SAT Solving Sudoku

Aresh Pourkavoos

April 16, 2022

Sudoku is a puzzle game which consists of a 9×9 square grid, subdivided into 3×3 blocks and partially filled with the digits 1-9. The goal is to fill in the rest of the digits such that every row, column and block contains each digit exactly once. Sudoku is a well-studied problem, and there are many algorithms out there to solve it, from pencil-and-paper tricks used mostly by human players to trial-and-error methods used by computers, such as backtracking. However, I will focus on one particular approach: SAT solving.

SAT is short for "satisfiability," and a SAT solver is a program that checks whether a given statement is satisfiable. In this case, a statement is a logical proposition such as "P and Q", and a statement is satisfiable if there is some way to assign each variable (P and Q, in this case) to either true or false such that the entire statement is true. "P and Q," for example, is satisfiable: it has exactly one solution, where P and Q are both true. "P and not P," on the other hand, is not satisfiable, since for both possible values of P, the statement is false. A brute-force SAT solver would simply check every possible combination, of which there are 2^n, where n is the number of variables. However, there are much more optimized algorithms in use, making SAT solvers practical for huge formulas with thousands of variables or more.

This article will consist of two main sections: translating Sudoku into a form a SAT solver can recognize, and how the solver actually finds the answer. However, it's not immediately obvious how Sudoku could be turned into a SAT problem, since each cell is filled with one of 9 possible digits, not one of 2 truth values. We can represent a choice of 9 by having 9 variables for each cell, one for each possible digit, but this creates another problem. There are 512 possible assignments to these variables, which include all of them being false, i.e. leaving a cell empty, and more than one being true, i.e. placing two or more digits in the same cell. We need to constrain these variables (in typical SAT solver fashion) to ensure that exactly one variable is true. There are two parts to this problem: forbidding empty cells and preventing two digits in the same cell. The former can be expressed as an or statment containing the given variables:

$$p_1 \lor p_2 \lor \ldots \lor p_n$$

The latter looks at all pairs of variables, saying that they can't both be true:

$$\neg(p_1 \land p_2) \land \ldots \land \neg(p_1 \land p_n) \land \neg(p_2 \land p_3) \land \ldots \land \neg(p_2 \land p_n) \land \ldots \land \neg(p_{n-1} \land p_n)$$

For n variables, there are $\binom{n}{2} = \frac{n(n-1)}{2}$ pairs. For a given Sudoku cell, where there are 9 variables, it takes $\binom{9}{2}$ of them to say that at most one variable may be true, and the or clause containing all 9 variables makes 37. Since there are 81 cells, it takes $37 \times 81 = 2997$ formulas just to say that every cell is filled with exactly one digit. To express the rest of the puzzle, similar constraints must be applied to other sets of 9 variables:

- Every row of cells contains every digit exactly once, i.e. for every row and digit, exactly 1 of the 9 variables that represent filling a cell within the given row with a given digit must be true.
- Every column contains every digit exactly once, i.e. for every column and digit, exactly 1 of the 9 variables that represent filling a cell within the given column with a given digit must be true.
- Every 3×3 block contains every digit exactly once, i.e. for every block and digit, exactly 1 of the 9 variables that represent filling a cell within the given block with a given digit must be true.

Each of these 3 items places constraints on 81 sets of 9 variables each, just like the requirement that every cell contains exactly one digit. Thus each one adds another 2997 formulas for a total of $2997 \times 4 = 11988$.

Strictly speaking, these variables and formulas are all that are necessary to define the constraints of a Sudoku puzzle formatted in the way described. But is it the only way, or even the best way? In SAT solving, a trick that is often used to speed up computation at the cost of increasing memory use is to introduce *auxiliary variables*, which are defined to be equivalent to some expression of the original input. Among other things, their efficiency can come from reducing the number of formulas, as we will do with Sudoku.

