0 \leq j,k \leq m-1,$$

where $S^{k,l}_{i,j}$ corresponds to the symbol of S located at k-th row and l-th column inside i-th region row, j-th region column.

Then, these symbols are ordered pairs defined as

$$S^{k,l}_{i,j} = \begin{cases} (L_{k,j}, 0), & \text{if } i = l = 0 \\ (k + j \pmod{m}, i + l \pmod{n}), & \text{otherwise.} \end{cases}$$

Under this construction, the original cells of the QCP instance are mapped to cells of the GSP instance with $i = l = 0$, that is, to cells of the first region column in the regions of the first region row.

It is straightforward to observe the GSP instances with rectangular block regions have solution if and only if the QCP instance has solution, because the same argument used in [49] can be used here to show that every solution to the QCP instance can be embedded into the GSP instance to give it a solution, and at the same time every solution to the GSP instance gives a solution to the QCP instance by looking at the symbols located at cells with $i = l = 0$. □

6 Encodings and Solvers

In order to experimentally measure the hardness of the GSP instances we have decided to solve them by their translation to the SAT and CSP domains. In the following we present the encoding strategies and the state-of-the-art solvers we have applied.

6.1 SAT and CSP

We consider the best performing encodings for the QCP analyzed in [6], and we extend them with the suitable representation of the additional alldiff constraints for the blocks in the GSP. Through the rest of the section we will consider a GSP instance on s symbols.

SAT

We focus on a particular type of SAT formulas, the CNF (clausal normal form) formulas.

Definition 1 Boolean variables are denoted with lower case letters $b_1, \ldots, b_n$ and can be assigned truth values 0 (or F) or 1 (or T). A *literal* is an expression of the form b_i or $\neg b_i$, where b_i is a Boolean variable. The *complement* of a literal l of the form b_i ($\neg b_i$), denoted by $\bar{l}$, is $\neg b_i$ (b_i). A *clause* c of length s is a disjunction of s literals, $l_1 \vee \ldots \vee l_s$. A *CNF formula* of length t is a conjunction of t clauses, $c_1 \wedge \ldots \wedge c_t$.

Definition 2 A *truth assignment* for a CNF formula is a mapping that assigns to every Boolean variable to value T or F. A truth assignment I *satisfies* a literal b_i ($\neg b_i$) iff $b_i = T$ ($b_i = F$), *satisfies* a clause C iff it satisfies at least one of the literals in C, and *satisfies* a CNF formula Γ iff it satisfies all clauses in Γ. The SAT problem consists of deciding whether there exists a truth assignment to the variables such that the formula becomes satisfied.

The GSP can be modelled as a set of permutations, i.e., every row, column and block region has to be a permutation of s symbols. Permutations can be modelled through the use of the alldiff constraint, which states that a set of variables have to be assigned to different values.

When dealing with SAT solvers, in the end we need to encode the problem into a SAT formula in clausal normal form (CNF), since this is the input that accepts a SAT solver. In order to show how we can encode an alldiff constraint into CNF, we recall two well known constraints: *at least one* and *at most one*, which will be the building blocks of the alldiff constraint, and we show how these constraints are translated into CNF.

Given a set of $B = \{b_1, \ldots, b_n\}$ of Boolean variables;

- the *at least one* constraint on B represents that at least one Boolean variable in B has to be true.

$$\sum_{1}^{n} b_i \geq 1,\ b_i \in B$$

The standard encoding into CNF produces the following clause:

$$b_1 \vee \ldots \vee b_n$$

- the *at most one* constraint on B represents that at most one Boolean variable in B can be true.

$$\sum_{1}^{n} b_i \leq 1,\ b_i \in B$$

The standard encoding into CNF produces $n * (n-1)/2$ clauses:

$$\neg b_i \vee \neg b_j,\ \forall i \neq j,\ 1 \leq i, j \leq n$$

Lets consider that we have a group of s cells that take one symbol from a set of size s. Then, if we want to model an alldiff constraint on the symbols assigned to these cells using the CNF encoding of the *at least one* and *at most one* constraints, we can do it in the following way:

- We use s Boolean variables per cell. Therefore, we have a set B of b_{ij} Boolean variables, $1 \leq i, j \leq s$, such that if the ith cell takes the jth symbol then the truth assignment evaluates b_{ij} to true, false otherwise.

1 Each symbol is assigned to *at least one* cell.

$$\forall j, 1 \leq j \leq s, \sum_{i=1}^{i=s} b_{ij} \geq 1,\ b_{ij} \in B$$

2 Each symbol is assigned to *at most one* cell.

$$\forall j, 1 \leq j \leq s, \sum_{i=1}^{i=s} b_{ij} \leq 1,\ b_{ij} \in B$$

The previous *at most one* and *at least one* can be now translated into CNF. Notice that we could have also added the following two constraints:

3 Each cell has to have *at least one* symbol.

$$\forall i, 1 \leq i \leq s, \sum_{j=1}^{j=s} b_{ij} \geq 1,\ b_{ij} \in B$$

4 Each cell has to have *at most one* symbol.

$$\forall i, 1 \leq i \leq s, \sum_{j=1}^{j=s} b_{ij} \leq 1,\ b_{ij} \in B$$

Any subset of the previous constraints that includes at least 1 or 2 and also includes any pair *at least one* and *at most one* models the alldiff constraint of our problem. The addition of redundant constraints usually helps to increase the propagation power of the filtering algorithms incorporated into the solvers.

