A Sudoku-Solver for Large Puzzles using SAT

Uwe Pfeiffer, Tomas Karnagel and Guido Scheffler

Technische Universitt Dresden (TUD), Germany
{Uwe.Pfeiffer,Tomas.Karnagel}@mailbox.tu-dresden.de, guido.scheffler@email.de

Abstract

$n^2 \times n^2$ Sudoku puzzles can be straightforwardly encoded as SAT problems. However, solving such puzzles for large n requires a significant amount of optimization. We present some ideas for reducing the number of clauses, for improving the encoding, and for the selection of suitable SAT solvers.

1 Introduction

Sudoku puzzles are quite common these days. In a $n^2 \times n^2$ Sudoku the aim is to assign the numbers 1 to n^2 to each of the $n^2 \times n^2$ cells arranged in $n \times n$ square blocks, in a way that every number appears only once in each row, column and block. There are many ways of solving such problems automatically. We are using the well-known technique of solving satisfiability (SAT) problems. It is possible to encode the Sudoku problem with all constraints in Conjunctive Normal Form (CNF)[1, 2], which is used as input format for most SAT solvers. [3, 4] has shown how to solve Sudokus with $n = 12$ efficiently using a SAT solver with an optimized CNF encoding. Other approaches using evolutionary algorithms (see e.g. [5, 6, 7, 8, 9]) or considering Sudokus as constraint problems (see e.g. [10, 11]) turned out to be not nearly as fast as the SAT approaches or were not even able to solve the Sudoku puzzles at all, and we are unaware of any published system which was able to solve Sudokus for $n > 12$.

The aim of our work was to build a Sudoku solver for participating in a local competition at the TUD where Sudoku puzzles with $3 \leq n \leq 15$ had to be solved using SAT solvers. The goal of the competition was to build a correct system, which should solve as many Sudokus as fast as possible given a certain time-out per puzzle. This paper presents the architecture of our solver and the decisions that were made during development.

2 Encoding Sudoku Problems as Formulas in CNF

Each cell of a $n^2 \times n^2$ Sudoku puzzle can contain a number between 1 and n^2. This can be represented by a triple (x, y, v), where x denotes the column, y the row, and v the assigned number. In a valid solution of a $n^2 \times n^2$ puzzle, each cell, row, column, and block contains exactly one element of $[1, n^2]$. This can be encoded with two conditions. The first one is often called *definedness* and can be encoded for cells in the formula

$$\bigwedge_{x=1}^{n^2} \bigwedge_{y=1}^{n^2} \bigvee_{v=1}^{n^2} (x, y, v).$$

The second condition is *uniqueness*. For cells we obtain

$$\bigwedge_{x=1}^{n^2} \bigwedge_{y=1}^{n^2} \bigwedge_{v=1}^{n^2-1} \bigwedge_{w=v+1}^{n^2} (\neg(x, y, v) \vee \neg(x, y, w)).$$

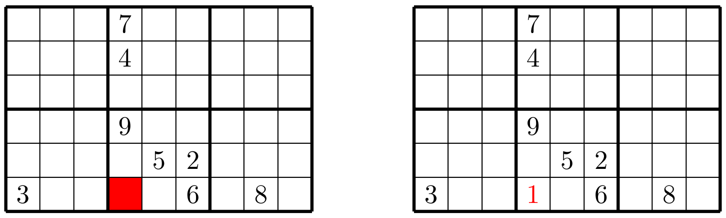

Figure 1: A Naked Single leading to the unit clause $(4, 6, 1)$.

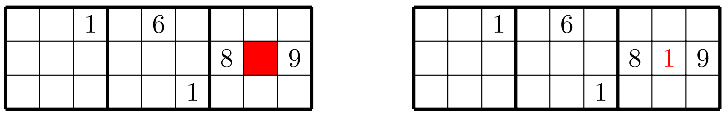

Figure 2: A Hidden Single leading to the unit clause $(8, 2, 1)$.

Similar definedness and uniqueness conditions are to be specified for rows, columns, and blocks (see [3]).

