Solving Sudoku via SAT

Bart Massey
12 April 2016

In this document, we describe the construction of a Sudoku solver that operates in three phases: construction of a prop CNF SAT instance from the Sudoku instance, use of a SAT solver to solve the SAT instance, and extracting the solution to the Sudoku instance from the SAT solution. The constructor and extractor are written in Python, and off-the-shelf SAT solvers are used. Runtimes are very fast independent of problem "difficulty", on the order of 150 milliseconds on the author's desktop box.

1 Solving Sudoku

This section describes the properties of a Sudoku solution. It is written using the Z specification notation for precision and clarity. It is assumed that the reader is at least familiar with the Sudoku grid.

We begin by describing some of the sizes and dimensions relevant to the problem.

$$
\begin{aligned}
&\mathit{SQUARE} == 3\\
&\mathit{SIDE} == \mathit{SQUARE} * \mathit{SQUARE}\\
&\mathit{ROW} == 1\mathrel{..}\mathit{SIDE}\\
&\mathit{COL} == 1\mathrel{..}\mathit{SIDE}\\
&\mathit{VALUE} == 1\mathrel{..}\mathit{SIDE}
\end{aligned}
$$

An important concept in Sudoku is the "sub-board" or "group". It is important to know what coordinate group a given coordinate is in.

$$
\begin{array}{|l}
\mathit{group} : 1\mathrel{..}\mathit{SIDE} \rightarrow 1\mathrel{..}\mathit{SQUARE}\\
\hline
\forall x : 1\mathrel{..}\mathit{SIDE} \bullet\\
\qquad \mathit{group}(x) = (x - 1) \operatorname{div} \mathit{SQUARE} + 1
\end{array}
$$

There are basically three constraints on a partial Sudoku solution: the same value cannot appear twice within a row; the same value cannot appear twice within a column; the same value cannot appear twice within a sub-board.

Sudoku

$$board : ROW \times COL \mathrel{+\mkern-14mu\rightarrow} VALUE$$

$$
\begin{array}{l}
\forall\, r : ROW;\ c_1, c_2 : COL \mid \\
\quad c_1 \neq c_2 \land \\
\quad (r, c_1) \in \operatorname{dom}(board) \land (r, c_2) \in \operatorname{dom}(board) \bullet \\
\qquad board(r, c_1) \neq board(r, c_2) \\[1ex]
\forall\, r_1, r_2 : ROW;\ c : COL \mid \\
\quad r_1 \neq r_2 \land \\
\quad (r_1, c) \in \operatorname{dom}(board) \land (r_2, c) \in \operatorname{dom}(board) \bullet \\
\qquad board(r_1, c) \neq board(r_2, c) \\[1ex]
\forall\, r_1, r_2 : ROW;\ c_1, c_2 : COL \mid \\
\quad (r_1, c_1) \neq (r_2, c_2) \land \\
\quad (r_1, c_1) \in \operatorname{dom}(board) \land (r_2, c_2) \in \operatorname{dom}(board) \land \\
\quad group(r_1) = group(r_2) \land group(c_1) = group(c_2) \bullet \\
\qquad board(r_1, c_1) \neq board(r_2, c_2)
\end{array}
$$

A solution to a Sudoku instance is a total assignment of values to squares of the instance that respects the initial partial assignment and the legality constraints.

SolveSudoku

$$
\begin{array}{l}
problem? : Sudoku \\
solution! : Sudoku
\end{array}
$$

$$
\begin{array}{l}
problem?.board \subseteq solution!.board \\
solution!.board \in ROW \times COL \rightarrow VALUE
\end{array}
$$

2 Prop. CNF SAT Instance Extraction

The logical description of the previous section greatly facilitates reducing a Sudoku instance to a Prop. CNF SAT instance whose solution gives a solution to the Sudoku instance.

We will represent the board relation using atoms of the form B_{rcv} which will be interpreted as true iff the board at row r and column c has value v. There are $9^3 = 729$ such atoms. We will establish a series of constraints, extracted from the Z specification, that together ensure that any satisfying assignment to the Bs will be interpretable as a solution to the Sudoku instance.

1. We require that the solution match the given Sudoku instance. To do this, we emit unary clauses of the form
$$B_{rcv}$$
for each r, c and v specified in the Sudoku instance description.
2. We require that the *board* relation is total. To do this, we emit clauses of the form
$$(B_{rc1} \lor B_{rc2} \lor \ldots \lor B_{rc9})$$
for every r and c. Note that these are the only clauses in the description of arity greater than two.
3. We require that the *board* relation is a function. To do this, we emit clauses of the form
$$(\neg\, B_{rcv_1} \lor \neg\, B_{rcv_2})$$
for every r, c, v_1 and v_2 such that $v_1 \neq v_2$.

4. We require that no row contains the same value in two different columns. To do this, we emit clauses of the form
$$(\neg B_{rc_1v} \vee \neg B_{rc_2v})$$
for every r, c_1, c_2 and v such that $c_1 \neq c_2$.

5. We require that no column contains the same value in two different rows. To do this, we emit clauses of the form
$$(\neg B_{r_1cv} \vee \neg B_{r_2cv})$$
for every r_1, r_2, c and v such that $r_1 \neq r_2$.

6. We require that the same value not appear in two different positions in the same sub-board. To do this, we emit clauses of the form
$$(\neg B_{r_1c_1v} \vee \neg B_{r_2c_2v})$$
for every r_1, r_2, c_1, c_2 and v such that $(r_1, c_1) \neq (r_2, c_2)$ but both coordinates are in the same sub-board.

3 Decoding The Answer

Once a satsifying assignment is found for the problem of the previous section, it remains to turn this assignment back into a solved Sudoku board. To do so, we note that, since *board* is a total function, for each row r and column c there will be exactly one value v for which B_{rcv} is true: this v is the value which should be filled in at position (r, c) in the board.

4 Implementation and Evaluation

All of this is implemented using two Python programs, one to encode and one to extract the answer, together with an off-the-shelf SAT solver.

The two SAT solvers I have tried are the readily-available open-source solvers Picosat and Minisat2. These solver both require the same input format: the famous DIMACS format. In this format, one row of text corresponds to one clause. Each atom is given a number starting at 1. Positive literals are denoted by atom number, negative literals by its integer negation. The DIMACS output format is similar.

The running system has been tested on several Sudoku instances, including "the hardest Sudoku" and another "very hard" instance. End-to-end run times are uniformly in the 200 millisecond range. The program has also been tested on a blank Sudoku board, generating a filled grid, and on an unsolvable Sudoku instance, which it correctly repots as unsolvable.