Once we know how to encode an alldiff constraint into CNF we can introduce the first SAT encoding for the GSP. This encoding extends the (2D) encoding proposed in [30] for the QCP. The SAT encoding uses s Boolean variables per cell; each variable represents a symbol assigned to a cell, and the total number of variables is s^3.

The clauses corresponding to the 2D-GSP encoding represent the following constraints:

- Each cell c has to have *at least one* symbol (alo-cell$_c$).
- Each symbol is assigned to *at most one* cell in each row r (amo-row$_r$)
- Each symbol is assigned to *at most one* cell in each column cl (amo-column$_{cl}$).

The previous constraints encode the alldiff constraints on the rows and columns. Then, in order to fully encode a GSP, for each block region b we add the clauses that represent the following constraint:

- Each symbol is assigned to *at most one* cell in each block b (amo-block$_b$).

In [30] it was already shown that the 2D encoding for QCP is not really efficient in terms of the computation time needed by the SAT solver to find a truth assignment that satisfies the formula. A second encoding, the SAT 3-dimensional (3D) encoding was proposed. This encoding adds a set of redundant constraints that help to increase the propagation power of SAT solvers. In [2], it was shown that this propagation power is equivalent to the application of the Arc consistency filtering algorithm applied by CSP solvers. The clauses corresponding to our extension of the 3D encoding for the GSP (3D-GSP) represent the following constraints:

- Each cell c has to have *at least one* symbol (alo-cell$_c$).
- Each cell c has to have *at most one* symbol (amo-cell$_c$).
- Each symbol is assigned to *at least one* cell in each row r (alo-row$_r$);
- Each symbol is assigned to *at most one* cell in each row r (amo-row$_r$)
- Each symbol is assigned to *at least one* cell in each column r (alo-column$_{cl}$);
- Each symbol is assigned to *at most one* cell in each column cl (amo-column$_{cl}$).
- Each symbol is assigned to *at least one* cell in each block b (alo-block$_b$).
- Each symbol is assigned to *at most one* cell in each block b (amo-block$_b$).

As we can see, we have added for every *at least one* and *at most one* constraints their counterparts. The above SAT encoding was proposed independently in [2] and [36]. For the experimental investigation we considered the best SAT solvers in the previous experimental studies on the QCP [30,2]: Satz [34], zChaff [40], MiniSAT [18], and Siege[4].

CSP

We also wanted to evaluate the encodings of the GSP into the Constraint Satisfaction Problem (CSP) and to test the performance of the CSP solvers.

[4] Available at http://www.cs.sfu.ca/∼cl/software/siege/

Definition 3 A *constraint satisfaction problem (CSP)* instance, or *constraint network* is defined as a triple $\langle X, D, C\rangle$, where $X = \{x_1, \ldots, x_n\}$ is a set of variables, $D = \{d(x_1), \ldots, d(x_n)\}$ is a set of domains containing the values the variables may take, and $C = \{C_1, \ldots, C_p\}$ is a set of constraints. Each constraint $C_i = \langle S_i, R_i\rangle$ is defined as a relation R_i over a subset of variables $S_i = \{x_{i_1}, \ldots, x_{i_k}\}$, called the *constraint scope*. The relation R_i may be represented extensionally as a subset of the Cartesian product $d(x_{i_1}) \times \cdots \times d(x_{i_k})$. This is also called a *good* representation of the relation. A *nogood* representation presents the relation extensionally as the complement of the *good* representation into the previous Cartesian product. A *Binary CSP* is a CSP where all the constraints have a scope of size at most two.

Definition 4 An *assignment* for a CSP instance $\langle X, D, C\rangle$ is a mapping that assigns to each variable $x_i \in Y$, where $Y \subseteq X$, a value from $d(x_i)$. An assignment I satisfies a constraint $\langle\{x_{i_1}, \ldots, x_{i_k}\}, R_i\rangle \in C$, if $\langle I(x_{i_1}), \ldots, I(x_{i_k})\rangle \in R_i$. An assignment I with domain Y is consistent, if for every constraint $C_i \in C$ defined on variables $Y' \subseteq Y$, I restricted to Y' satisfies C_i.

The Constraint Satisfaction Problem (CSP) consists of, given a CSP instance, finding an assignment that satisfies the instance, if it exists, or showing that it is unsatisfiable.

The CSP encoding of the GSP extends the "bichannelling model" used in [17] for QCP, that uses dual variables linked with the primal variables in order to produce a higher propagation during search. We have two different sets of variables:

– A set of *primal variables* $X = \{x_{ij} \mid 1 \leq i \leq s, 1 \leq j \leq s\}$; the value of x_{ij} is the symbol assigned to the cell in the ith row and jth column.
– Two sets of *dual variables*: $R = \{r_{ik} \mid 1 \leq i \leq s, 1 \leq k \leq s\}$, where the value of r_{ik} is the column j where symbol k occurs in row i; and $C = \{c_{jk} \mid 1 \leq j \leq s, 0 \leq k \leq s\}$ where the value of c_{jk} represents the row i where symbol k occurs in column j.

The domain of all variables is $\{1, \ldots, s\}$, where these values represent respectively symbols, columns, and rows. Variables of different types are linked by channeling constraints:

– *Row channeling constraints* link the primal variables with the row dual variables: $x_{ij} = k \Leftrightarrow r_{ik} = j$.
– *Column channeling constraints* link the primal variables with the column dual variables: $x_{ij} = k \Leftrightarrow c_{jk} = i$.

