Solve Sudoku

How do you solve – or create – a Sudoku puzzle using programmer's logic?

Sudoku puzzles. If you've ever spent any time tackling them, I'm sure you've come up with your own way of solving them. But how does that translate to a computer program? Is it worth translating your hard-won algorithm directly to your favourite language, or are there better ways of solving a Sudoku puzzle? And having done that, how do you go about creating a new one? Even better, how do you rank them by difficulty without having to solve them by hand?

The first time I came across Sudoku was in a letter from my parents. They sent us some newspaper clippings of some weird square grids partially filled in with numbers from *The Telegraph*. For some reason, the UK caught the Sudoku bug before we did in Colorado, and these were the first examples I'd seen or even heard of. We tried the puzzles out (carefully photocopying them first, so we could both have a go) and found that we liked them. Eventually, a few months later, even our local paper started carrying them. I found examples and programs on the web, and the rest is history.

A brute-force technique

Once I'd perfected my own 'mental' algorithm (my set of solving techniques, in other words), I started wondering how to write a program to solve a Sudoku grid. Like many before me, I started off with the brute-force algorithm through a depth-first technique.

The standard Sudoku puzzle is a grid of 9 x 9 squares (or 'cells'), with some of the cells containing a digit, and most being empty. Furthermore, the grid is subdivided into nine 3 x 3 boxes. Your task is to fill the empty cells with the digits from 1 to 9 such that each row, column and box contains exactly one of each digit. Figure 1 shows a typical puzzle (this was classified as 'hard' when I generated it, in case you want to play). ▸

4		9		7				
					2			
		8		9	6			
	5	6		2				4
		1				7		
9				3		8	6	
			7	4		3		
			9					
				1		6		5

▲ **Figure 1: A typical hard Sudoku puzzle.**

The brute-force technique goes something like this. Find the first empty cell (usually by starting at the top-left corner). Put a '1' in there. Check to see whether that digit violates one of the Sudoku rules (that is, check to see whether there's already a 1 in the same row, column or box). If it's valid, find the next empty cell and repeat the algorithm. If there's a clash, increment the digit you just added and do the check again. Repeat this inner loop until either you find a valid digit for that cell – in which case move on to the next empty cell – or you find that you've run out of digits to test.

In the latter case, you must backtrack to the previous cell you filled in and increment the digit you find there, check and then move forward again. Sometimes you'll find that you backtrack several cells in a cascade. If the puzzle has a solution, you'll be sure to find it using this method.

Figure 2 shows the initial steps in a brute-force algorithm to solve the puzzle in Figure 1. I've indicated the choices made in these first steps by crossing digits out. Note that by the time you reach the top-right cell, there are no digits left that can go in there, so you must backtrack.

This type of algorithm is known as a 'depth-first search' algorithm. In setting a guess for the cell and moving onto the next cell, we're trying to move ahead as fast as possible (increasing the depth we travel from the first cell), and we only backtrack when necessary. It's as if the grid were a maze, and we follow each path until we can't advance any more, whereupon we go back to a place where we can try the next path.

Depth-first algorithms are generally solved using a stack. You push some state onto the stack when you make a choice, and then, in order to backtrack, all you need to do is to pop off the previous state. In the Sudoku-solving algorithm, the state is pretty simple: it's just the 'address' of the cell that you've just modified.

For the cell address, the simplest method is to use a number from 0 to 80, counting from the top-left cell along each row and working your way down the grid. So the first row will have cells 0 to 8, the second row 9 to 17, and so on. This addressing scheme has another benefit: given a particular cell address, it's simple arithmetic to work out the other cells in the same row and in the same column. (For the row, divide the cell address by 9, dropping the remainder, then multiply by 9 to get the address of the first cell in the row. Other cells are then found by adding 1 each time. For the column, divide by 9 and the remainder is the address of the top cell in that column. Other cells are then found by adding 9 each time.) The box cells are a little more difficult, and it's better to use a lookup table to find the top-left cell in the box. Other cells can then be found by setting up and using other tables (for instance, the top-left box contains cells 0, 1, 2, 9, 10, 11, 18, 19 and 20).

Using this addressing scheme, the algorithm goes like this. First, we set up an array of 81 cells to either contain 0 (these cells are empty) or the given digit from the original puzzle. Then we:

1. Find the next empty cell.
2. If there are no more empty cells, we're done.
3. If there is one, set this cell to 1.
4. Check that the row, column, and box for this cell do not duplicate the number in the current cell.
5. If the test passes, push the cell address onto the stack and return to step 1.
6. If we have a duplicate, increment the value in the current cell.
7. If the value in the cell is 9 or less, then go to step 4.
8. Set the value in the cell to 0, pop off the previous cell address (this will be the current cell) and go to step 6.

