# Patterns of Hard Sudokus

Kris Oosting
Technical Artificial Intelligence, Vrije Universiteit Amsterdam
kris.oosting@dfsxpertsys.com

2011

**Abstract**

Sudoku is a well-known puzzle that has been a favorite topic for automated reasoning in artificial intelligence. This paper addresses the questions if there is a direct relation between classes of Sudoku problems and the computational hardness of their related SAT problem, and what the patterns of givens and solutions are of hard Sudokus. The steps taken were the encoding of Sudoku puzzles into conjunctive normal form (CNF) using minimal, efficient, and extended encoding, solving them using a satisfiability (SAT) solver, and subsequently creating the patterns of givens and solutions for 49,151 hard Sudoku puzzles.

Experimental results demonstrate that hard Sudokus require more computational time depending on the encoding used, and that a pattern of givens and solutions emerge. These patterns could fuel a new way of looking at hard Sudokus.

*Key words*: Sudoku, SAT, CNF, Pattern, Automated Reasoning.

## Introduction

Sudoku puzzles[1] have been a widely studied topic in the study of automated reasoning in artificial intelligence since 2005 when the puzzle became an international hit. The study of hard Sudokus, puzzles which are difficult to solve by humans, have been particularly popular amongst AI researchers.

A Sudoku puzzle consists of a 9 by 9 grid. A 9×9 Sudoku has 9 rows, 9 columns, and 9 blocks[2]. In total this 9×9 grid has 81 cells. The rules for solving a Sudoku are:

1. The cell values of a 9×9 Sudoku are: 1, 2, 3, 4, 5, 6, 7, 8, or 9.
2. The puzzle is considered solved when:
    a. All 81 cells have a value between 1 and 9,
    b. All the cells of a row have different values,
    c. All the cells of a column have different values,
    d. All the cells of a block have different values.
3. A puzzle has only one solution.
4. A puzzle starts with a number of givens[3].

Although many studies have investigated Sudoku as a SAT problem, probably no studies have investigated patterns of givens, and patterns of solutions of hard Sudokus when using a SAT solver to solve the Sudoku puzzle.

There are Sudokus with other grid configurations, like for example 16×16, 25×25 or 81×81, which are not part of the research described in this paper. This

[1] For more information about the history of Sudokus one could read the Sudoku Wiki (http://en.wikipedia.org/wiki/Sudoku).

[2] Also called regions or sub-grids.

[3] Also called clues.

paper is part of the assignment of the Automated Reasoning in Artificial Intelligence course[4] with research questions:

1. Is there a direct relation between classes of Sudoku problems and the computational hardness of their related SAT problem?
2. What is the pattern[5] of givens and what is the pattern of solutions of hard Sudokus?

The hypotheses were:

1. There is a relation between classes of Sudoku problems and the computational hardness of their related SAT problem.
2. Hard Sudokus have a pattern of givens.
3. Solutions of hard Sudokus have popular numbers for each cell; i.e. numbers that occur frequently in a specific cell for each unique solution.

The content of this paper is organized as follows: the next section describes a Sudoku puzzle in a SAT context. Subsequently, the research design is discussed, followed by the results of the experiment. Finally, discussion and conclusions are described and suggestions for further research are given.

## Sudoku and SAT

Lynce and Ouaknine [8] state that a Sudoku puzzle can easily be represented as a SAT problem, albeit one require a significant number of propositional variables. Encoding Sudoku puzzles into CNF requires 9 × 9 × 9 = 729 propositional variables. The 9×9 grid covers all the possible cells. Every cell can have a number from 1 to 9. Because a SAT problem is represented using propositional variables, each cell is associated with 9 propositional variables. These propositional variables can be assigned truth values 0 (false) or 1 (true). A variable of the Sudoku puzzle can be represented as $s_{xyz}$. Variable $s_{xyz}$ is true *if and only if* the cell in row $x$ and column $y$ is assigned number $z$; i.e. $[x,y] = z$. For example, $s_{136} = 1$ means that the cell [1,3] = 6, or in case a human is filling in the Sudoku puzzle, a number 6 has been written in the cell with row 1 and column 3. The set of propositional variables for the Sudoku puzzle is $V = \{(x, y, z)|1 \leq x, y, z \leq 9\}$ . The Suduko rules are represented as a set of nine-ary and binary clauses in CNF. The number of clauses depends on which encoding is used to encode the Sudoku puzzle. The givens of the Sudoku puzzle are represented as unit clauses. The number of unit clauses depends on the number of givens.

