A SAT-based Sudoku Solver

Tjark Weber

Institut für Informatik, Technische Universität München
Boltzmannstr. 3, D-85748 Garching b. München, Germany
webertj@in.tum.de

Abstract. This paper presents a SAT-based Sudoku solver. A Sudoku is translated into a propositional formula that is satisfiable if and only if the Sudoku has a solution. A standard SAT solver can then be applied, and a solution for the Sudoku can be read off from the satisfying assignment returned by the SAT solver. No coding was necessary to implement this solver: The translation into propositional logic is provided by a framework for finite model generation available in the Isabelle/HOL theorem prover. Only the constraints on a Sudoku solution had to be specified in the prover’s logic.

1 Introduction

Sudoku, also known as Number Place in the United States, is a placement puzzle. Given a grid – most frequently a $9 \times 9$ grid made up of $3 \times 3$ subgrids called “regions” – with various digits given in some cells (the “givens”), the aim is to enter a digit from 1 through 9 in each cell of the grid so that each row, column and region contains only one instance of each digit. Fig. 1 shows a Sudoku on the left, along with its unique solution on the right [12]. Note that other symbols (e.g. letters, icons) could be used instead of digits, as their arithmetic properties are irrelevant in the context of Sudoku. This is currently a rather popular puzzle that is featured in a number of newspapers and puzzle magazines [1, 3, 9].

Several Sudoku solvers are available already [6, 10]. Since there are more than $6 \cdot 10^{21}$ possible Sudoku grids [5], a naïve backtracking algorithm would be infeasible. Sudoku solvers therefore combine backtracking with – sometimes complicated – methods for constraint propagation. In this paper we propose a SAT-based approach: A Sudoku is translated into a propositional formula that is satisfiable if and only if the Sudoku has a solution. The propositional formula is then presented to a standard SAT solver, and if the SAT solver finds a satisfying assignment, this assignment can readily be transformed into a solution for the original Sudoku. The presented translation into SAT is simple, and requires minimal implementation effort since we can make use of an existing framework for finite model generation [11] available in the Isabelle/HOL [8] theorem prover.

* This work was supported by the PhD program Logic in Computer Science of the German Research Foundation.
Fig. 1. Sudoku example and solution

2 Implementation in Isabelle/HOL

An implementation of the Sudoku rules in the interactive theorem prover Isabelle/HOL is straightforward. Digits are modelled by a datatype with nine elements 1, ..., 9. We say that nine grid cells \(x_1, \ldots, x_9\) are valid if they contain every digit.

**Definition 1 (valid).**

\[
\text{valid}(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9) \equiv \bigwedge_{d=1}^{9} \exists i = 1 \ldots 9. x_i = d.
\]

Labeling the 81 cells of a 9 × 9 grid as shown in Fig. 2, we can now define what it means for them to be a Sudoku solution: each row, column and region must be valid.

**Definition 2 (sudoku).**

\[
\text{sudoku}(x_{ij})_{i,j \in \{1, \ldots, 9\}} \equiv \bigwedge_{i=1}^{9} \text{valid}(x_{i1}, x_{i2}, x_{i3}, x_{i4}, x_{i5}, x_{i6}, x_{i7}, x_{i8}, x_{i9})
\]

\[
\wedge \bigwedge_{j=1}^{9} \text{valid}(x_{1j}, x_{2j}, x_{3j}, x_{4j}, x_{5j}, x_{6j}, x_{7j}, x_{8j}, x_{9j})
\]

\[
\bigwedge_{i,j \in \{1,4,7\}} \text{valid}(x_{ij}, x_{i(j+1)}, x_{i(j+2)}, x_{(i+1)j}, x_{(i+1)(j+1)}, x_{(i+1)(j+2)})
\]

\[
x_{(i+2)j}, x_{(i+2)(j+1)}, x_{(i+2)(j+2)}
\]

The next section describes the translation of these definitions into propositional logic.

3 Translation to SAT

We encode a Sudoku by introducing 9 Boolean variables for each cell of the 9 × 9 grid, i.e. \(9^3 = 729\) variables in total. Each Boolean variable \(p_{ij}^d\) (with
1 ≤ i, j, d ≤ 9) represents the truth value of the equation \(x_{ij} = d\). A clause
\[
\bigvee_{d=1}^{9} p^d_{ij}
\]
ensures that the cell \(x_{ij}\) denotes one of the nine digits, and 36 clauses
\[
\bigwedge_{1 ≤ d < d' ≤ 9} \neg p^d_{ij} \lor \neg p^{d'}_{ij}
\]
make sure that the cell does not denote two different digits at the same time.

