A Theory Learning SAT-Solver for Sudokus

Project Report

Jonsson, Haukur 11137304

Rajamanickam, Santhosh 11650702

Rapp, Max 11404310

October 10, 2017

1 Introduction

Sudokus are commonly studied paradigm for SAT-solvers as well as constraint programming. Constraint programming sacrifices generality for efficiency by tailoring solvers for hard problems using various constraints that go beyond the mere axioms of propositional logic. A common approach in constraint programming are "Satisfiablity-Modulo-Theories"(SMT)-solvers. They supplement a SAT-solver by a specialised theory(T)-solver exploiting both the high degree of optimization of SAT-solvers and the domain-specific efficiency of T-solvers.

For example, in the DPLL(T)-algorithm a DPLL-based procedure is used to SAT-check a formula F starting from an empty T-box. If the formula is satisfiable it is passed to a specialised T-solver that checks whether it is T-consistent. If not, $\neg F$ is added to the SAT-solver's axioms and so forth.

Both SAT and SMT approaches have their merits: SAT is a very general approach that requires minimal Human preprocessing while SMT can be much more efficient for specific domains.

In this report, we investigate the middle ground between the two approaches for the domain of 9x9 Sudokus. Namely, we do not use a T-solver to add to the axioms but check the clauses learnt by the SAT-solver on specific Sudokus for global validity on Sudokus. If a valid clause is found, it is added to the axioms.

Based on previous results we hypothesize that in this way it is possible to train SAT-solvers to become more efficient in the current domain. We attempt to keep the procedure as general as possible so that it can be applied to any domain, not only Sudokus.

The rest of this report is organised as follows: the results of previous work that led us to our hypothesis are summarised in section 3; we present our methodology including the choice of dataset, encodings, SAT-solver and performance metrics in section 4; in section 5 we propose an algorithm for SAT-theory learning including clause pruning, validity checking and validity pruning and various training protocols; section 6 summarizes our findings; finally, in section 7 we present our conclusions and suggest avenues for further research.

2 Preliminaries

Given a set of propositional variables, we call the variables and their negations *literals*. A *clause* c is a set of literals interpreted as a disjunction. We call a set S of clauses, interpreted as a conjunction of clauses, a $SAT-problem$ (in CNF). A function ϕ that maps some problem to a $SAT-problem$ is called an *encoding*. As far as possible we will treat a SAT-solver as a blackbox that tells us whether a SAT-problem in CNF is satisfiable and returns a satisfying assignment of the propositional variables. A SAT-problem is *proper* if it admits exactly one satisfying assignment. Given a set of SAT-problems $D = \{S_1, ..., S_n\}$ with $\cap_{i=1}^{n} S_i \neq \emptyset$ in CNF we refer to D as the *domain* of the problem, $A = \cap_{i=1}^{n} S_i$ as the $axioms$ of D and to $C_i = S_i \setminus \cap_{i=1}^{n} S_i$ as the *claims* or $assertions$ of problem i. For $c \in C$, if the SAT-solver returns "unsatisfiable" for $S = A \cup c$ we call $\neg c$ *valid* and a $theorem$. An axiom $a \in A$ is *redundant* iff it is a theorem for any domain $D_{\setminus a}$ with axioms $A_{\setminus a} = A \setminus a$. For any theorem, we call its minimal valid subclauses the $kernels$ of the theorem.

In this study D_{min}, $D(eff)$, and D_{ext} are the sets of proper SAT-problems obtained by applying the corresponding encoding functions $\phi_{minimal}$, $\phi_{efficient}$ and $\phi_{extended}$ to a dataset of Sudokus to be specified below. The encodings furnish different sets of axioms A_{min}, A_{eff} and A_{ext}. In addition, each individual SAT-problem is characterised by claims corresponding to the Sudoku's givens. We henceforth use Sudoku to refer to the SAT-problem corresponding to the Sudoku as well when the context is clear.

3 Related Work

Redundant axioms for Sudoku are mainly considered in constraint programming approaches. E.g. Demoen and de la Banda (2014, 11) find that: "Redundant constraints are very often good for the performance of CP systems, and indeed, all solvers we checked perform much slower (about a factor 2000) with a minimal set of big constraints." Simonis (2005) suggests several such redundant constraints and finds they boost the performance of CP propagation schemes.