There are different encodings for Sudoku puzzles. The minimal and extended encoding by Lynce and Ouaknine [8], the efficient encoding by Weber [11], and the optimized encoding by Kwon and Jain [6]. The efficient encoding is in between minimal and extended. The optimized encoding uses an optimization algorithm to optimize the CNF clauses before running a SAT solver. Minimal encoding is powerful enough to solve hard 9×9 Sudokus. For n×n Sudokus where n=16, 25, 36, 49, 64, or 81, the efficient, extended or optimized encodings are advised. Lynce and Ouaknine [8] presented the formulas, which represent the Sudoku rules for minimal encoding, as follows:

There is at least one number in each entry (definedness):

$$Cell_d = \bigwedge_{x=1}^{9} \bigwedge_{y=1}^{9} \bigvee_{z=1}^{9} s_{xyz}$$

Each number appears at most once in each row (uniqueness):

$$Row_u = \bigwedge_{y=1}^{9} \bigwedge_{z=1}^{9} \bigwedge_{x=1}^{8} \bigwedge_{i=x+1}^{9} (\neg s_{xyz} \vee \neg s_{iyz})$$

[4] As part of the Master AI study at the Vrije Universiteit Amsterdam 2011.

[5] There are pattern Sudokus where colors are used instead of numbers. These Sudokus are not investigated in this paper.

Each number appears at most once in each column:

$$Column_u = \bigwedge_{x=1}^{9}\bigwedge_{z=1}^{9}\bigwedge_{y=1}^{8}\bigwedge_{i=y+1}^{9}(\neg s_{xyz} \vee \neg s_{xiz})$$

Each number appears at most once in each 3×3 block:

$$Block_u = \bigwedge_{z=1}^{9}\bigwedge_{i=0}^{2}\bigwedge_{j=0}^{2}\bigwedge_{x=1}^{3}\bigwedge_{y=1}^{3}\bigwedge_{k=y+1}^{3}(\neg s_{(3i+x)(3j+y)z} \vee \neg s_{(3i+x)(3j+k)z})$$

$$Block_u = \bigwedge_{z=1}^{9}\bigwedge_{i=0}^{2}\bigwedge_{j=0}^{2}\bigwedge_{x=1}^{3}\bigwedge_{y=1}^{3}\bigwedge_{k=x+1}^{3}\bigwedge_{l=1}^{3}(\neg s_{(3i+x)(3j+y)z} \vee \neg s_{(3i+k)(3j+l)z})$$

The efficient encoding includes all the clauses of the minimal encoding, as well as the following constraint:

Each number appears at most one number in each entry:

$$Cell_u = \bigwedge_{x=1}^{9}\bigwedge_{y=1}^{9}\bigwedge_{z=1}^{8}\bigwedge_{i=z+1}^{9}(\neg s_{xyz} \vee \neg s_{xyi})$$

The extended encoding includes all the clauses of the efficient encoding, as well as the following constraints:

Each number appears at least once in each row:

$$Row_d = \bigwedge_{y=1}^{9}\bigwedge_{z=1}^{9}\bigwedge_{x=1}^{9} s_{xyz}$$

Each number appears at least once in each column:

$$Column_d = \bigwedge_{x=1}^{9}\bigwedge_{z=1}^{9}\bigwedge_{y=1}^{9} s_{xyz}$$

Each number appears at least once in each block:

$$Block_d = \bigwedge_{z=1}^{9}\bigwedge_{i=0}^{2}\bigwedge_{j=0}^{2}\bigwedge_{x=1}^{3}\bigwedge_{y=1}^{3} s_{(3i+x)(3j+y)z}$$

The formula for the minimal encoding according to Lynce and Ouaknine [8] is:

$$\phi = Cell_d \cup Row_u \cup Column_u \cup Block_u \cup Assigned$$

The formula for the efficient encoding according to Weber [11] is:

$$\phi = Cell_d \cup Cell_u \cup Row_u \cup Column_u \cup Block_u \cup Assigned$$

The formula for the extended encoding according to Lynce and Ouaknine [8] is:

$$\phi = Cell_d \cup Cell_u \cup Row_d \cup Row_u \cup Column_d \cup Column_u \cup Block_d \cup Block_u \cup Assigned$$

where *Assigned* is the encoding for the givens of the Sudoku puzzle. These givens are represented by unit clauses $s_{xyz}$.

Table 1 shows the number of generated clauses by the different encodings. The high number of nine-ary clauses of the extended encoding is due to the encoding of *each number appears at least once in each block*. This line adds an extra of 243 clauses.