Since there are just as many digits as cells in each row, column, and region, Def. 1 is equivalent to the following characterization of validity, stating that the nine grid cells \(x_1, \ldots, x_9\) contain distinct values.

**Lemma 1 (Equivalent characterization of validity).**

\[
\text{valid}(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9) \iff \bigwedge_{1 ≤ i < j ≤ 9} x_i \neq x_j
\]
\[
\iff \bigwedge_{1 ≤ i < j ≤ 9} \bigwedge_{d=1}^{9} x_i \neq d \lor x_j \neq d.
\]

The latter characterization turns out to be much more efficient when translated to SAT. While Def. 1, when translated directly, produces 9 clauses with 9 literals each (one literal for each equation), the formula given in Lemma 1 is translated into 324 clauses (9 clauses for each of the 36 inequalities \(x_i \neq x_j\)), but each clause of length 2 only. This allows for more unit propagation [14] at the Boolean level, which – in terms of the original Sudoku – corresponds to cross-hatching [12] of digits, a technique that is essential to reduce the search space.
The 9 clauses obtained from a direct translation of Def. 1 could still be used as well: unit propagation on these clauses would correspond to counting the digits 1 – 9 in regions, rows, and columns to identify missing numbers. However, in our experiments we did not experience any speedup by including these clauses.

This gives us a total of 11745 clauses: 81 definedness clauses of length 9, 81·36 uniqueness clauses of length 2, and 27·324 validity clauses,\(^1\) again of length 2. However, we do not need to introduce Boolean variables for cells whose value is given in the original Sudoku, and we can omit definedness and uniqueness clauses for these cells as well as some of the validity clauses – therefore the total number of variables and clauses used in the encoding of a Sudoku with givens will be less than 729 and 11745, respectively.

Note that our encoding already yields a propositional formula in conjunctive normal form (CNF). Therefore conversion into DIMACS CNF format [4] – the standard input format used by most SAT solvers – is trivial. Isabelle can search for a satisfying assignment using either an internal DPLL-based [2] SAT solver, or write the formula to a file in DIMACS format and execute an external solver. We have employed zChaff [7] to find the solution to various Sudoku classified as “hard” by their respective authors (see Fig. 3 for an example), and in every case the runtime was only a few milliseconds.

\[\begin{array}{|c|c|c|} \hline & & \\ \hline 2 & 1 & \ \ \\ \hline & 6 & 3 \\ \hline 7 & 5 & 1 \\ \hline \end{array}\] \hspace{1cm} \[\begin{array}{|c|c|c|} \hline & & \\ \hline 1 & 2 & 6 \\ \hline 4 & 8 & 5 \\ \hline 9 & 3 & 7 \\ \hline \end{array}\]  

Fig. 3. hard Sudoku example and solution

4 Concluding Remarks

We have presented a straightforward translation of a Sudoku into a propositional formula. The translation can easily be generalized from 9 × 9 grids to grids of arbitrary dimension. It is polynomial in the size of the grid, and since Sudoku is NP-complete [13], no algorithm with better complexity is known. The translation, combined with a state-of-the-art SAT solver, is also practically successful: 9 × 9 Sudoku puzzles are solved within milliseconds.

Traditionally the givens in a Sudoku are chosen so that the puzzle’s solution is unique; nevertheless our algorithm can be extended to enumerate all possi-

\(^1\) This number includes some duplicates, caused by the overlap between rows/columns and regions: certain cells that must be distinct because they belong to the same row (or column) must also be distinct because they belong to the same region.
ble solutions (by explicitly disallowing all solutions found so far, and perhaps using an incremental SAT solver that allows adding clauses on-the-fly to avoid searching through the same search space multiple times).

Particularly remarkable is the fact that our solver, while it can certainly compete with hand-crafted Sudoku solvers, some of which use rather complex patterns and search heuristics, required very little implementation effort. Aside from Lemma 1, no domain-specific knowledge was used. The impressive performance is largely due to the SAT solver. Even the translation into propositional logic was not written by hand, but is an instance of a framework for finite model generation that is readily available in the Isabelle/HOL theorem prover. Only the Sudoku rules had to be defined in the prover, and this was a trouble-free task.

References