A Sudoku Program in Fortran 95

Michael Metcalf
420 East 61st. Street, Apt. 38B
New York, NY 10021, USA
michaelmetcalf@compuserve.com

The craze

A remarkable craze swept though many countries in the year 2005. Already known and popular in Japan, sudoku was introduced into the UK and thence into other European countries and the USA. Its rules are simple. Given a 9 x 9 grid with some numbers already in place:

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

the solver has to fill in the missing values such that, in each row, each column and each 3 x 3 box, the digits 1 to 9 appear once and once only. The starting digits in this example are arranged symmetrically, which is merely a matter of taste. However, it is widely accepted that a properly formed puzzle has a unique solution and that it can thus be solved, without having to make any guesses, by pure logic.

An excellent source of further details is the Wikipedia article at

http://en.wikipedia.org/wiki/Sudoku

Generating a grid

I first came across sudoku by chance in a Web news item. Surfing a little, I found a site which contained a statement that a program in APL had taken all night to solve a difficult puzzle. Another offered a program to solve puzzles but that might require occasional prompting. I was intrigued enough to make a start on a program of my own, but decided to start out by writing one to generated valid complete grids. One way to do this is to write a solver that uses a brute-force backtracking method — one that sets up and tests every allowed combination — and then to use that to find a valid grid by seeding it with a few random placements. Such programs are both common and fast, but give little insight into the difficulty of a puzzle and into the type of strategies that a human solver might have to employ, although they can easily determine the number of solutions a puzzle might have.

I decided instead to generate a grid by filling its first row with the nine digits in a random order, obtained using `random_number`, and then adding subsequent rows of randomly arranged digits, sorting their order as necessary to avoid clashes with digits already in place on previous rows or already in the current box. If this cannot be done, a fresh set of random numbers is used. For the third and sixth rows, an additional check is required that the box constraint is also fulfilled. Also, a deadlock condition can arise when the three digits missing from an almost completed box have already been assigned to a single column above the box. If this occurs, the program restarts. This program worked rather slowly but on looking at it again after a very long summer break, I realized that the sorting algorithm took account of the clash of a digit only in its original position, and not in its possible intended new one. When this was corrected, the program was able to generate a valid grid in less that 1msec. Since it had been written in a parameterized way, it worked too for any square grid from 4 x 4 upwards. Thus it could generate 16 x 16 grids in about 4msecs as well as 25 x 25 in about 60msecs. (All timings are with CVF 6.6C with high optimization on a 1.4GHz PC.) The method slows down beyond that, 36 x 36 grids taking a few seconds and more massive ones being still a ‘work in progress’. (Larger grids not only require more computation, but contain ever more complicated potential deadlock conditions)

An option in the generator enables the construction of X-sudokus, in which there is the additional constraint that the two diagonals also each contain all the digits 1 to 9; this takes about 2secs for 9 x 9 and 15 minutes for 16 x 16.

The solver

The next step was to use the program as a solver, by modifying the program to deal with the additional constraints provided by the given digits in a puzzle. For any row, there are constraints from the previous rows and from the set values on the row itself and from those below it. This worked (when all bugs were ironed out and some additional deadlock code implemented), but was very slow especially for puzzles with few givens and for ill-formed puzzles with multiple solutions. It was clear that some logic had to be applied and I then coded a solver that applied a number of logic rules. Here, the Fortran 95 features `where`, `any` and `count` really came into their own. For instance, the program starts by initializing an array, defined as `block(9, 9, 9)`, to contain, in its last dimension, the candidate digits 1 to 9 in order at each position in a 9 x 9 grid. Many of these candidates can be removed immediately because they violate one or more of the row/column/box constraints. The code to do that is:

```
do row = 1, rank2
   do col = 1, rank2
      if(part(row, col) == 0) cycle
      where (block(row, :, part(row, col)) == part(row, col))
                                 block(row, :, part(row, col)) = 0
      where (block(:, col, part(row, col)) == part(row, col))                &
                                 block(:, col, part(row, col)) = 0
      where (block(((row-1)/rank)*rank+1:((row-1)/rank)*rank+rank,           &
                  ((col-1)/rank)*rank+1:((col-1)/rank)*rank+rank, part(row, col))      &
                                                      == part(row, col))      &
             block(((row-1)/rank)*rank+1:((row-1)/rank)*rank+rank,           &
                  ((col-1)/rank)*rank+1:((col-1)/rank)*rank+rank, part(row, col)) = 0
   end do
end do
```

where `rank` is the size of a box (*e.g.* 3), `rank2` is the size of the grid (*e.g.* 9), and `part` contains the initial given values (as in the puzzle at the beginning of this article) and subsequently the partial solution.