In a Sudoku puzzle some values are assigned to some cells initially. This can be used to remove redundancies by reducing the definedness and uniqueness formulas. For example, if it is known that the cell (x, y) is set to v, then any clause which contains (x, y, v) can be removed and from every remaining clause the literal $\neg(x, y, z)$ can be deleted. The reader is referred to [3] for more details. In the case of $15^2 \times 15^2$ Sudokus used in our experiments these techniques led to a reduction from more than 5×10^9 clauses to about 10^6 clauses.

3 Improving the Encoding and Selecting the Solver

In order to further reduce the number of clauses we applied additional preprocessing steps, which are described in the following subsection. Thereafter, the preprocessing techniques are evaluated by various SAT solvers.

3.1 Preprocessing

Ideally, domain dependent preprocessing techniques should speed up a SAT solver significantly without taking too much additional time. We have selected the following techniques.

Naked Single (NS) Using methods from Constraint Solving we record for each cell the number of possible values. If there is only one possible value v left for a cell (x, y), then the unit clause (x, y, v) is generated (see Figure 1).

Hidden Single (HS) If there is only one valid position left for a value in a row, column, or block, then a corresponding unit clause is generated (see Figure 2).

Intersection Removal (IR) If all possible occurrences of a certain number can be pinpointed to precisely two entities (an entity can be a row, column, or block), all other possible candidates for that number can be removed from those entities. For example if all possible occurrences of 1 in a row are in one block, then 1 must occur in the block in this particular row and all other possible occurrences of 1 in this block can be removed (see Figure 3).

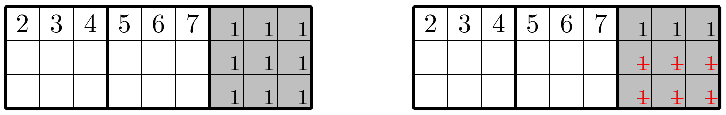

Figure 3: Intersection Removal: value 1 must occur in the top most row of the grey block.

Preprocessing	SAT4J		zchaff		MiniSAT		PrecoSAT		clasp	
	time	TO	time	TO	time	TO	time	TO	time	TO
none	7617	6	13784	27	5116	3	2843	0	2300	1
(NS) + (HS)	7053	6	11298	22	4391	3	2990	2	2050	1
(NS) + (HS) + (IR)	6130	5	9826	21	3543	3	2710	0	1653	1

Table 1: Performance gain of the preprocessing techniques; time in seconds,TO denotes time-outs.

In the following subsection we will discuss various SAT solvers. The performance gain of the domain dependent preprocessing techniques is shown in Table 1. Some simple puzzles were already solved after these preprocessing steps.

3.2 Systematic SAT Solvers

Initially, we considered the SAT solvers SAT4J-2.2.0, zchaff-2007.3.12, MiniSAT-2.070721, PrecoSAT-465r2-100514, and clasp-1.2.0-SAT09. All tests were made on a 3.0GHz Core2 Duo with 2GB of memory running Ubuntu Linux. We used 251 Sudokus with $3 \leq n \leq 15$ and a time-out of 300 seconds (see [12]). Table 1 shows that SAT4J, zchaff, and MiniSAT were outperformed by PrecoSAT and clasp as they were too slow and generated too many time-outs. But shall we prefer PrecoSAT over clasp?

3.3 Positioning of the Clauses

While trying to improve the generation times for the CNF files, we experimented with the order in which clauses are generated. Using the same 251 Sudoku puzzles as before, some of the tested orderings are shown in Table 2. The first ordering was the one used in the previous tests and lists the clauses for all the cells first and, thereafter, the clauses for rows, columns, and blocks. The second ordering uses the same idea but lists the uniqueness clauses first and, thereafter, all definedness clauses. The third ordering lists all information of a cell and its occurrence in row, column, and block as close as possible together.

Besides the standard parameter setting of clasp, we tested clasp with various heuristics, where VSIDS outperformed the other ones. After testing the different orderings and heuristics we selected clasp-VSIDS and the third clause ordering.

3.4 Stochastic SAT Solvers

We also considered the stochastic local search solvers WalkSAT [13] and UBCSAT [14].[1] We tested various heuristics and selected the best one for our Sudoku puzzles: RNovelty+ [15].