The algorithm will exhaustively search the solution space for the grid.

The problem with the algorithm is the time taken. In general, it's very fast – almost instantaneous – but it's fairly easy to construct a pathological Sudoku puzzle grid that will cause the algorithm to grind through more backtracking than is strictly necessary and thereby take a lot of time. As an example, one could construct a grid where the first row is empty and the solution is 987654321: the brute-force algorithm as described would be backtracking like crazy.

To counter these pathological examples to a certain extent (they're essentially created by hand to show off the problems with brute-force solving), you can change the way you select the next empty cell. Instead of routinely looking for the next empty cell sequentially, you could use an algorithm to select the next cell at random. The easiest way of doing this in practice is to create an array of all the empty cells at the beginning, shuffle this array and then use the elements in sequential order of the shuffled array.

Of course, this algorithm assumes that the puzzle has a solution. What if it doesn't? The problem will be that our stack of previously visited 'empty' cells will become exhausted and we won't be able to pop an address off it. We can either check for this condition explicitly or we can push a 'special' cell address onto the stack right at the beginning (say, cell address -1) and if we pop it we know there's no solution. In either event, we've covered all eventualities.

Generating puzzles

Now that you have a working Sudoku puzzle solver, it's time to consider the problem of generating your own Sudoku puzzles. If you look in the newspapers and at the Sudoku

Spotlight on… Exact cover

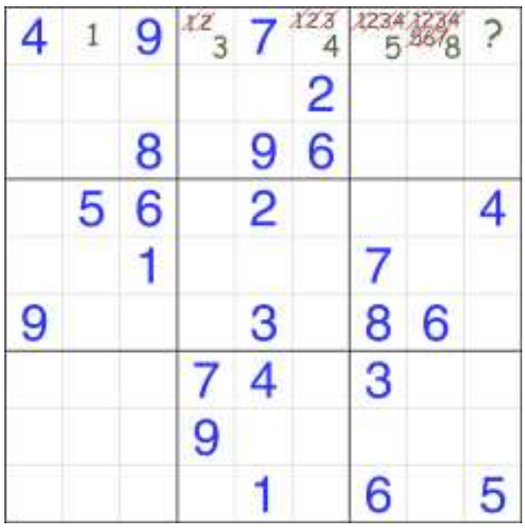
Suppose you have a set of items and a collection of subsets of that set. An exact cover is a group of subsets from that collection, such that every item from the original set can be found in only one subset in that group. The exact cover problem is how to determine if such an exact cover exists and what it is.

The easiest way to visualise this is imagining a matrix of 1s and 0s. Your task is to find a set of rows from this matrix such that there is exactly one 1 in each column. (Figure 3 shows a small example.) The set of rows you find is known as an 'exact cover'. Finding the exact cover is extremely hard to solve in general, and the running time is exponential.

However, if the matrix is sparse enough, it becomes possible to solve for the exact cover in a reasonable time using an algorithm invented by Donald Knuth called Dancing Links. Solving a Sudoku puzzle is such a problem, and Knuth showed how you could convert a Sudoku puzzle into a 729 x 324 sparse matrix and then solve it for the exact cover (the solution to the original puzzle). ■

Diabolical puzzles

Once computer programmers started writing programs to create Sudoku puzzles, it was only a matter of time before they investigated how hard they could make the puzzles. Soon a category of puzzles emerged that were deemed too hard (or required logic that was too complex) for human solvers. All of these puzzles required, at some point, the solver to resort to trial-and-error in order to finish the puzzle – in effect to guess and to backtrack by hand, possibly several times. Needless to say, these kinds of puzzles would take an hour or more to solve manually. ■



▲ **Figure 2: Initial steps to solve the hard puzzle by brute force.**

collections in book form, you'll see that they tend to have mirror symmetry about the centre axes. For ease of describing an algorithm, I decided to just use 180-degree rotational symmetry about the centre. Here's how to generate a new puzzle using this definition:

1. Select, at random, a number from 0 to 40 (cell 40 is the centre cell). Call this cell x.
2. Select at random a digit from 1 to 9 and put it in cell x.
3. Providing that x is not 40, generate another random digit and put it in cell (80 - x). Because of the way we set up the cell addressing system, you'll find that that cell is rotationally opposite the first.
4. Try and solve the puzzle with the solver.

Multiple solutions

Unfortunately, there's a big problem with this algorithm. Our solver only tests for two outcomes: no solution or a solution. It doesn't check to see whether a puzzle actually has more than one solution. As you can imagine, therefore, merely putting two digits into an empty grid is going to produce a puzzle with many solutions.

