

Chapter 18

Sudoku

The remarkably popular puzzle demonstrates man versus machine, backtracking and recursion, and the mathematics of symmetry.

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

Figure 18.1. *A Sudoku puzzle with especially pleasing symmetry. The clues are shown in blue.*

You probably already know the rules of *Sudoku*, but figures 18.1 and 18.2 illustrate them. Figure 18.1 is the initial 9-by-9 grid, with a specified few digits

known as the *clues*. I especially like the symmetry in this example, which is due to Gordon Royle of the University of Western Australia [1]. Figure 18.2 is the final completed grid. Each row, each column, and each major 3-by-3 block, must contain exactly the digits 1 through 9. In contrast to magic squares and other numeric puzzles, no arithmetic is involved. The elements in a *Sudoku* grid could just as well be nine letters of the alphabet, or any other distinct symbols.

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

Figure 18.2. The completed puzzle. The digits have been inserted so that each row, each column, and each major 3-by-3 block contains 1 through 9.

Sudoku is actually an American invention. It first appeared, with the name Number Place, in the Dell Puzzle Magazine in 1979. The creator was probably Howard Garns, an architect from Indianapolis. A Japanese publisher, Nikoli, took the puzzle to Japan in 1984 and eventually gave it the name *Sudoku*, which is a kind of kanji acronym for “numbers should be single, unmarried.” The Times of London began publishing the puzzle in the UK in 2004 and it was not long before it spread back to the US and around the world.

The fascination with solving *Sudoku* by hand derives from the discovery and mastery of a myriad of subtle combinations and patterns that provide tips toward the solution. The Web has hundreds of sites describing these patterns, which have names like “hidden quads”, “X-wing” and “squirmbag”.

It is not easy to program a computer to duplicate human pattern recognition capabilities. Most *Sudoku* computer codes take a very different approach, relying on the machine’s almost limitless capacity to carry out brute force trial and error. Our MATLAB program, `sudoku.m`, uses only one pattern, singletons, together with recursive backtracking.

To see how our `sudoku` program works, we can use *Shidoku* instead of *Sudoku*. “Shi” is Japanese for “four”. The puzzles, which are almost trivial to solve by

1			
		3	
	2		
			4

Figure 18.3. *Shidoku*

1	3 4	2 4	2
2 4	4	3	1 2
3 4	2	1	1 3
3	1 3	1 2	4

Figure 18.4. *Candidates*

1	3 4	2 4	2
2 4	4	3	1 2
4	2	1	1 3
3	1	1 2	4

Figure 18.5. *Insert singleton*

1	3	4	2
2	4	3	1
4	2	1	3
3	1	2	4

Figure 18.6. *Solution*

hand, use a 4-by-4 grid. Figure 18.3 is our first *Shidoku* puzzle and the next three figures show steps in its solution. In figure 18.4, the possible entries, or candidates, are shown by small digits. For example, row two contains a “3” and column one contains a “1” so the candidates in position (2,1) are “2” and “4”. Four of the cells have only one candidate each. These are the *singletons*, shown in red. In figure 18.5, we have inserted the singleton “3” in the (4,1) cell and recomputed the candidates. In figure 18.6, we have inserted the remaining singletons as they are generated to complete the solution.

1			
	2		
		3	
			4

Figure 18.7. *diag(1:4)*

1	3 4	2 4	2 3
3 4	2	1 4	1 3
2 4	1 4	3	1 2
3 2	1 3	1 2	4

Figure 18.8. *No singletons.*

1			
3	2		
		3	
			4

Figure 18.9. *Backtrack step.*

1	4	2	3
3	2	4	1
4	1	3	2
2	3	1	4

Figure 18.10. *Solution is not unique.*

The input array for figure 18.7 is generated by the MATLAB statement

```
X = diag(1:4)
```