The concrete CSP encoding we use in our experimental investigation is a Binary CSP represented with nogoods. Finally, for each block we add the nogoods representing the alldiff constraint over the set of primal variables involved in each block of the Sudoku problem.

For the experimental investigation we used a variation of the MAC solver by Régin and Bessière [10] (MAC-LAH) proposed in [6] that incorporates the technique of failed literals and the Satz's heuristic in terms of a CSP approach. The description of the heuristic is the following:

1. For each free CSP variable of domain size 2, we propagate each value of the domain in order to see if the domain can be reduced. As a result, the domain can remain as before, can be a singleton or can be empty. In the first case, we weight the variable using the balance function H of Satz's heuristics, where $w(x = i)$ is the

number of times that domains whose size is greater than 2 have been reduced after propagating the value i. In the second case, we fix the variable to the only value of its domain. In the third case, we have detected an inconsistency and we backtrack.

2. We select the first free CSP variable of domain size 2 with greatest value of function H.
3. If there is no candidate variable in step 2, we apply the default heuristic of the CSP solver (in our experiments, the minimum domain first heuristic).

As pointed out in [6] it is interesting to notice that each pair of constraints (alo, amo) into the SAT encoding, actually represent a CSP variable that takes the ith value, when the ith Boolean variable involved into the pair is true and the rest are false (the only allowed interpretations). For example, the pair (alo-cell$_c$, amo-cell$_c$) models the primal CSP variable for the cell c. The channeling constraints considered in the CSP encoding are represented in the SAT encoding by the pairs (alo,amo) used for the rows and columns.

In particular, every pair (alo,amo) into the SAT encoding corresponds to a cardinality constraint of the form $\sum_n b_i = 1$, where the b_i's represent the Boolean variables involved into the constraint. It is worth mentioning that there are solvers that can directly manage this kind of constraints; many-valued SAT solvers as described in [5], Satisfiability Modulo Theories (SMT) solvers, or Pseudo-Boolean solvers. However, in the current work we have not conducted any experimental investigation in this sense.

6.2 Higher arity CSP encoding

With higher arity constraints solvers, although we can use both approaches (so called primal and dual or 3D encoding) to solve the problem, we have used only 3D encoding. The higher arity constraint solver used is a state of the art solver, named Minion[21]. Problem variables are defined as multivalued variables (in our case, using Minion scalar variable type) and we have two sets, primal and dual variables, that correspond to the same sets as defined in CSP encoding section (6.1).

The encoding of GSP using the primal encoding with higher arity constraints uses the following global constraints, defined over all variables of the row, column or region block, as corresponds:

– One *alldiff* constraint for every row
– One *alldiff* constraint for every column
– One *alldiff* constraint for every region block

Channeling constraints (to ensure that values are consistent between primal variables and their corresponding dual variables), have been built, due to Minion language limitations, using two reification (implication) constraints. And so, we have:

– A set of row channeling constraints, linking problem variables (primal variables) with row variables.
– A set of column channeling constraints, linking problem variables (primal variables) with column variables.

7 Typical case complexity

In this section, we experimentally analyze the complexity behavior of GSWHP problems depending on the employed method for punching holes. We first compare the complexity patterns among singly balanced GSWHP problems for distinct block region shapes, looking as well to the performance of different solving algorithms at the hardest zone of the complexity pattern. Finally, we show how this complexity is raised when more balanced puncturing methods for generating holes (doubly and fully) are used.

7.1 Singly Balanced

We consider the complexity of solving GSWHP instances generated with the Singly Balanced method. Our first set of results shows the complexity of Solving GSWHP instances with different block factor forms, comparing it with the complexity of solving QWHP instances. Fig. 4 shows the results for GSWHP with blocks 15×2, 10×3 and 6×5 (size 30) and QWHP also of size 30 (the encoding used for QWHP is 3D). We employ 100 instances per point and MiniSAT solver with 5,000 seconds cutoff. Complexity patterns are quite similar. First, all of them show the characteristic easy-hard-easy behavior depending on the number of holes and associated to a phase transition effect that is mentioned in the next section. This typical behavior for Sudoku problems was already cited in [2,33]. Second, it's worth to note that the closer to a square is the block region shape, the greater is its peak complexity. So, for the same size, the easiest instances are from QWHP and the hardest ones those from GSWHP with almost square blocks[5]. Observe that the difference between QWHP and GSWHP with square blocks is about three orders of magnitude for this size. Our conjecture is the following; for a GSWHP instance with blocks $n \times m = s$, for a given s, each cell has the following constraints:

$s - 1$ constraints with cells in the same block
$(m - 1) \cdot n$ constraints with the remainder cells in the same row
$(n - 1) \cdot m$ constraints with the remainder cells in the same column

giving a total number of constraints per cell of

$$3s - 1 - n - \frac{s}{n}.$$

Deriving respect to n and setting equal to zero, we obtain a maximum for $n = \sqrt{s}$, i.e. squared blocks. Being GSWHP a satisfiable problem, independently of the puncturing method, squared blocks lead to more constrained problems and as experimentation proves, harder to solve.

The shape of the blocks can also make a difference with respect to the effect on the block constraints when the heuristic of the algorithm is evaluating the next assignment to perform. Consider a Sudoku with block regions with m rows and n columns. If the heuristic of the solver is evaluating the effect of coloring a cell (i, j) with color c, observe that an inmmediate consequence of such assignment will be that color c will be removed from the domain of all the cells in column j and row i (and from all the cells of its block region). But for block regions different of the block region of the cell

[5] Since 30 is not a perfect square, we cannot have perfectly square blocks.