**Table 1. Number of Clauses per Encoding.**

| Encoding | Nine-ary clauses (set of at-least-one clauses) | Binary clauses (sets of at-most-one clauses) | Total # clauses |
|---|---:|---:|---:|
| *Minimal* | 81 | 8,748 | 8,829 |
| *Efficient* | 81 | 11,664 | 11,745 |
| *Extended* | 324 | 11,664 | 11,988 |

From Lynce & Ouaknine [8] and Gordon Royle [9] we learned that the minimum number of givens, or clues, is 17. To assist conversion from CNF to the Sodoku domain, not 729 but 999 variables are introduced, where all variables containing a 0 would simply be ignored. Then, 133 could mean that number 3 holds in cell S(1,3), etc. Now it is possible to read the solution from the output of the SAT solver.

The CNF files for the SAT solver are represented in CNF DIMACS format [2]. The input file for the SAT solver begins with comment lines which are indicated with a lower-case character c. For example: `c This is a comment line.` The problem line is indicated with a "p". The format is: `p format #variables #clauses`. For example for the Sudoku problem it would be: `p cnf 999 8864.` The clauses appear immediately after the problem line. A negated variable is represented by a negative number and a non-negated variable is represented by a positive number. Each clause is terminated by the value 0. For example, in the Sudoku problem on row 1 and column 4 there is number 3 would be represented as 143. If there was not a number 3 then it was notated as -143. The output file consists of one or more comment lines, followed by a clause satisfaction line. This clause satisfaction line has the format "s S", where S can be `SATISFIABLE`, `UNSATISFIABLE` or `UNKNOWN`. The variable line starts with a lower-case character v followed by the values of the solution. A positive value *x* means that it could be set *true* or a negative value –*x* means that it should be set *false*.

## Research Design

To test the hypotheses code had to be written to process the Sudoku puzzles, to create the appropriate CNF file for the SAT solver, to run the SAT solver, to process the results of the SAT solver and to present the results of the analysis. The formulas for the encoding of the Sudoku rules were obtained from Lynce and Ouaknine [8]. These formulas were programmed in LISP to generate the proper CNF clause for the Sudoku rules. The minimal, efficient, and extended encodings were used for all sets of givens.

The steps taken by the program are presented in figure 1 and are described

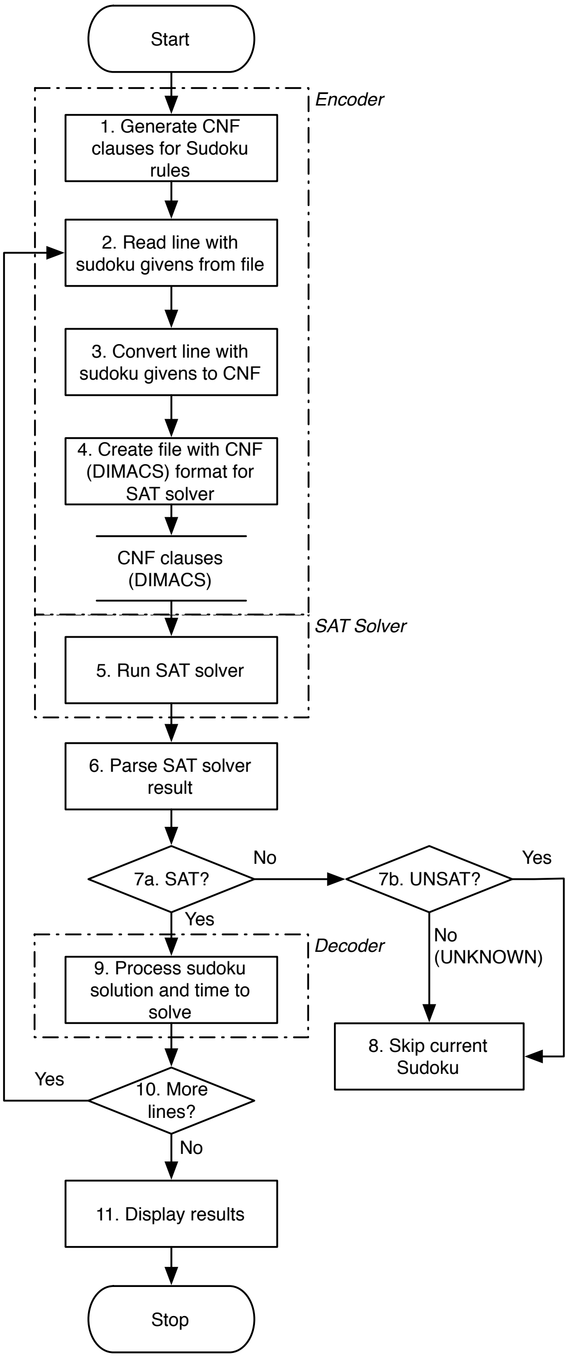
Start
Encoder
1. Generate CNF clauses for Sudoku rules
2. Read line with sudoku givens from file
3. Convert line with sudoku givens to CNF
4. Create file with CNF (DIMACS) format for SAT solver
CNF clauses (DIMACS)
SAT Solver
5. Run SAT solver
6. Parse SAT solver result
7a. SAT?
No
7b. UNSAT?
Yes
Yes
No (UNKNOWN)
Decoder
9. Process sudoku solution and time to solve
8. Skip current Sudoku
Yes
10. More lines?
No
11. Display results
Stop