As opposed to CP-systems SAT-solvers usually only have very limited propagation schemes such as unit propagation. However, the literature on Sudoku as a SAT-problem considers redundant axioms from a different perspective. A large part of it is devoted to optimizing CNF-encodings for Sudoku with respect to SAT-solver performance. Lynce and Ouaknine (2006) suggest and test two encodings: a *minimal* and an *extended* encoding. The extended encoding is nothing else than an encoding containing redundant axioms. They compare the encodings efficiency with respect to SAT-solvers' ability to solve Sudokus without search. They find among other things that substituting the minimal by the extended encoding increases the likelihood of solving a Sudoku without search using only unit propagation from 1% to 69%. In addition, Kwon and Jain (2006) compared these encodings with respect to runtime over various Sudoku-sizes and found that the extended encoding scales much better with increased problem size.

This apparent convergence in two strains of literature led us to hypothesize that redundant axioms might be beneficial to SAT-solver performance in spite of their limited inference tools. We thus wanted to find out which redundant axioms would be most helpful in strengthening inference and minimizing search. One way to go about this is to look for redundant axioms that the SAT-solver actually *learns* during the search process.

4 Methodology

We decided to test our hypothesis experimentally. In order to do this, it was necessary to choose a suitable dataset, SAT-solver and a meaningful metric of performance as well as a CNF-encoding of Sudoku.

4.1 Dataset

The domain of all proper Sudokus has a cardinality of more than $6.67 * 10^21$. Therefore it is necessary to work with a subdomain. We used the collection of minimum (i.e. 17 givens) proper sudokus assembled by Gordon Royle (*Minimal Sudoku Dataset*, n.d.). A minimal Sudoku is a Sudoku from which no given can be removed whilst remaining a proper Sudoku. The dataset consists of 49151 such sudokus.

From this dataset, we used 10000 sudokus to measure the performance of the SAT solver across different encodings and different base clauses. We used another 5000 sudokus to train our SAT solver to learn new clauses which are valid for sudokus.

4.2 Sudoku Encodings and DIMACS

We use the three different types of encoding functions specified in (Lynce & Ouaknine, 2006) and (Kwon & Jain, 2006) for our experiments: the minimal, the efficient, and the extended encoding.

Each encoding uses different axioms. Namely, $A_{min} \subset A_{eff} \subset A_{ext}$. Since the minimal encoding is a complete axiomatization of Sudoku, all of the additional axioms in the other encodings are redundant.

The encodings all have to encode the following four constraints:

- each cell has to be filled with a number from 1-9
- each row must contain each number exactly once
- each column must constain each number exactly once
- each square block (eight on the edges, one in the center) of nine cells must contain each number exactly once

We follow Kwon and Jain (2006) in representing them. The constraints are encoded using the same sets of propositional variables: in an 9×9 Sudoku puzzle each cell must contain a number between 1 and 9. Thus, each cell is associated with 9 propositional variables which can be represented by 3-tuples (r, c, v) as variables. Then we get $V = \{(r, c, v) | 1 \leq r, c, v \leq 9\}$ as the set of propositional variables

The difference in the encodings stems from the way they encode the constraints. The following conjunctions refer to each cell, row, column or block having at least one number from 1 to n (i.e. $definedness$) or at most one number from 1 to n (i.e. *uniqueness*). A subscript d to denote definedness constraints and subscript u to denote the uniqueness constraints.