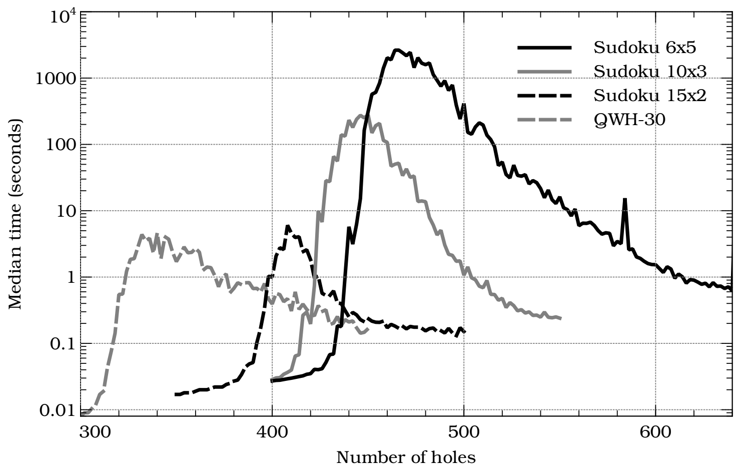

Fig. 4 Empirical complexity patterns for singly balanced GSWHP instances with different block regions form factor and same size

(i, j) the impact will not be uniform. For a block region that intersects with column j, m cells from the block region will become more constrained, so that from the total $mxn = s$ cells of the block region the color c will be only possible to be assigned to at most $m \times n - m$ cells. Analogously, for a block region that intersects with row i, n cells from the block region will become more constrained, so in this case from the total $m \times n$ cells of the block region the color c will be only possible to be assigned to at most $m \times n - n$ cells. So, observe that these two quantities will be different when $m \neq n$. For example, if $n >> m$, the block regions that interset row i will lose a higher percentatge number of possible cells to receive color c than the block regions that intersect column j. That means that for rectangular block regions, the assignments that the heuristic chooses can lead towards overconstrained subproblems more quickly than with square block regions, so partial assignments that do not lead to complete solutions may be pruned earlier.

We observe the same qualitative behavior when using different SAT algorithms. The main difference is the magnitude of the peak of the complexity curve. Fig. 5 shows a plot with the performance of different algorithms in the critically constrained area for different GSWHP problems. The plot shows, for different sizes and different algorithms, the percentage of solved instances from a test-set of 200 instances, when using a cutoff of 10^4 seconds. For small sizes, all algorithms solve almost all the instances. But as the hardness increases, the solver MiniSAT clearly outperforms the other solvers.

7.2 Doubly and Fully Balanced

Next, we consider the doubly and fully balanced method for punching holes. When using doubly balanced, the typical hardness of the GSWHP instances seems to be very similar to the singly balanced method for small sizes, however, as we increase the size of the instances, bigger differences appear in computational hardness. This

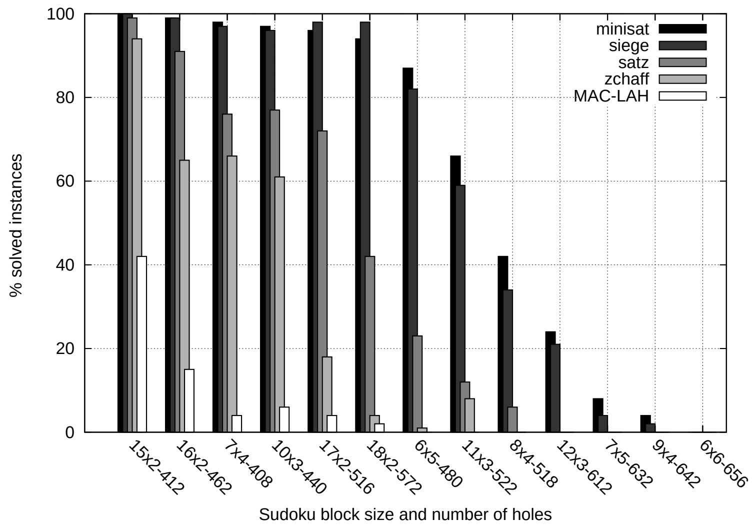

Fig. 5 Empirical complexity patterns for singly balanced GSWHP instances with different block shapes. 10^4 seconds time out

Table 1 Comparison of percentage of solved GSWHP instances generated with three methods (singly, doubly, and fully balanced) for putting holes, for instances at the peak of hardness. 500 instances per row with a 5,000 seconds time out

			Satz			**Minisat**	
block	holes	singly	doubly	fully	singly	doubly	fully
5×5	344	98.8	98.8	77.0	100.0	100.0	95.6
7×4	414	71.6	67.4	n/a	98.8	97.0	n/a
10×3	446	58.4	54.0	n/a	95.8	93.4	n/a
6×5	480	18.6	11.6	n/a	81.4	71.4	n/a
16×2	462	85.8	79.2	n/a	98.4	96.8	n/a
8×4	518	2.6	1.8	n/a	37.4	31.6	n/a
17×2	504	48.6	37.8	n/a	85.2	76.0	n/a
18×2	572	33.8	24.9	n/a	91.6	86.0	n/a
6×6	686	0.0	0.0	0.0	3.2	1.6	0.0

is probably due to the fact that the singly balanced method tends to distribute the holes uniformly between blocks in such a way that the difference with respect to the doubly balanced method is not significant for small instances. This can be quantified by looking at the percentage of solved instances from a test-set with 500 instances, for both methods, when working with a cutoff of 5,000 seconds. Table 1 shows these values. Solved ratios are almost the same for 5×5 Sudokus, but as the order increases, so does the relative difference between doubly and singly balance methods in terms of ratio of solved instances as well as in terms of time to solve them as reflected in Table 2. Our doubly balanced method, then, gives harder instances than those produced by balanced QWH, guaranteeing satisfiabilitiy as well, and therefore constituting a good benchmark for the evaluation of local and systematic search methods.