Other candidates can be removed by the application of various logic rules. For instance, if a row has two unfilled cells that each can contain only the same two candidate values, say 3 and 6, then those two values can be removed as candidates from any other unfilled cell on that row. After the elimination of candidates, there will be cells in which only one candidate remains. That is the solution for that cell. Similarly, there might be a single candidate for a certain value along a row, column or box, and the code to accept that value as the solution for a cell and to update the others is, for a row:

```
do row = 1, rank2
v1: do val = 1, rank2
     if(count(block(row, :, val) == val) /= 1) cycle
        do col = 1, rank2
           if(part(row, col) /= 0) cycle
            if(block(row, col, val) == val)) then
            part(row, col) = val
            call update
            exit v1
         end if
      end do
   end do v1
end do
```

where the subroutine `update` sets `block(row, col, :) = 0` and is otherwise is similar to the first code extract. The solver iterates until either the puzzle is solved or no further progress is possible.

This program is very fast. Only for ill-formed puzzles with multiple solutions, or for a small class that requires very subtle logic rules (X-wing, swordfish *etc*., for the initiated), does the original solver also have to be invoked to finish off the solution.

Generating a puzzle

At this stage, all the components were available to generate valid puzzles. A full grid can be generated and then a random set of values removed (typically around two-thirds for 9 x 9 and just over half for 16 x 16). The resulting candidate puzzle can be solved and if it has a unique solution it is a valid puzzle. Otherwise, a different random set of values is removed, and the procedure repeated until a set resulting in a unique solution is found. The more given values we start off with, the faster this happens. At the step where random values are removed, a symmetry condition can be applied. The time required to produce a puzzle is several times the time required to generate a grid.

Note that this technique does not require a test on whether a candidate puzzle has no solution. As we start from a valid grid this can never be the case.

Building on 9 x 9 grids, it is possible to use the program to construct samurai (or butterfly) puzzles. These are puzzles of five 9 x 9 grids in which a central puzzle has an overlapping puzzle on each of its corner boxes. A samurai has to be constructed such that no individual puzzle can be solved without recourse to information from its neighbours. This requires a mixture of automation and hand intervention. A check that the final result has one and only solution requires, for the moment, a hand check, at least up to the point where certain critical cells in the overlapping boxes have been correctly solved.

How hard is a puzzle?

In general, it is not possible to judge the degree of difficulty of a sudoku puzzle by inspection. In particular, the number of givens is only a weak indicator of its difficulty. After attempting several ways of assessing the difficulty of published puzzles (a convenient calibration), with essentially no success, I finally hit upon the idea of using a technique that a human solver might apply: although an immediate assessment is not possible, after a first round of filling in some cells with ‘obvious’ solutions and noting various further possibilities one has a pretty good idea what one is up against. An excruciating puzzle, for example, will have very few open possibilities on any given round. An easy one is done in just a few rounds with many options on each one. The logic solver is instrumented to measure how much ‘work’, after an initial ‘clean-up’ pass, each algorithm carries out, and how many iterations it all takes, as well as how much remains to be done if ‘brute-force’ is still required. These values are all combined into a so-called ‘difficulty index’ which, for 9 x 9, has a minimum of -9 (for an already-solved puzzle) and goes up to about 40. The program discards any generated puzzle with a value below 9 as being trivial.

Using this index, the program can be run until it produces a puzzle with a unique solution above a given level of difficulty. All the non-trivial puzzles that it produces on the way can be stored for potential use. Each time a puzzle is stored, a new grid is generated.

An accepted sudoku can often be ‘polished’ by the judicious removal of a few selected givens. This increases the difficulty index but must be done so as to retain the uniqueness of the solution. For a symmetric puzzle, symmetry too must be maintained. In the puzzle above, the program has removed three cells from column 5 of the initial puzzle.

Shapes, slogans and symbols

The technique described above is to generate a grid and to find a puzzle which yields that grid as its unique solution. It is possible to do the opposite: define a *pattern* for the puzzle — a pretty shape, a few letters or whatever — and generate grids until one is generated that corresponds to a unique solution of the given pattern. An example, produced for children at the UNIS school in New York, is:

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

(where the use of colour to distinguish the letters better is rendered here by Roman/*italic*). This works too for 25 x 25, where examples, too big to show here as grids, are:

```
MIND         and     7TH
THE                  JAN
GAP.                 1986
```

These puzzles tend to have a fairly low difficulty index, but can be produced in times ranging from less than a second to several minutes.

Conclusion

This note brings no particular insights into the sudoku game, nor into Fortran techniques. It is rather a record of how I came to write a flexible application that can solve puzzles as well generate grids and puzzles, assess their difficulty and, within reason, mould set values into interesting shapes. However, the elegance of Fortran 95's array-handling features made the programming a delight, and the whole program is only about 2000 non-comment lines. Where it does, perhaps, score over other programs is in its use of allocatable arrays to allow it to work for various grid sizes. I even temporarily disabled the sanity check on the grid-size prompt to allow me to enter a degenerate size of 1, and the program duly generated a 1 x 1 grid with the digit 1 in its solitary cell! Also, the technique of generating a puzzle from a solution, rather than a solution from a potential puzzle, has proved to be a flexible way of working.

A number of people were patient enough to solve sample puzzles for me and to offer advice. In particular, I thank Mike Gerard, Hans Grote and Eric McIntosh, all associated with CERN, Geneva, for their forbearance.