As figure 18.8 shows, there are no singletons. So, we employ a basic computer science technique, recursive backtracking. We select one of the empty cells and tentatively insert one of its candidates. We have chosen to consider the cells in

the order implied by MATLAB one-dimensional subscripting, $X(:)$, and consider the candidates in numerical order, so we tentatively insert a “3” in cell (2,1). This creates a new puzzle, shown in figure 18.9. Our program is then called recursively. In this example, the new puzzle is easily solved and the result is shown in figure 18.10. However, the solution depends upon the choices that we made before the recursive call. Other choices can lead to different solutions. For this simple diagonal initial condition, the solution is not unique. There are two possible solutions, which happen to be matrix transposes of each other.

```
>> Y = shidoku(diag(1:4))

Y =
     1     4     2     3
     3     2     4     1
     4     1     3     2
     2     3     1     4

>> Z = shidoku(diag(1:4)')'

Z =
     1     3     4     2
     4     2     1     3
     2     4     3     1
     3     1     2     4
```

Mathematicians are always concerned about existence and uniqueness in the various problems that they encounter. For *Sudoku*, neither existence nor uniqueness can be determined easily from the initial clues. With the puzzle in figure 18.1, if we were to insert a “1”, “5” or “7” in the (1,1) cell, the row, column and block conditions would still be satisfied, but it turns out that the resulting puzzle has no solution. It would be very frustrating if such a puzzle were to show up in your daily newspaper. Backtracking generates many impossible configurations. The recursion is terminated by encountering a puzzle with no solution.

Uniqueness is also a elusive property. In fact, most descriptions of *Sudoku* do not specify that there has to be exactly one solution. Again, it would be frustrating to find a different solution from the one given by your newspaper. The only way that I know to check uniqueness is to exhaustively enumerate all possibilities.

A number of operations on a *Sudoku* grid can change its visual appearance without changing its essential characteristics. All of the variations are basically the same puzzle. These equivalence operations can be expressed as array operations in MATLAB. For example

```
p = randperm(9)
z = find(X > 0)
X(z) = p(X(z))
```

permutes the digits representing the elements. Other operations include

```

X'
rot90(X,k)
flipud(X)
fliplr(X)
X([4:9 1:3],:)
X(:,[randperm(3) 4:9])

```

If we do not count the comments and GUI, `sudoku.m` involves less than 40 lines of code. The outline of the main program is:

- Fill in all singletons.
- Exit if a cell has no candidates.
- Fill in a tentative value for an empty cell.
- Call the program recursively.

Here is the code for the main program. All of the bookkeeping required by backtracking is handled by the recursive call mechanism in MATLAB and the underlying operating system.

```

function X = sudoku(X)
% SUDOKU Solve Sudoku using recursive backtracking.
%   sudoku(X), expects a 9-by-9 array X.

% Fill in all "singletons".
% C is a cell array of candidate vectors for each cell.
% s is the first cell, if any, with one candidate.
% e is the first cell, if any, with no candidates.

[C,s,e] = candidates(X);
while ~isempty(s) && isempty(e)
    X(s) = C{s};
    [C,s,e] = candidates(X);
end

% Return for impossible puzzles.

if ~isempty(e)
    return
end

% Recursive backtracking.

if any(X(:) == 0)
    Y = X;
    z = find(X(:) == 0,1);           % The first unfilled cell.

```

```

    for r = [C{z}]                % Iterate over candidates.
        X = Y;
        X(z) = r;                % Insert a tentative value.
        X = sudoku(X);           % Recursive call.
        if all(X(:) > 0)         % Found a solution.
            return
        end
    end
end
end

```

The key internal function is `candidates`.

```

function [C,s,e] = candidates(X)
    C = cell(9,9);
    tri = @(k) 3*ceil(k/3-1) + (1:3);
    for j = 1:9
        for i = 1:9
            if X(i,j)==0
                z = 1:9;
                z(nonzeros(X(i,:))) = 0;
                z(nonzeros(X(:,j))) = 0;
                z(nonzeros(X(tri(i),tri(j)))) = 0;
                C{i,j} = nonzeros(z)';
            end
        end
    end
    L = cellfun(@length,C); % Number of candidates.
    s = find(X==0 & L==1,1);
    e = find(X==0 & L==0,1);

end % candidates