**Figure 1. Process Overview**

in more detail below.

Step 1. Generates CNF clauses for the Sudoku rules. Sudoku puzzle givens are presented as a line with numbers or as a line with numbers and dots. For example, 000100200304005 or ...1..2..3.4..5. The 0 and the dot '.' represent an empty cell. Both formats could be read in their own format or when mixed. During this experiment files with easy, hard and extreme Sudoku givens were used to measure the time the SAT solver needed to solve the puzzles.

Step 2 and 3. The file with Sudoku givens was read line by line. Every line of givens was converted to unit clauses in CNF DIMACS format.

Step 4. The number of unit clauses was added to the number of clauses to encode the Sudoku rules. This step was necessary, because it is not known in advance how many givens there are in a line read from the file with givens. The complete CNF file in DIMACS format for the current Sukoku puzzle was saved on disc.

Step 5. The SAT4J solver was called from the program. The file saved in step 4 was used by the SAT4J solver as input. The SAT4J solver generated an output file in DIMACS format.

Step 6. This output file was parsed by the program for the following strings:

- The solution line "`s SATISFIABLE`" or "`s UNSATISFIABLE`" to determine if the Sudoku puzzle could be solved.
- The variable line "v" containing the numbers of the solution.
- The comment section "`c ... done. Wall clock time 0.067 s.`" and "`c Total wall clock time (in seconds) : 0.621`" to determine the time needed by the solver to solve the puzzle. The time in seconds depends on the time the solver needed to solve the puzzle and is variable. The latter was subtracted from the first, which represented the time needed to read the input file, to give the time the solver needed for finding the solution.

Step 7 and 8. If *satisfiable* then process the Sudoku puzzle solution as presented by the SAT solver and store the time needed by the SAT solver to solve the puzzle. If *unsatisfiable* then skip the Sudoku puzzle and do not use it for calculating the end results. When the SAT solver reported *unsatisfiable*, the current Sudoku was skipped and was not included in the analysis. *Unknown* could mean an incorrect number of clauses.

Step 9. The Sudoku givens, solution and time it took by the SAT solver to solve it were stored for analysis.

Step 10. If there are more lines of givens in the input file then the process was started again with step 2, else continue with step 11.

Step 11. The results were analyzed and the patterns were displayed.

The experimental setup was as follows:

- Apple MacPro, 2 x 2.66 GHz Dual-Core Intel Xeon, 6GB RAM, Mac OS X version 10.6.7.
- LispWorks 6.0.1 LISP development environment. [7]
- SAT4J java SAT solver (org.sat4j.core.jar) [10]
- Sudoku sets with givens from:
    - Vengard Hansen Easy (10 puzzles) [3].
    - Vengard Hansen Top 10 Hard [3].
    - Arto Inkala Top 10 (classified as hard to extreme) [4].
    - Gordon Royle Set of 49,151 puzzles with all 17 givens classified as hard [9].

LISP was used as a programming language for its easy prototyping and the ability to process all types of input. The SAT4J SAT solver was used because it is fast and it produces output in DIMACS format which is easy to process.

The analysis for finding the patterns was performed on the complete set of 49,151 lines of Sudoku givens from Gordon Royle [9].

The sets of Vengard Hansen and Arto Inkala were used to determine the hard Sudoku time threshold. The Sudokus which took less than the hard Sudoku time threshold to process were considered to be easy and their numbers were not used in the final analysis.

## Results

This section describes the results of the analysis. First, the times for processing the different set of Sudoku givens by the SAT solver using different encodings are given. Second, the times needed for processing the different set of Sudoku givens by the SAT solver and a brute force algorithm are given. Finally the patterns from the analysis of the Gordon Royle set [9] are presented.

### Time to Solve

Table 2 shows the experimental results of processing time using different encodings to find a solution for a Sudoku puzzle by the SAT solver. Figure 2 givens the graphical representation for the average processing time.