[1] The UBCSAT implementation differs from WalkSAT in that the statistics used in the RNovelty+ heuristics are generated completely from scratch instead of incrementally. This seems to speed up the solver by a factor of 2.

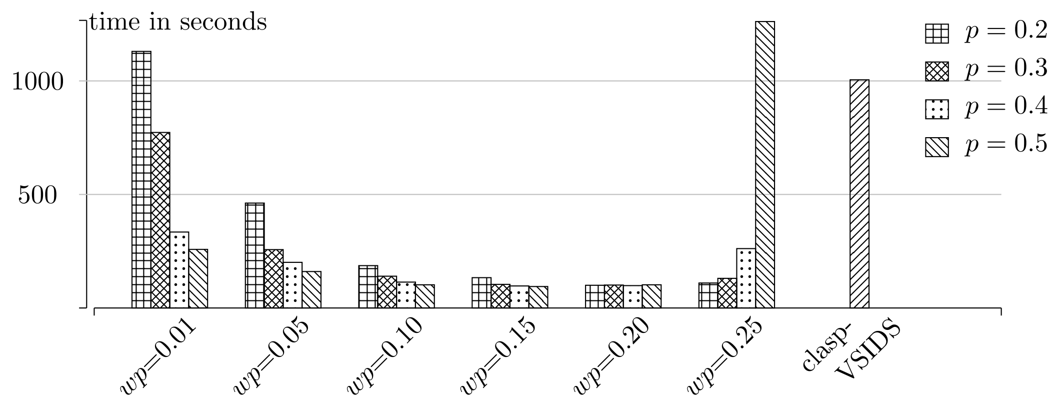

Figure 4: Different parameter settings for the search with UBCSAT for Sudokus with $n \geq 7$.

RNovelty+ has two major parameters: p and wp. Initially, the solver randomly assigns truth values to the variables. Thereafter, the heuristic chooses randomly a clause and flips a variable such that the number of satisfied clauses is maximized. If a variable is chosen for the second time, then depending on the *probability* p (see [16] for details) it is either flipped back or the second best variable is flipped. With *walk probability* wp there will be a random walk, i.e., a randomly chosen variable is flipped. In addition, there may be *restarts*. Based on our experiments we decided to avoid restarts as they seem to increase the runtime. We set the cut-off for restarts to 10^7 flips, which we never reached in our tests. Figure 4 shows the performance of different parameter settings for UBCSAT with RNovely+ as well as of clasp-VSIDS.

3.5 Comparison of clasp-VSIDS and UBCSAT

We had chosen clasp-VSIDS as the fastest systematic solver which could be used for all puzzle sizes. UBCSAT was our fastest stochastic solver, but – somewhat surprisingly – was unable to solve smaller puzzles given the time-out of 300 seconds (see Table 3). On the other hand, UBCSAT solved puzzles for $n > 6$ much faster than clasp-VSIDS. For the subset consisting of all puzzles with $n > 6$ clasp-VSIDS needed 1060 seconds whereas UBCSAT needed only 95 seconds.

4 The Final Solver Suite

Finally, we implemented *our* SAT based Sudoku solver in Java. We used the extended encoding presented in Section 2. We applied the domain dependent preprocessing techniques discussed

input clauses	PrecoSAT		clasp		clasp-VSIDS	
	time	TO	time	TO	time	TO
1st ordering	2710	0	1653	1	1395	0
2nd ordering	2765	0	1688	0	1437	0
3rd ordering	2461	0	1581	0	1368	0

Table 2: Evaluation of different orderings of the input clauses; time in seconds, TO denotes time-out.

in Subsection 3.1. We decided to use two instead of only one SAT solver: clasp-VSIDS for Sudokus with $n \leq 6$, which was about 300 seconds faster than PrecoSAT for puzzles of that size, and UBCSAT with RNovelty+ for $n > 6$.

To sum up, we started naively with SAT4J, some time-outs (of 300 seconds) and more than 2 hours on the mentioned benchmark of Sudoku puzzles, and ended up with a combination of clasp-VSIDS and UBCSAT solving all puzzles in 400 seconds.