```

For each empty cell, this function starts with `z = 1:9` and uses the numeric values in the associated row, column and block to zero elements in `z`. The nonzeros that remain are the candidates. For example, consider the (1,1) cell in figure 18.1. We start with

```
z = 1 2 3 4 5 6 7 8 9
```

The values in the first row change `z` to

```
z = 1 0 0 0 5 6 7 8 9
```

Then the first column changes `z` to

```
z = 1 0 0 0 5 0 7 0 9
```

The (1,1) block does not make any further changes, so the candidates for this cell are `C{1,1} = [1 5 7 9]`.

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

Figure 18.11. *The initial candidates for the puzzle in figure 18.1. There are no singletons.*

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

Figure 18.12. *The first step in the backtracking.*

Figure 18.1 is actually a very difficult puzzle, either by hand or by machine. Figures 18.11 through 18.14 are a few snapshots of the solution process. The initial candidates are shown in figure 18.11. There are no singletons, so the first recursive step, shown in figure 18.12, happens immediately. (We know that this puzzle with a “1” in the (1,1) cell has no solution, but the program needs to rediscover that fact every time.) Figure 18.13 shows how the first column is filled in for the first time at

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

Figure 18.13. After 22 steps the first column has been filled with possible values, but this track will eventually fail. The cyan values are generated by the backtracking and the green values are implied by the others.

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

Figure 18.14. After 14,781 steps, we appear to be close to a solution, but it is impossible to continue. The eventual solution is in figure 18.2.

step 22. The elements in cyan are the tentative values from the backtracking and the elements in green are implied by those choices. But we're still a long way from the solution. After 3,114 steps the recursion puts a "5" in the (1,1) cell and after 8,172 steps it tries a "7". Figure 18.14 shows the situation after 14,781 steps. We appear

to be close to a solution because 73 of the 81 cells have been assigned values. But the first row and last column already contain all the of the digits from 1 through 9, so there are no values left for the (1,9) cell in the upper right corner. The candidate list for this cell is empty and the recursion terminates. (There are also two cells in the third row showing the same singleton “3”. This could be another reason to terminate the recursion, but it is not necessary to check.) Finally, after 19,229 steps backtracing finally tries a “9” in cell (1,1). The “9” is a good idea because less than 200 steps later, after 19,422 steps, the program reaches the solution shown in figure 18.2. This is many more steps than most puzzles require.

References

- [1] Gordon Royle, Symmetry in Sudoku,
<http://people.csse.uwa.edu.au/gordon/sudoku/sudoku-symmetry.pdf>
- [2] Sudoku Squares and Chromatic Polynomials,
<http://www.ams.org/notices/200706/tx070600708p.pdf>
- [3] Strategy families,
http://www.scanraid.com/Strategy_Families
- [4] Nikoli,
http://www.nikoli.co.jp/en/misc/2008speech_from_ussc.htm

Exercises

18.1 *Solve*. Solve a Sudoku puzzle by hand.

18.2 *sudoku_puzzle*. The EXM program `sudoku_puzzle` generates 15 different puzzles. The comments in the program describe the origins of the puzzles. How many steps are required by `sudoku.m` to solve each of the puzzles?

18.3 *Patterns*. Add some human puzzle solving techniques to `sudoku.m`. This will complicate the program and require more time for each step, but should result in fewer total steps.

18.4 *sudoku_alpha*. In `sudoku.m`, change `int2str(d)` to `char('A'+d-1)` so that the display uses the letters 'A' through 'I' instead of the digits 1 through 9. See figure 18.15. Does this make it easier or harder to solve puzzles by hand.

18.5 *sudoku16*. Modify `sudoku.m` to solve 16-by-16 puzzles with 4-by-4 blocks.

	B			C			D	
F								C
		D				E		
			H		F			
H				A				F
			G		E			
		G				F		
D								H
	C			D			B	

Figure 18.15. Use the letters 'A' through 'I' instead of the digits 1 through 9.