**Table 2. SAT Solver processing time.**

| **Set** | **Minimal Encoding** | **Efficient Encoding** | **Extended Encoding** |
|---|---|---|---|
| | **Time to solve** | **Time to solve** | **Time to solve** |
| *Vengard Hansen 10 Easy (average 28 givens)* | Slowest: 0.055 s.<br>*Average: 0.0264 s.*<br>Fastest: 0.017 s. | Slowest: 0.057 s.<br>*Average: 0.0347 s.*<br>Fastest: 0.022 s. | Slowest: 0.035 s.<br>*Average: 0.0271 s.*<br>Fastest: 0.022 s. |
| *Vengard Hansen Top 10 Hard (average 20 givens)* | Slowest: 0.161 s.<br>*Average: 0.1115 s.*<br>Fastest: 0.024 s. | Slowest: 0.139 s.<br>*Average: 0.1069 s.*<br>Fastest: 0.035 s. | Slowest: 0.064 s.<br>*Average: 0.0387 s.*<br>Fastest: 0.029 s. |
| *Arto Inkala AI Sudoku Top 10 (average 23 givens)* | Slowest: 0.13 s.<br>*Average: 0.0784 s.*<br>Fastest: 0.024 s. | Slowest: 0.128 s.<br>*Average: 0.0794 s.*<br>Fastest: 0.033 s. | Slowest: 0.0701 s.<br>*Average: 0.051 s.*<br>Fastest: 0.03 s. |
| *Gordon Royle Hard Sudokus with 17 givens (49,151 puzzles)* | Slowest: 0.787 s.<br>*Average: 0.0929 s.*<br>Fastest: 0.017 s. | Slowest: 0.823 s.<br>*Average: 0.1024 s.*<br>Fastest: 0.0278 s. | Slowest: 0.093 s.<br>*Average: 0.0332 s.*<br>Fastest: 0.021 s. |

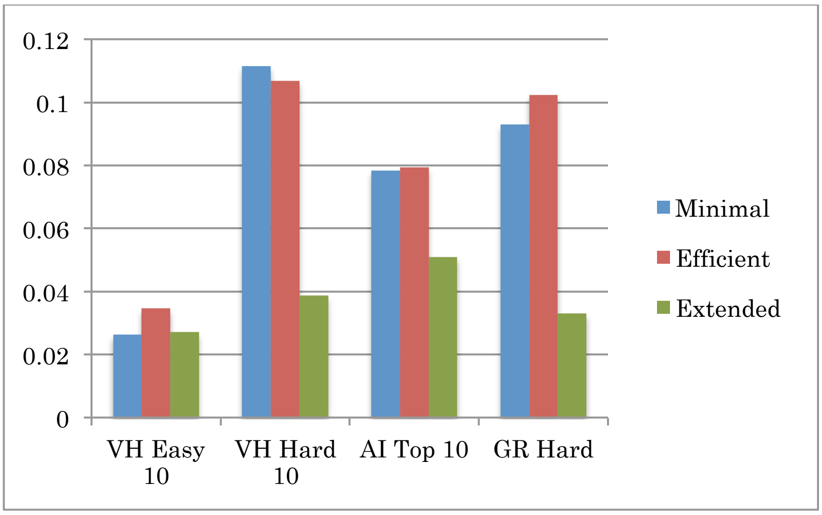

**Figure 2. Average Processing Time per Encoding.**

By using the numbers from table 2 the hard Sudoku time threshold was set to: 0.03 sec. Total computing time for the Gordon Royle set [9] was 5h:45m for the extended encoding, 6h:22m for the minimum encoding, and 6h:38m for the efficient encoding.

To put computing time in a wider perspective a brute-force algorithm [1] was used for a random line of givens of each set of givens. The time the brute-force algorithm needed to solve easy or hard Sudokus was dependent on the number of givens. For example, to solve an easy Sudoku it took 0.32 sec. And to solve a hard Sudoku it took between 0.358 sec and 8 min 25 sec to solve. Due to these long times, the brute-force algorithm was not used for the complete Gordon Royle set of 49,151 puzzles as comparison. Table 3 shows the experimental results from brute-force versus SAT with minimal encoding. The Sudoku solutions were the same.