The final competition involved 251 previously unseen Sudoku puzzles with $3 \leq n \leq 15$, however the time-out was now set to 60 seconds. Within this limit we solved all but 5 puzzles, where each unsolved puzzle was a $6^2 \times 6^2$ puzzle.

5 Conclusion and Future Work

We found the encoding of [3] quite appropriate for Sudoku puzzles. Preprocessing had a major impact and should be investigated further. We would like to have a better understanding to what extend the ordering of the input clauses influences the performance. Why is UBCSAT slow on small puzzles, but can solve the larger ones so easily? We would also like to rank puzzles according to their difficulty, but we don't know how to measure the difficulty. We would also be interested in puzzles with $n > 15$.

Acknowledgement The authors would like to thank Steffen Hlldobler for the encouragement to write this article and for providing advice.

References

[1] Tjark Weber. A sat-based sudoku solver. In 12 th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR 2005, pages 11–15, 2005.

[2] Ins Lynce and Jol Ouaknine. Sudoku as a sat problem. In In Proceedings of the 9 th International Symposium on Artificial Intelligence and Mathematics, AIMATH 2006, Fort Lauderdale. Springer, 2006.

[3] Gihwon Kwon and Himanshu Jain. Optimized cnf encoding for sudoku puzzles.

[4] Gihwon Kwon. Effect on preprocessing in sat with sudoku puzzle. pages 127–135, 2008.

[5] T. Mantere and J. Koljonen. Solving, rating and generating sudoku puzzles with ga. pages 1382–1389, 2007.

[6] T. Mantere and J. Koljonen. Solving and analyzing sudokus with cultural algorithms. pages 4053–4060, 2008.

[7] T. Mantere and J. Koljonen. Sudoku solving with cultural swarms. pages 60–67, 20.-22.August 2008.

Puzzle size n	3	4	5	6	7	8
clasp-VSIDS	0.0012	0.0066	3.9111	13.7748	13.6198	2.3302
UBCSAT	0.0098	1.7832	300(14 TO)	40.13(1 TO)	0.7989	0.2201
Puzzle size n	**9**	**10**	**11**	**12**	**13**	**15**
clasp-VSIDS	4.7218	10.0627	3.5732	9.5042	2.8468	7.5786
UBCSAT	0.2561	0.2202	0.3983	0.4437	0.5703	0.9775

Table 3: Time in seconds per puzzle for clasp-VSIDS and UBCSAT (with $p = 0.5$ and $wp = 0.15$).

[8] Meir Perez and Tshilidzi Marwala. Stochastic optimization approaches for solving sudoku. CoRR, abs/0805.0697, 2008.

[9] J. Almog. Evolutionary computing methodologies for constrained parameter, combinatorial optimization: Solving the sudoku puzzle. pages 1–6, sep. 2009.

[10] Helmut Simonis. Sudoku as a constraint problem. In Brahim Hnich, Patrick Prosser, and Barbara Smith, editors, Proc. 4th Int. Works. Modelling and Reformulating Constraint Satisfaction Problems, pages 13–27, 2005.

[11] T.K. Moon and J.H. Gunther. Multiple constraint satisfaction by belief propagation: An example using sudoku. pages 122–126, jul. 2006.

[12] Used Sudokus: `http://www.wv.inf.tu-dresden.de/Research/SATSolving/sudoku_sat_2010`.

[13] Bart Selman Henry, Henry A. Kautz, and Bram Cohen. Noise strategies for improving local search. pages 337–343. MIT press, 1994.

[14] Dave A. D. Tompkins and Holger H. Hoos. Ubcsat: An implementation and experimentation environment for sls algorithms for sat and max-sat. In In SAT, pages 37–46, 2004.

[15] Holger H. Hoos. On the run-time behaviour of stochastic local search algorithms for sat. In In Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99), pages 661–666, 1999.

[16] David Mcallester, Bart Selman, and Henry Kautz. Evidence for invariants in local search. In In Proceedings of AAAI-97, pages 321–326, 1997.