Besides, when applicable (squared blocks), the fully balanced method generates even harder instances. Due to the hardness of the problems generated by this method,

Table 2 Median time (in seconds) for GSWHP problems from Table 1 where the percentage of solved instances is greater than 50%

		Satz			**Minisat**		
block	holes	singly	doubly	fully	singly	doubly	fully
5×5	344	13	12	733	3	3	129
7×4	414	1,236	1,480	n/a	55	65	n/a
10×3	446	2,646	4,227	n/a	91	117	n/a
6×5	480	–	–	n/a	663	1,481	n/a
16×2	462	150	465	n/a	15	40	n/a
8×4	518	–	–	n/a	2,529	5,027	n/a
17×2	504	–	–	n/a	220	497	n/a
18×2	572	–	–	n/a	103	213	n/a

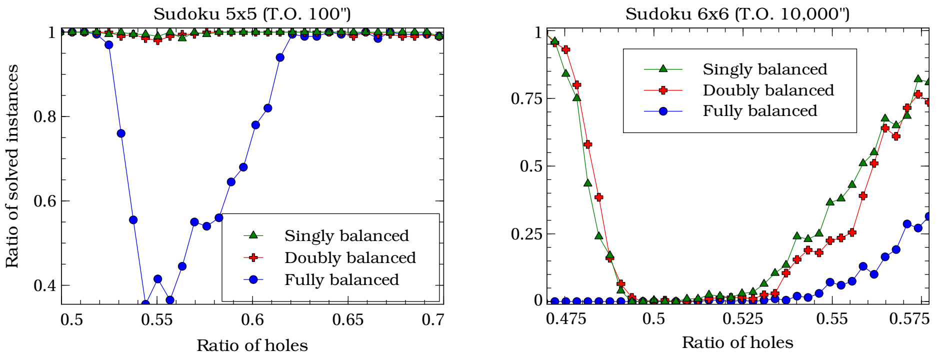

Fig. 6 Comparison of the hardness of instances generated using single, doubly balanced and fully balanced methods of punching holes. Plot shows the rate of solved instances (using Minisat over 200 generated instances) for a specified time out in seconds as a function of the number of holes

we are able to compute results only for small GSWHP of size 5×5, but even in this sort of problems, fully balanced method is able to produce much harder instances (in time) than singly and doubly balanced. In order to depict such differences, Fig. 6 compares the hardness of the three balanced methods for GSWHP sizes of 5×5 and 6×6, along a broad range of holes, above and below the peak of hardness.

We also conducted experimentation using state of the art solvers implementing high arity constraints such as Minion[21]. Results in Table 3 show the behavior of a MAC [10] as compared to Minion. In this case the instances were created with the doubly balanced hole pattern. The results show that, even for rather small puzzle sizes, Minion is unable to solve a significant number of instances.

8 Backbone

In this section, results about the correlation between the backbone of the GSWH instances and computational hardness are discussed. A variable pertains to the backbone of the instance if it adopts the same value in all the solutions of the instance. The backbone fraction is the number of backbone variables versus the total number of variables

Table 3 Comparison of solved GSWHP instances with Minion [21] and MAC [10] solvers (dual encoding used, time out of 10,000 seconds, median time in seconds)

		MAC		**Minion**	
block	holes	% solved	median	% solved	median
4×4	148	100	0.19	100	7.98
5×4	222	100	1.7	11	$> 10,000$
5×5	346	42	$> 10,000$	0	$> 10,000$

ratio. As GSWHP instances have always at least one solution, the backbone is well defined for them. Observe that instances with a unique solution will have all of their variables in the backbone, whereas instances with many different solutions will have an smaller backbone fraction. So, the backbone fraction can be used to quantify how much different are all the solutions of an instance.

It has been previously shown for other NP problems that there is a correlation between problem hardness and backbone fraction such that the hardest problems seem to be concentrated around the point where a sudden change on the backbone fraction occurs, like for example SAT [38] and QWHP [1]. So, given that the usual 3x3 Sudoku puzzles have the additional property of having unique solution, it is natural to question whether for Generalized Sudoku Problems, like GSWHP, the property of solution uniqueness is really fundamental for problem hardness.

It is known that computing the exact backbone is an intractable problem [31], so we consider only an approximation of the full backbone. In particular, we use the look-ahead backbone provided by the solver Satz when solving the SAT encoding of GSWHP instances. This is the set of variables that Satz discovers to have a unique value by checking all possible individual assignments (for every variable check the effect of assigning it either the value true or false) with unit propagation[6] and fixing every discovered backbone variable, until no new backbone variable is discovered.