$$Cell_d = \bigwedge_{r=1}^{n} \bigwedge_{c=1}^{n} \bigwedge_{v=1}^{n} (r, c, v)$$

$$Cell_u = \bigwedge_{r=1}^{n} \bigwedge_{c=1}^{n} \bigwedge_{v_i=1}^{n-1} \bigvee_{v_j=v_i+1}^{n} \neg(r, c, v_i) \vee \neg(r, c, v_j)$$

$$Row_d = \bigwedge_{r=1}^{n} \bigwedge_{v=1}^{n} \bigvee_{c=1}^{n} (r, c, v)$$

$$Row_u = \bigwedge_{r=1}^{n} \bigwedge_{v=1}^{n} \bigwedge_{c_i=1}^{n-1} \bigwedge_{c_j=c_i+1}^{n} \neg(r, c_i, v) \vee \neg(r, c_j, v)$$

$$Col_d = \bigwedge_{c=1}^{n} \bigwedge_{v=1}^{n} \bigvee_{r=1}^{n} (r, c, v)$$

$$Col_u = \bigwedge_{c=1}^{n} \bigwedge_{v=1}^{n} \bigwedge_{r_i=1}^{n-1} \bigwedge_{r_j=r_i+1}^{n} \neg(r_i, c, v) \vee \neg(r_j, c, v)$$

$$Block_d = \bigwedge_{r_{offs}=1}^{\sqrt{n}} \bigwedge_{c_{offs}=1}^{\sqrt{n}} \bigwedge_{v=1}^{n} \bigvee_{r=1}^{\sqrt{n}} \bigvee_{c=1}^{\sqrt{n}} (r_{offs} * \sqrt{n} + r, c_{offs} * \sqrt{n} + c, v)$$

$$Block_d = \bigwedge_{r_{offs}=1}^{\sqrt{n}} \bigwedge_{c_{offs}=1}^{\sqrt{n}} \bigwedge_{v=1}^{n} \bigwedge_{r=1}^{n} \bigwedge_{c=r+1}^{n}$$
$$\neg(r_{offs} * \sqrt{n} + (rmod\sqrt{n}), c_{offs} * \sqrt{n} + (rmod\sqrt{n}), v)$$
$$\vee \neg(r_{offs}\sqrt{n} + (rmod\sqrt{n}), c_{offs} * \sqrt{n}(cmod\sqrt{n}), v)$$

In addition one more conjunction is needed to represent givens. The givens are then represented as a conjunction of unit clauses. Let $G_i = \{(r_1, c_1, v_1), ..., (r_k, c_k, v_k)\}$ be the set of givens of Sudoku S_i. Then we have:

$$Assigned = \bigwedge_{i=1}^{k} (r, c, v), where (r_j, c_j, v_j) \in G_i$$

The differences in the three mentioned encodings then consist in which of the definedness and uniqueness conditions they include (Kwon & Jain, 2006):

$$\begin{aligned}
\phi_{minimal} &= Cell_d \wedge Row_u \wedge Col_u \wedge Block_u \wedge Assigned \\
\phi_{efficient} &= Cell_d \wedge Cell_u \wedge Row_u \wedge Col_u \wedge Block_u \wedge Assigned \\
\phi_{extended} &= Cell_d \wedge Cell_u \wedge Row_d \wedge Row_u \wedge Col_d \wedge Col_u \wedge \\
&\quad Block_d \wedge Block_u \wedge Assigned
\end{aligned}$$

Note that $\phi_minimal$ is a complete axiomatization of Sudoku and the additional axioms of the other encodings are thus redundant.

The encodings are represented in the DIMACS to pass them to the SAT-solver. It has the following format:

At the begining of the file there may be one or more comment lines. Comment lines start with a 'c'. They are followed by:

p FORMAT VARIABLES CLAUSES

FORMAT is for programs to detect which format is to be expected. In our case this is "cnf"

VARIABLES is the number of unique variables in the expression

CLAUSES is the number of clauses in the expression

This line is followed by the information in SAT-problem. Unique variables are enumerated from 1 to n. A negation is represented as '-'. Each line represents one disjunction and is delimited by "0". Example: $(A \vee \neg B \vee C) \wedge (B \vee D \vee E) \wedge (D \vee F)$ is represented in DIMACS as:

```
c This is a comment
c This is another comment
p cnf 6 3
1 -2 3 0
2 4 5 0
4 6 0
```

4.3 The Minisat-Solver

The SAT-problems in DIMACS format are then passed on to a SAT-solver. For our experiment, we use a SAT solver called Minisat. Minisat implements the DPLL algorithm and additionally implements conflict-driven learning and non-chronological backtracking. As Minisat has an upper limit on the number of learned clauses it employs an activity heuristic when deciding which learned clauses to keep when it. The same heuristic is applied when doing backjumping. The heuristic essentially records which clauses and literals have been in recent conflicts and favours them over "stale" clauses and literals. For more information see (een2003extensible).