At the moment, each set of 9 variables requires 37 formulas to say that exactly one of them must be true. We will lower that number to 25 by introducing 3 auxiliary variables to each set, each representing the fact that one of a set of 3 of the 9 is true. In other words, given p_1 through p_9, we define

$$\begin{gathered} q_1 \leftrightarrow p_1 \vee p_2 \vee p_3, \\ q_2 \leftrightarrow p_4 \vee p_5 \vee p_6, \\ q_3 \leftrightarrow p_7 \vee p_8 \vee p_9. \end{gathered}$$

However, $\leftrightarrow$ is not an operation recognized by most SAT solvers. Instead, the biimplication must be broken up:

$$\begin{aligned} q_1 &\rightarrow (p_1 \vee p_2 \vee p_3), \\ (p_1 \vee p_2 \vee p_3) &\rightarrow q_1. \end{aligned}$$

In fact, the logical statements that SAT solvers accept are in a much more restrictive format known as conjunctive normal form, or CNF. A CNF statement is a formula consisting of clauses joined by $\wedge$, where each clause consists of terms joined by $\vee$, where each term is either a variable or its negation. For example,

$$(p_1 \vee \neg p_2) \wedge (p_2 \vee p_3)$$

is in CNF, but

$$\neg(p_1 \vee \neg p_2) \wedge (p_2 \vee p_3)$$

is not, since CNF does not allow the clause $(p_1 \vee p_2)$ to be negated. However, it is possible to turn this into CNF by using De Morgan's laws:

$$\neg p_1 \wedge p_2 \wedge (p_2 \vee p_3)$$

Now there are three clauses to satisfy: $\neg p_1$, p_2, and $p_2 \vee p_1$.

We can turn the first implication into CNF by using the equivalence

$$(a \rightarrow b) \cong (\neg a \vee b),$$

turning it into

$$\neg q_1 \vee p_1 \vee p_2 \vee p_3.$$

Applying this to the reverse implication, on the other hand, returns

$$\neg(p_1 \vee p_2 \vee p_3) \vee q_1,$$

which is still not in CNF. De Morgan's laws yield

$$(\neg p_1 \wedge \neg p_2 \wedge \neg p_3) \vee q_1,$$

and the inner expression may be distributed:

$$(\neg p_1 \vee q_1) \wedge (\neg p_2 \vee q_1) \wedge (\neg p_3 \vee q_1).$$

This is in CNF, so it may be fed to the solver. Note that this expression is also equivalent to

$$(p_1 \rightarrow q_1) \wedge (p_1 \rightarrow q_1) \wedge (p_1 \rightarrow q_1),$$

which makes sense intuitively: any of p_1, p_2, and p_3 is enough to imply q_1 on its own.

From there, the fact that at least one p is true may be written as $q_1 \vee q_2 \vee q_3$, which is still just one formula as before. The savings come from the statement that at most one p is true, which is broken into two main parts. The first of these is that at most one q can be true, since otherwise, there would be two ps "contained in" different qs which are both true. Thus

$$\begin{aligned} &\neg(q_1 \wedge q_2), \\ &\neg(q_1 \wedge q_3), \\ &\neg(q_2 \wedge q_3). \end{aligned}$$

The second is that "within" each q, at most one p is true. For example, for q_2, the statmements would be

$$\begin{aligned} &\neg(p_4 \wedge p_5), \\ &\neg(p_4 \wedge p_6), \\ &\neg(p_5 \wedge p_7). \end{aligned}$$

Similar statements may be written for p_1 through p_3 and for p_7 through p_9, bringing the total to 12.

$$\begin{array}{rrr}
\neg p_1 \vee \neg p_2 & \neg p_4 \vee \neg p_5 & \neg p_7 \vee \neg p_8 \\
\neg p_1 \vee \neg p_3 & \neg p_4 \vee \neg p_6 & \neg p_7 \vee \neg p_9 \\
\neg p_2 \vee \neg p_3 & \neg p_5 \vee \neg p_6 & \neg p_8 \vee \neg p_9 \\
\neg q_1 \vee p_1 \vee p_2 \vee p_3 & \neg q_2 \vee p_4 \vee p_5 \vee p_6 & \neg q_3 \vee p_7 \vee p_8 \vee p_9 \\
\neg p_1 \vee q_1 & \neg p_4 \vee q_2 & \neg p_7 \vee q_3 \\
\neg p_2 \vee q_1 & \neg p_5 \vee q_2 & \neg p_8 \vee q_3 \\
\neg p_3 \vee q_1 & \neg p_6 \vee q_2 & \neg p_9 \vee q_3 \\
 & & \neg q_1 \vee \neg q_2 \\
 & & \neg q_1 \vee \neg q_3 \\
 & & \neg q_2 \vee \neg q_3 \\
 & & q_1 \vee q_2 \vee q_3
\end{array}$$

In this representation, there are not only fewer clauses to check but also a Sudoku solving trick that we get for free: the technique of intersection removal. Unit propagation is able to reason with this and proceed further toward a solution before having to guess than before, which is great because backtracking is expensive.