In our approximation of the backbone, we consider the fraction of look-ahead backbone variables discovered by Satz over the number of variables of the (satisfiable) SAT encoded instance. Our satisfiable SAT instances are obtained after preprocessing our GSWHP instances for discarding the cells that are discovered to have a unique possible value due to the initial partial assignment of the cells of the Sudoku and the propagation of the Sudoku constraints.

Fig. 7 shows the evolution of the backbone together with the complexity (median time) to solve the instances, for different region shapes, but normalized so that the value at the peak of hardness coincides with the maximum number of look-ahead backbone variables. We observe that this approximated backbone fraction starts to increase until it reaches a point where it decreases abruptly and then it again starts to increase, but this time more slowly. The point where it reaches the minimum value is around the value where the hardness starts to increase towards its peak. So, this point of "sudden" decrease in the backbone fraction can be used as a sign for the beginning of the hard region of the problem. It is remarkable that even though the backbone is hard to approximate, in this problem this approximated backbone provides valuable information. Observe that the sudden decrease in the backbone fraction actually indicates the point

[6] Unit propagation is a linear-time constraint propagation rule that simplifies formulas with unit literals by discarding clauses that contain a unit literal as satisfied clauses, and eliminating complementary literals from the rest of clauses because they cannot be satisfied.

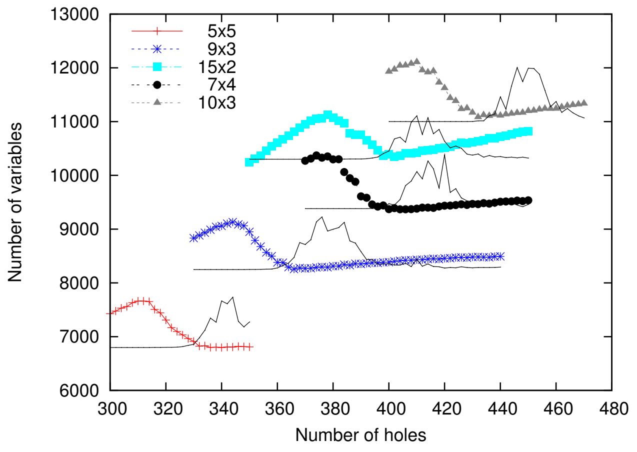

Fig. 7 Look-ahead backbone and normalized complexity patterns for Sudokus with different blocks

where problem instances change from having one solution to many. This indicates that Sudoku problems with few solutions may be harder than Sudoku problems with just one solution.

We have obtained an approximated location of this minimum point through a doubly exponential regression model (see Fig. 8), using the location of this minimum value for every possible region form ($m \times n$), from size 26 to 49. The model obtained is:

$$holes = e^{0.537} \cdot m^{1.57} \cdot n^{1.72}$$

The coefficient of regression (R^2) is 0.989, thus indicating that the model is quite good. We have also obtained an analogous regression model, but for the location of the hardness peak. For this model we used data obtained experimentally with Satz, but only for problems with sizes ranging from 26 to 30. The model obtained is:

$$holes = e^{-0.217} \cdot m^{1.8} \cdot n^{1.97}$$

Again, we obtain a high value for R ($R^2 = 0.997$). Observe that for the hardest problems ($m = n$), the relative difference between these two points is only $O(m^{1.14})$, much smaller than the whole range of possible holes (m^4). So, as m increases, the width of the hard part of the phase transition seems to decrease, in a normalized scale.

9 Conclusions

As hardness characterization of random generated problems has been deeply studied in the past, this paper focuses on getting a better understanding of where the harder problem instances are for a class of more structured problems. Given the importance of balance in random problems, it was natural to think that balancing on more structured problems also should be important. However it is not easy to study the effect of balance in problems with a rich and complex structure, as the procedures needed to enforce

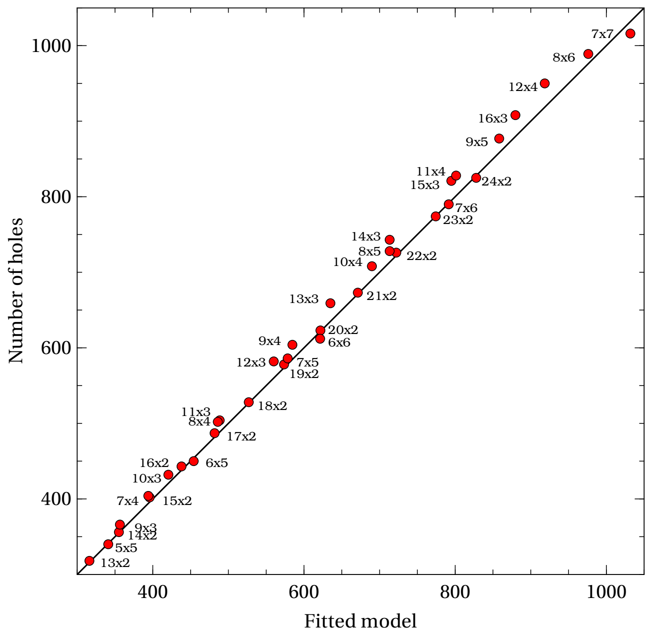

Fig. 8 Regression model for the hardness peak location on Sudokus from size 26 to 49

some level of balance become more and more complex as the the problem constraints gain in complexity.

So, on top of a widely known problem, Sudoku, we formalize a generalization, Generalized Sudoku Problem (GSP), relaxing one of the intrinsic restrictions of the Sudoku problem (block region square shape), to have a problem that still keeps enough structural properties but that now has a good margin to adjust its balancing properties by simply varying block region squareness.