For our purposes, we needed to record the learned clause for a given conflict. To do this we needed to edit the Minisat code such that when encountering a conflict the deduced learned clause is printed out. By printing out the learned clause straight away we avoided the upper limit of Minisat learned clauses.

4.4 Performance Metric

In addition, we had Minisat output the number of *decisions* per Sudoku. A decision consists in choosing a propositional variable and tentatively assigning it a truth value. The solver then checks whether the problem is satisfiable given the decision. Decisions occur when the SAT-solver exhausts its inference capabilities and resorts to search. Therefore the number of decisions is a good metric of the proportions of search vs inference the SAT-solver used in solving a SAT-problem. A low number of decisions indicates heavy use of inference, a high number indicates a lot of search. Given our hypothesis that redundant axioms can help inference, we would expect the number of decisions to decrease when more redundant axioms are added.

5 The learning Algorithm

The learning algorithm collects all learnt clauses reported by the SAT-solver, prunes the learnt clauses by removing clauses which are definitely not valid and then checks the validity of the remaining clauses. If a clause is found to be valid we add it to the encoding currently being used in order to avoid the SAT-solver learning the same clause twice.

To test for validity of a clause we need check if the negation of that clause is not satisfiable with the encoding. If the negation is not satisfiable then the clause is valid.

5.1 Clause Pruning

Since the number of learnt clauses can be very high for a single SAT-problem and we need to run a SAT-solver on each learnt clause it is important to remove clauses that we either know to be valid or not valid.

To detect valid clauses we check if the learnt clause is a superset of a clause in the extended encoding, i.e. clause in the encoding implies the learnt clause and therefore the learnt clause is valid. To detect a non-valid clause we first check if it is a unit clause, as they are never valid in Sudoku. Secondly we check if the clause is not satisfied in any of our previous solutions, i.e. we check if we have a known counter-example to the clause. This step prunes the most number of clauses, is rather fast and gets better with runtime. Lastly, we check if the clause has length 9 or less and contains exactly one negated variable as these clauses are never valid in sudokus (proof omitted). The effectiveness of our pruning went from being able to prune away about 60

5.2 Validity Pruning

As the learnt clause is not necessarily the kernel and we don't want valid clauses which contain redundant variables in our encoding, we need to find the kernel. For the cases which we know the valid clause to be a superset of a clause in some encoding, we know that the kernel is exactly the clause in the encoding. For the cases in which the learnt clause is not a superset of any encoding we need to compute the minimally valid clause. Without going into too much detail about the algorithm which computes the kernel we simply state that it has a worst case factorial runtime with the number of variables in a clause as we attempt to construct a smaller clause with different combinations of variables, each of which needs to be checked for validity.

6 Results

Below are the results of the experiments conducted with just using the base clauses and using the learnt clauses obtained during the learning phase using the dataset mentioned in section(4.1)

Table 1: Number of Decisions made by SAT solver for dataset size = 10000

	Without using learnt clause	Using learnt clause
Minimal	4449142	4300697
Efficient	4267712	4202448
Extended	42134	42151

The results clearly indicate that the redundant axioms from the extended encoding yield an enormous decrease in the number of decisions: using the extended encoding the SAT-solver needs only about 1% of the decisions it uses with the minimal or efficient encoding. This clearly shows that if the SAT-solver can learn axioms from the extended encoding, it should be possible to achieve a substantial decrease in the number of decisions.