**Table 3. Brute-force versus SAT with minimal encoding.**

| **Set** | **Brute Force** | **Minimal Encoding** |
|---|---|---|
| *Vengard Hansen 10 Easy (23 givens)* | 0.32 s. | 0.036 s. |
| *Vengard Hansen Top 10 Hard (21 givens)* | 4.52 s. | 0.114 s. |
| *Arto Inkala AI Sudoku Top 10 (23 givens)* | 0.358 s. | 0.045 s. |
| *Gordon Royle Hard Sudokus with 17 givens (49,151 puzzles)* | 8 m 25 s. | 0.039 s. |

The used lines of givens were (selected at random):
From VH Easy:
"....1...7.1.....5.4...3..8..6..7......59..2.4......9...26.....95....2..11.8..5...″.
From VH Hard:
"....839..1......3...4....7..42.3....6.......4....7..1..2........8...92.....25...6".
From AI Sudoku:
"4...6..7.......6...3...2..17....85...1.4......2.95..........7.5..91...3...3.4..8.".
From Gordon Royle:
"000000010400000000020000000000050407008000300001090000300400200050100000000806000".

## Pattern of Givens

The Gordon Royle [9] set of 49,151 lines with each 17 givens was used to determine if there were any patterns in givens and solutions. Table 4 shows the experimental results of the analysis of the number of givens per cell.

**Table 4. Number of Givens per Cell (out of 49,151).**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 18321 | 15892 | 9726 | 18472 | 15133 | 9412 | 17118 | 14912 | 9681 |
| 14430 | 11954 | 6501 | 14616 | 11209 | 6592 | 13487 | 11523 | 6784 |
| 7580 | 5653 | 1938 | 7511 | 4966 | 2096 | 6252 | 4682 | ***1834*** |
| 20620 | 15035 | 8941 | ***21009*** | 15657 | 9902 | 18893 | 15878 | 9808 |
| 15937 | 11355 | 5909 | 15435 | 11195 | 5899 | 14123 | 11294 | 6220 |
| 10101 | 6210 | 2528 | 9807 | 6167 | 2480 | 7664 | 5803 | 2462 |
| 19733 | 14056 | 8051 | 20105 | 14381 | 8821 | 19686 | 15009 | 9058 |
| 16310 | 11208 | 5932 | 16377 | 11314 | 6019 | 14547 | 10849 | 5827 |
| 10522 | 6348 | 2451 | 10590 | 6014 | 2274 | 8063 | 5265 | 2150 |

Lowest number of givens for a cell: 1834. Highest number of givens for a cell: 21009.

Table 5 shows the values of table 4 as percentages. For example, Cell [1,1] contained a given number in 37.3% of the time, where Cell [3,9] had a number only for 3.7% of the time. These values were converted to colors. A color pattern makes it easier to spot a pattern in a collection of numbers. Figure 3 shows the color pattern of the Gordon Royle set.

**Table 5. Percentages of Givens per Cell.**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 37.30% | 32.30% | 19.80% | 37.60% | 30.80% | 19.10% | 34.80% | 30.30% | 19.70% |
| 29.40% | 24.30% | 13.20% | 29.70% | 22.80% | 13.40% | 27.40% | 23.40% | 13.80% |
| 15.40% | 11.50% | 3.90% | 15.30% | 10.10% | 4.30% | 12.70% | 9.50% | **3.70%** |
| 42.00% | 30.60% | 18.20% | **42.70%** | 31.90% | 20.10% | 38.40% | 32.30% | 20.00% |
| 32.40% | 23.10% | 12.00% | 31.40% | 22.80% | 12.00% | 28.70% | 23.00% | 12.70% |
| 20.60% | 12.60% | 5.10% | 20.00% | 12.50% | 5.00% | 15.60% | 11.80% | 5.00% |
| 40.10% | 28.60% | 16.40% | 40.90% | 29.30% | 17.90% | 40.10% | 30.50% | 18.40% |
| 33.20% | 22.80% | 12.10% | 33.30% | 23.00% | 12.20% | 29.60% | 22.10% | 11.90% |
| 21.40% | 12.90% | 5.00% | 21.50% | 12.20% | 4.60% | 16.40% | 10.70% | 4.40% |