Our results show that as more balance is introduced between the constraints of the problem, the hardness of the typical instances increases. For doing such a study, we have provided both an algorithm to generate initial complete Sudokus and algorithms to force different levels of balance between the constraints of the problem, by controlling the distribution of holes, such that almost all the constraints of a class are indistinguishable from the point of view of how much they constraint the problem.

Finally, as future work, we plan to study the effect of balance on more complex structured problems, like for example combinatorial auctions [41] and mixed multi-unit combinatorial auctions [12], that are of direct interest in real applications (electronic commerce), but they still do not have too many kinds of constraints such that defining ways of controlling the balance between the constraints becomes inaccessible. As possible ways to further extend our understanding of hard instances of structured problems, we believe that connections between our notion of balanced problems and the concepts of solution symmetry and constraint symmetry [14] may hold, given that it is natural to think that higher levels of balance between the constraints of a problem, will increase the possibilities of having constraint symmetries.

Acknowledgements This research was partially supported by the Spanish CICYT Projects MULOG2 (TIN2007-68005-C04-02) and ARINF (TIN2009-14704-C03-01) and TASSAT (TIN2010-20967-C04-03).

References

1. Achlioptas, D., Gomes, C., Kautz, H., Selman, B.: Generating satisfiable problem instances. In: Proc. of National Conference on Artificial Intelligence, (AAAI'00), pp. 193–200 (2000)
2. Ansótegui, C., Béjar, R., Fernández, C., Gomes, C.P., Mateu, C.: The impact of balancing on problem hardness in a highly structured domain. In: Twenty-First National Conference on Artificial Intelligence (AAAI'06) (2006)
3. Ansótegui, C., Béjar, R., Fernández, C., Mateu, C.: On Balanced CSPs with High Treewidth. In: Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, (AAAI'07), pp. 161–166 (2007)
4. Ansótegui, C., Béjar, R., Fernández, C., Mateu, C.: From High Girth Graphs to Hard Instances. In: Proceedings of the 14th international conference on Principles and Practice of Constraint Programming, (CP'08), pp. 298–312. Springer-Verlag, Berlin, Heidelberg (2008). DOI http://dx.doi.org/10.1007/978-3-540-85958-1_20
5. Ansótegui, C., Larrubia, J., Li, C.M., Manyà, F.: Exploiting multivalued knowledge in variable selection heuristics for SAT solvers. Annals of Mathematics and Artificial Intelligence **49**(1–4), 191–205 (2007)
6. Ansótegui, C., del Val, A., Dotú, I., Fernández, C., Manyà, F.: Modelling choices in quasigroup completion: SAT vs CSP. In: Proc. of National Conference on Artificial Intelligence, (AAAI-04) (2004)
7. Argelich, J., Lynce, I.: CNF instances from the software package installation problem. In: In Proceedings of 15th RCRA workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion, (RCRA'08) (2008)
8. Bailey, R.A., Cameron, P.J., Connelly, R.: Sudoku, gerechte designs, resolutions, affine space, spreads, reguli, and hamming codes. American Mathematical Monthly **115**, 383–404 (2008)
9. Bailey, R.A., Kunert, J., Martin, R.J.: Some comments on gerechte designs. Journal of Agronomy and Crop Science **165**, 121–130 (1990)
10. Bessière, C., Régin, J.C.: MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In: Proceedings of the Second International Conference on Principles and Practice of Constraint Programming, (CP'96), pp. 61–75 (1996)
11. Brockington, M., Culberson, J..: Camouflaging independent sets in quasi-random graphs. In: Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, pp. 75–88. AMS (1996)
12. Cerquides, J., Endriss, U., Giovannucci, A., Rodríguez-Aguilar, J.A.: Bidding languages and winner determination for mixed multi-unit combinatorial auctions. In: Proceedings of the 20th International Joint Conference on Artificial Intelligence, (IJCAI'07), pp. 1221–1226 (2007)
13. Chen, H., Gomes, C.P., Selman, B.: Formal models of heavy-tailed behavior in combinatorial search. In: Proceedings of the International Conference on Principles and Practice of Constraint Programming, (CP'01), pp. 408–421 (2001)
14. Cohen, D.A., Jeavons, P., Jefferson, C., Petrie, K.E., Smith, B.M.: Symmetry definitions for constraint satisfaction problems. Constraints **11**(2–3), 115–137 (2006)
15. Culberson, J., Luo, F.: Exploring the k-colorable landscape with iterated greedy. In: Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, pp. 245–284. AMS (1996)
16. Delahaye, J.P.: The science behind sudoku. Scientific American (June 2006), 80–87 (2006)
17. Dotú, I., del Val, A., Cebrián, M.: Redundant modeling for the quasigroup completion problem. In: Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming, (CP'03), pp. 288–302 (2003)
18. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Proceedings of the International Conference on Theory and Applications of Satisfiability Testing, (SAT'2003) (2003)
19. Felgenhauer, B., Jarvis, F.: Mathematics of sudoku i. Mathematical Spectrum **39**, 15–22 (2006)