The below table gives the results concerning the new clauses which are learnt during the training phase. We have categorized the results based on the type of the clause that is learnt which corresponds to the types of clauses explained in section(4.2)

Table 2: Number of clauses learnt in each category for dataset size = 5000

	$Block_d$	Row_d	Col_d	$Cell_d$	$Block_u$	Row_u	Col_u	$Cell_u$	New
Minimal	0	0	0	0	55	23	0	1358	7
Efficient	0	0	0	0	53	25	0	509	3
Extended	0	0	0	0	0	0	0	0	0

A decrease in decisions to the base case was not achieved. It is clearly visible that the SAT-solver was only able to learn clauses belonging to the minimal and efficient encodings. We hypothesize that the reason for this is that the length of axioms of type Row_d, Col_d, and $Block_d$ which characterise the extended encoding is much longer than those from the other encodings: Row_d, Col_d, and $Block_d$ axioms are nine-place disjunctions whereas $Cell_u$, Row_u, Col_u and $Block_u$ are only two-place disjunctions. This makes it much harder for the SAT-solver to find extended encoding axioms.

The most interesting clauses that are learnt are that corresponds to the new column. These clauses do not fall into any of the previous category. Below are some new clauses learnt during the training in $(row, column, number)$ representation:

Valid clauses learnt during training with minimal encoding

$$(9,3,8) \vee (2,3,8) \vee (4,3,8) \vee (8,3,8) \vee \neg(6,3,2) \vee (1,3,8) \vee (3,3,8) \vee (5,3,8) \vee \neg(7,3,9)$$

$$(9,3,8) \vee (2,3,8) \vee (8,3,8) \vee \neg(5,1,8) \vee (1,3,8) \vee (3,3,8) \vee \neg(7,3,9)$$

$$(9,7,4) \vee (8,9,4) \vee \neg(9,8,8) \vee \neg(9,9,1) \vee (8,7,4) \vee (8,8,4) \vee \neg(7,4,4)$$

The learnt clauses are of course very specific and to be able to make some sense of them, some generalization is needed. The first clause simply states that if some numbers x and y are in column j and the number z is missing from column j then z is where x and y are not. We observed a few of number of the clauses.

The second clause can then be interpreted as stating that when you a number x in block k and s.t. x is not in column j but column j is a part of block k, then x is somewhere in column j, excluding the cells that are in block k. It further states that if you have some other number y in column j - also excluding the cells that are in block k - then x is in some other place than y. In short this is stating that when a number is placed in some block then a column which is a part of the block, that number in that column needs to be from a different block as well as the fact that if a cell is filled in that column, the number cannot be in that cell.

The third clause is similar to the second clause but rather states that when a number is placed in a column which is a part of some block, the number in that block needs to be from a different column. We also found clauses which were like the second clause but exchanged row for column.

7 Conclusion

In our project, we hypothesized it would be possible to train SAT-solvers, Minisat, in particular, to become more efficient in the current domain by extending the encoding to include valid learnt clauses. We measured the number decisions made by the SAT-solver to measure success. To do this we used the collection of minimum proper sudokus.

We recorded learnt clauses reported by the SAT-solver and checked if they were valid for all sudokus. Due to the high number of learnt clauses and the fact that for each learnt clause we had to solve a SAT problem it would have been unfeasible to process all learnt clauses this way so we devised ways of pruning the learnt clauses. The new axioms were then classified.

The efficient and minimal encoding make roughly the same number of decisions. The extended encoding makes about 1

During this process, some new axioms were found which were not in the extended encoding but might prove useful if generalized and added to the extended encoding. Future research could measure the effectiveness of these new axioms on the number of decisions made. It would also be worth looking into if a SAT-solver could be optimized to learn valid clauses when working on a closed domain of problems.

References

Demoen, B., & de la Banda, M. G. (2014). Redundant sudoku rules. *Theory and Practice of Logic Programming*, *14*(3), 363–377.

Kwon, G., & Jain, H. (2006). Optimized cnf encoding for sudoku puzzles. In *Proc. 13th international conference on logic for programming artificial intelligence and reasoning (lpar2006)* (pp. 1–5).

Lynce, I., & Ouaknine, J. (2006). Sudoku as a sat problem. In *Isaim.*

Minimal sudoku dataset. (n.d.). `http://staffhome.ecm.uwa.edu.au/ 00013890/sudoku17`. (Accessed: 2017-09-25)

Simonis, H. (2005). Sudoku as a constraint problem. *Proc. 4th Int. Works. Modelling and Reformulating Constraint Satisfaction Problems*, 13–27.