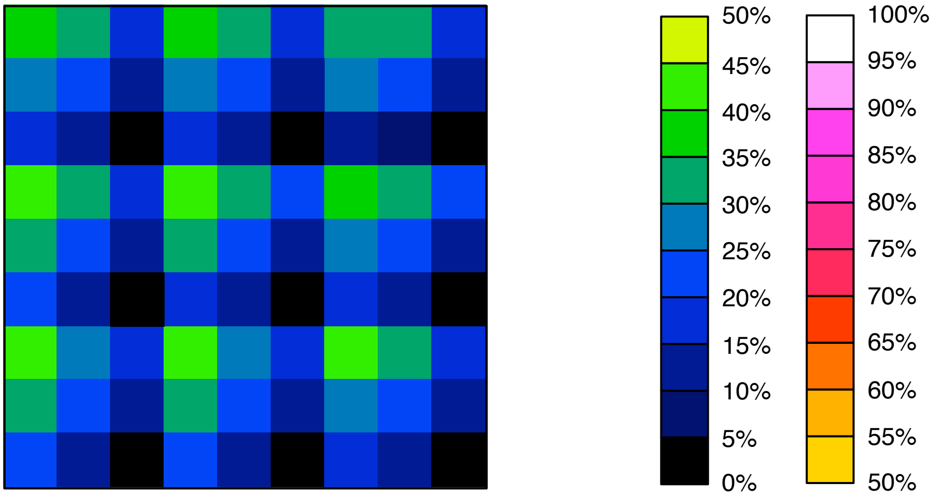

**Figure 3. Color Pattern of Sudoku Givens.**

### Pattern of Solution

Figure 4 shows the popular solution values per cell. All the satisfiable solutions of the SAT solver for hard Sudokus were analyzed to determine the frequency a number was assigned to a cell. All 49,151 puzzles of the Gordon Royle [9] set were used. Figure 5 shows this pattern in color.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 6 | 1 | 6 | 7 | 1 | 3 | 4 | 1 |
| 2 | 2 | 1 | 3 | 7 | 1 | 8 | 2 | 1 |
| 1 | 1 | 5 | 1 | 1 | 5 | 1 | 1 | 6 |
| 6 | 9 | 1 | 6 | 4 | 1 | 7 | 9 | 1 |
| 7 | 7 | 2 | 8 | 9 | 2 | 4 | 1 | 1 |
| 1 | 1 | 6 | 1 | 1 | 6 | 1 | 1 | 6 |
| 6 | 9 | 1 | 6 | 3 | 1 | 1 | 8 | 1 |
| 4 | 1 | 2 | 8 | 7 | 3 | 9 | 9 | 1 |
| 1 | 1 | 7 | 1 | 1 | 6 | 1 | 1 | 3 |

**Figure 4. Popular Solution Numbers per Cell.**

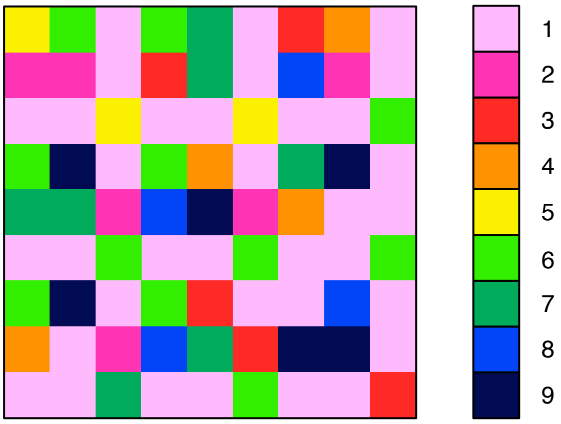

**Figure 5. Color Pattern of Solution Numbers.**

From figure 4 and 5 one could see that some numbers are more popular than others. Figure 6 shows the frequency of each number from 1 to 9. As shown in figure 4 number 1 has the highest frequency.

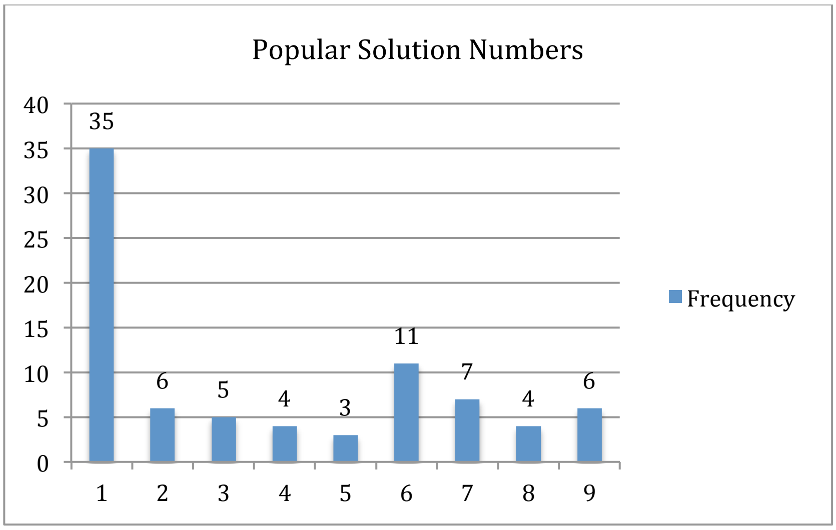

**Figure 6. Frequency of Popular Solution Numbers.**

## Discussion and Conclusion

The results from this research suggest that there is a relation between classes of Sudoku puzzles and the computational hardness of their related SAT problem. Table 2 and figure 2 suggest that hard Sudokus take on average 3.5 times longer to compute for minimal encoding and on average 1.5 times longer for extended encoding. Further, it also suggests that hard Sudokus have a pattern of givens as presented in table 5 and figure 3, and that solutions of hard Sudokus have a popular numbers in specific cells as presented in figure 4, 5 and 6.