20. Gao, Y., Culberson, J.: Consistency and random constraint satisfaction models. Journal of Artificial Intelligence Research **28**, 517–557 (2007)
21. Gent, I.P., Jefferson, C., Miguel, I.: Minion: A fast scalable constraint solver. In: 17th European Conference on Artificial Intelligence, (ECAI 2006), pp. 98–102 (2006)
22. Gent, I.P., Walsh, T.: The tsp phase transition. Artificial Intelligence **88**(1–2), 349–358 (1996)
23. Gomes, C., Selman, B.: Problem structure in the presence of perturbations. In: Proceedings of the Fourteenth National Conference on Artificial Intelligence, (AAAI'97), pp. 221–227. AAAI Press, New Providence, RI (1997)
24. Gomes, C.P., Selman, B., Crato, N.: Heavy-tailed distributions in combinatorial search. In: Proceedings of the Third International Conference of Constraint Programming, (CP'97). Springer-Verlag, Linz, Austria. (1997)
25. Gomes, C.P., Selman, B., Crato, N., Kautz, H.: Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. Journal of Automated Reasoning **24**(1–2), 67–100 (2000)
26. Hoffmann, J., Gomes, C.P., Selman, B.: Structure and problem hardness: Goal asymmetry and DPLL proofs in SAT-based planning. In: Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling, (ICAPS'06), pp. 284–293 (2006)
27. Hogg, T.: Exploiting the deep structure of constraint satisfaction problems with quantum computers. In: Proceedings of the National Conference on Artificial Intelligence, (AAAI'97), pp. 334–339 (1997)
28. Jacobson, M.T., Matthews, P.: Generating uniformly distributed random latin squares. Journal of Combinatorial Design **4**, 405–437 (1996)
29. Kannan, R., Tetali, P., Vempala, S.: Simple Markov-chain algorithms for generating bipartite graphs and tournaments. In: Proc. of the eighth annual ACM-SIAM Symposium on Discrete Algorithms, pp. 193–200 (1997)
30. Kautz, H., Ruan, Y., Achlioptas, D., Gomes, C., Selman, B., , Stickel, M.: Balance and filtering in structured satisfiable problems. In: Proc. of Iinternational Conference on Artificial Intelligence, (JCAI'01), pp. 193–200 (2001)
31. Kilby, P., Slaney, J., Thiebaux, S., Walsh, T.: Backbones and backdoors in satisfiability. In: Proc. of National Conference on Artificial Intelligence, (AAAI'05), pp. 193–200 (2005)
32. Kilby, P., Slaney, J.K., Walsh, T.: The backbone of the travelling salesperson. In: Proceedings of the International Joint Conference on Artificial Intelligence, (IJCAI'05), pp. 175–180 (2005)
33. Lewis, R.: Metaheuristics can solve sudoku puzzles. Journal of Heuristics **13**(4), 387–401 (2007)
34. Li, C.M., Anbulagan: Look-ahead versus look-back for satisfiability problems. In: Proceedings of the International Conference on Principles and Practice of Constraint Programming, (CP'97), pp. 341–355 (1997)
35. Lynce, I., Marques-Silva, J.: Haplotype inference with boolean satisfiability. International Journal on Artificial Intelligence Tools **17**(2), 355–387 (2008)
36. Lynce, I., Ouaknine, J.: Sudoku as a SAT problem. In: Proc. of Ninth International Symposium on Artificial Intelligence and Mathematics, (ISAIM-06) (2006)
37. MO, H.D., XU, R.G.: Sudoku square – a new design in field. Acta Agronomica Sinica **34**(9), 1489 – 1493 (2008)
38. Monasson, R., Zecchina, R., Kirkpatrick, S., Selman, B., Troyansky, L.: 2+p-sat: Relation of typical-case complexity to the nature of the phase transition. Random Structures and Algorithms **15**(3-4), 414–435 (1999)
39. Morgan, J.P.: Latin Squares and related experimental designs, chap. To be published. Wiley
40. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient sat solver. In: Proceedings of 39th Design Automation Conference (2001)
41. Sandholm, T., Suri, S.: Improved algorithms for optimal winner determination in combinatorial auctions and generalizations. In: Proc. of Seventeenth National Conference on Artificial Intelligence, (AAAI'00), pp. 90–97 (2000)
42. Simonis, H.: Sudoku as a constraint problem. In: Proc. of Fourth International Workshop on Modelling and Reformulating Constraint Satisfaction Problems (in CP-2005), pp. 13–27 (2005)
43. Smith, B.M., Grant, S.A.: Sparse constraint graphs and exceptionally hard problems. In: Proc. of the International Joint Conference on Artificial Intelligence, (IJCAI'95), pp. 646–654 (1995)

44. Vaughan, E.R.: The complexity of constructing gerechte designs. The electronic journal of combinatorics **16** (2009)
45. Weber, T.: A SAT-based Sudoku solver. In: G. Sutcliffe, A. Voronkov (eds.) LPAR-12, The 12th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, Short Paper Proceedings, pp. 11–15 (2005)
46. Williams, C.P., Hogg, T.: Exploiting the deep structure of constraint problems. Artificial Intelligence **70**, 73–117 (1994)
47. Xu, K., Boussemart, F., Hemery, F., Lecoutre, C.: Random constraint satisfaction: Easy generation of hard (satisfiable) instances. Artificial Intelligence **171**(8-9), 514–534 (2007)
48. Xu, K., Li, W.: Exact phase transitions in random constraint satisfaction problems. Journal of Artificial Intelligence Research **12**, 93–103 (2000)
49. Yato, T., Seta, T.: Complexity and completness of finding another solution and its application to puzzles. In: Proc. of National Meeting of the Information Processing Society of Japan (IPSJ) (2002)