As such, we need to run our solver in a special mode when creating a puzzle. It must not only report that the puzzle has no solutions or one solution, but it must also attempt to see if the puzzle has more than one solution to it. Note that we're really not interested in *how many* solutions a puzzle has in the latter case; we just need to see whether we can find a second one.

To do this, all we need to do is to continue the brute-force search after we find the first solution. That is, once we find (and signal) the first solution, we continue running the solver as if that solution didn't exist.

Once you have this tweaked solver, the algorithm to create a Sudoku puzzle works like this:

1. Select a cell from 0 to 40 (x). If it's empty, set it to a random digit. Set the opposite cell (80 - x) to another random digit. (Again, if x is 40, there's no opposite cell.)
2. Try and solve the puzzle.
3. If the puzzle has no solution, erase the grid and go back to step 1.
4. If the puzzle has more than one solution, go back to step 1.
5. You have a Sudoku puzzle with a unique solution, so print it.

Now, I'll admit step 3 is possibly overkill. However, it's much, much easier to implement this than to, say, scrub out the last two cells you randomly set and try again, since I would guess there's a possibility of having to backtrack a couple of times.

Calculating difficulty

Now that you have a working Sudoku solver (that is, an implementation that, given a puzzle, will work out the solution to that puzzle) and a way to create new puzzles, the next problem is to mechanically decide on the difficulty of a new puzzle.

As far as I can determine by looking at several newspapers and books, puzzles are categorised as easy, medium, hard and diabolical, with 'diabolical' being reserved for those puzzles where the human solver has to make a guess at some point and either solve the puzzle with that guess, or fail and make another guess at the same point (in other words, solving by trial and error).

	1	2	3	4	5
1	1	1	1	0	0
2	1	0	0	1	0
3	0	1	0	0	0
4	1	0	1	1	0
5	0	0	1	1	0
6	0	0	1	0	1

▲ **Figure 3: A simple exact cover problem: rows 2, 3 and 6 cover the columns exactly.**

As it happens, in all my research I haven't managed to find a really good algorithm for categorising the difficulty of Sudoku puzzles. Equally, I would point out that human categorising of Sudoku puzzles is not a well-defined art either: some puzzles marked as medium I've found easy, and some I've found to be hard. It all depends on the cataloguer's own collection of solving techniques and how yours differs from theirs.

There's a program available called Sudoku Explainer (**www.bit.ly/SSQLs**) that solves Sudoku puzzles by reasoning as a human solver would do. The program has implementations of various techniques and will explain how it solves a particular game step by step. It will attempt to apply the techniques it uses in order of complexity, and in doing so will calculate the difficulty of the puzzle according to the highest level of technique it has to use.

Given our brute-force algorithm, how could we determine difficulty? Here's one possible (and simple) algorithm: let the randomised brute-force solver attack the same puzzle in several separate runs. Since the randomiser would calculate a different path through the empty cells, it would solve the puzzle in a different way each time. For each run, you would count the number of backtrackings, so that at the end you can calculate the average.

Although the brute-force algorithm in no way mimics the way a human solver would solve a Sudoku puzzle, there might be some correlation between the number of backtrackings and the difficulty level as set by a human puzzle compiler. I will also note that the number of backtrackings would seem to be correlated with the number of filled cells in the original puzzle: the fewer the number of filled cells, the harder, in theory, the puzzle. ■

Julian M Bucknall has worked for companies ranging from TurboPower to Microsoft and is now CTO for Developer Express.
feedback@pcplus.co.uk

Sudoku variants

Given the popularity of Sudoku, several variants have been devised. The one that looks most like Sudoku itself is known as Killer Sudoku. It has the same 9 x 9 grid divided into boxes, but now cells are divided into 'cages' (a region of contiguous cells) with the sum of the cells in the cage printed within it. Generally, no cells are pre-filled. The solver has to use arithmetic as well as standard Sudoku techniques in order to solve the puzzle. Another variant is Kakuro, which looks a bit like a crossword. In this case, the clues are sums of the digits in the cells for the entries. The answers to the clues cannot contain duplicated digits. Again, the solver has to use arithmetic as well as logic to solve the puzzle. ■

The history of Sudoku

The very earliest grid number puzzles (involving the removal of numbers from magic squares) appear to have been created by French compilers in the late 19th century, but the more modern Sudoku was probably invented (although not given that name) by Howard Garns in 1979. These early puzzles were introduced into Japan in 1984 and given a ridiculously long name which was eventually shortened to Sudoku (or SuDoKu). These puzzles were introduced around the world in 2005, and from these beginnings they became the huge global hit they are today. ■