Different encodings can give different results. The results in table 2 support the hypothesis that hard Sudokus are also computational hard. The number of givens does not seem to be the only factor that determines hard Sudokus. The position of the givens in the Sudoku grid can also be a factor that influences computational hardness.

A pattern of givens can be seen from figure 3 and one can deduce that cells with high percentages of givens can be suitable candidate for creating a hard

Sudoku; i.e. put a number in these cells first. The numbers behind the pattern as listed in table 5 and table 4 show for example, that cell [4,4] contained a given number in 42.7% of the Gordon Royle puzzle set, where cell [3,9] had a given number only for 3.7% of the occurrences. The first cell seems to be popular for Sudoku givens and the latter cell is probably not so popular. These findings can support the hypothesis that hard Sudokus have a pattern of givens.

Figure 4 shows the popular solution values per cell. This can suggest that when solving a Sudoku the values with high frequency are the best to start with in the cell as indicated in figure 4. For example, start with a 5 in cell [1,1]. Number 1 is by far the most popular solution value. For future research it could be interesting to explore this programmatically. The results presented in figure 4, 5 and 6 can support the hypothesis that solutions of hard Sudokus have popular numbers for each cell.

Brute-force does not profile itself as an ideal solution to solve hard Sudokus. The research shows that a SAT solver can be much faster is solving hard Sudokus. Brute-force does not use any reasoning like SAT does. This can indicate that by using intelligence, in this case in the form of CNF formulas, better results can be obtained.

The results from this research are similar with that of Lynce and Ouaknine [8], and Kwon and Jain [6]. Extended encoding is the fastest for solving hard Sudokus. The number of clauses can cause the difference in computational times between the different encodings. Lynce and Ouaknine state that the minimal encoding suffices to characterize Sudoku puzzles, whereas the extended encoding adds redundant clauses to the minimal encoding. These additional clauses can be justified in terms of resolution, and therefore do not alter the solution of a problem instance.

The hard Sudoku time threshold depends on encoding used, and on the experimental setup. Therefore, a few test runs with easy and hard Sudokus have to be performed to determine the correct value. For example, on a supercomputer this threshold can be different than on a laptop computer.

From the results, some conclusions can be drawn. First, the findings in table 4, 5 and figure 4 can support the hypothesis that hard Sudokus have a pattern of givens. Second, the results presented in figure 4, 5 and 6 can support the hypothesis that solutions of hard Sudokus have popular numbers for each cell. Finally, The results in table 2 support the hypothesis that hard Sudokus are also computational hard.

This research does not cover the influence of the position of givens on the computational hardness. Future research could include the investigation how to use the patterns to create more efficient encodings for solving Sudoku using SAT solvers.

## References

[1] Buß, Frank. LISP code at http://www.frank-buss.de/lisp/. August 02, 2006.

[2] DIMACS. Satisfiability Suggested Format, last revision, May 8, 1993. http://dimacs.rutgers.edu.

[3] Hansen, Vengard. Menneske. http://www.menneske.no/sudoku/eng/top10.html

[4] Inkala, Arto. AI Sudoku Top 10. http://www.aisudoku.com/en/AIsudoku_Top10s1_en.pdf

[5] Kwon, G-H. (2006). SAT Encodings for Sudoku. *Presentation September 26, 2006.*

[6] Kwon, G. & Jian, H. (2006). Optimized CNF Encoding for Sudoku Puzzles. *13th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR 2006).*

[7] LispWorks. www.lispworks.com

[8] Lynce, I., & Ouaknine, J. (2006). Sudoku as a SAT Problem. *The Proceedings of AIMATH'06.*

[9] Royle, Gordon. University of Western Australia. http://mapleta.maths.uwa.edu.au/~gordon/sudokumin.php

[10]SAT4J solver. http://forge.ow2.org/projects/sat4j

[11]Weber, T. (2005). A SAT-based Sudoku Solver. *The Proceedings of LPAR'05.*