Chapter 14

Sudoku

Humans look for patterns, but machines use backtracking.

The fascination that people have for solving *Sudoku* puzzles without a computer 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 our intention to describe a MATLAB program that duplicates these human pattern recognition capabilities. Instead, our program, like most other *Sudoku* computer codes, takes a very different approach, one that relies on the machine's almost limitless capacity to carry out brute force trial and error. We use only one pattern, *singletons*, together with a fundamental computer science technique, recursive backtracking.

In the Sudoku world, backtracking is regarding as "guessing" and is considered bad form. But as a computational technique, it is easy to understand, straightforward to implement and guaranteed to succeed.

In this chapter, we will describe five MATLAB programs for experiments with *Sudoku*.

- `sudoku`. An interactive program that lets you follow the computer solution process.
- `sudoku_basic`. The same solution process as `sudoku`, but without the graphics interface. Much easier to read, and much faster if you just want the solution.
- `sudoku_all`. Enumerate all solutions to a Sudoku puzzle. A valid puzzle should have only one solution.

October 2, 2011

- `sudoku_assist`. An interactive program that lets you control the solution process, as you would by hand on paper.
- `sudoku_puzzle`. A collection of interesting puzzles.

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.

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

Figure 14.1. *A Sudoku puzzle featuring Lo-Shu, the magic square of order 3.*

You probably already know the rules of *Sudoku*. Figure 14.1 is an example of an initial 9-by-9 grid, with a few specified digits known as the *clues*. This particular puzzle reflects our interest in magic squares. It is the sum of the two matrices shown in figure 14. The matrix on the left is generated by the functions `kron`, which computes something known as the *Kronecker product*, and `magic`, which generates magic squares.

```
X = kron(eye(3),magic(3))
```

In this case, a Kronecker product involving the identity matrix produces a block matrix with three copies of `magic(3)` on the diagonal. By itself, this array is not a valid Sudoku puzzle, because there is more than one way to complete it. The solution is not unique. In order to get a puzzle with a unique solution, we have added the matrix on the right in figure 14.

Figure 14.3 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

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

								3
							8	
			1					
		3						
						1		
					3			
	2							
1								

Figure 14.2. *Our Lo-Shu based puzzle is the sum of these two matrices.*

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

Figure 14.3. *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. The original clues are shown in blue.*

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.

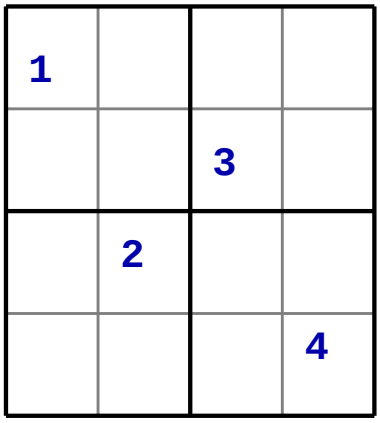

Figure 14.4. *Shidoku*

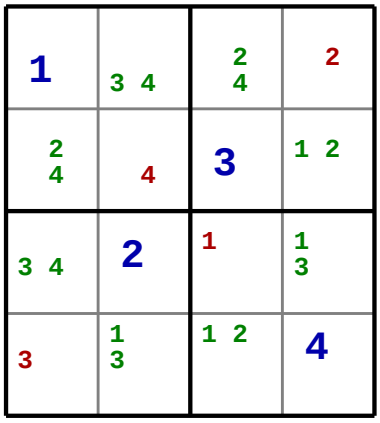

Figure 14.5. *Candidates*

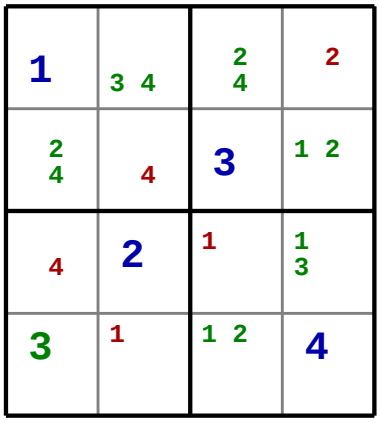

Figure 14.6. *Insert singleton*

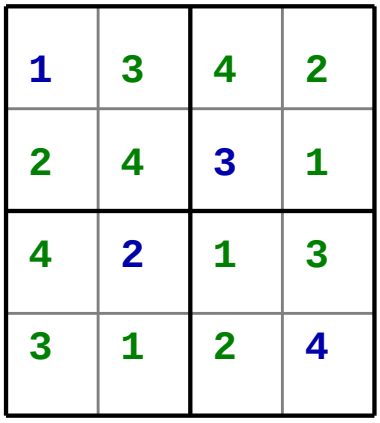

Figure 14.7. *Solution*

Shidoku

To see how our `sudoku` program works, we can use *Shidoku* instead of *Sudoku*. "Shi" is Japanese for "four". The puzzles use a 4-by-4 grid and are almost trivial to solve by hand. Figure 14.4 is our first *Shidoku* puzzle and the next three figures show steps in its solution. In figure 14.5, 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 14.6, we have inserted the singleton 3 in the (4,1) cell and recomputed the candidates. In figure 14.7, we have inserted the remaining singletons as they are generated to complete the solution.

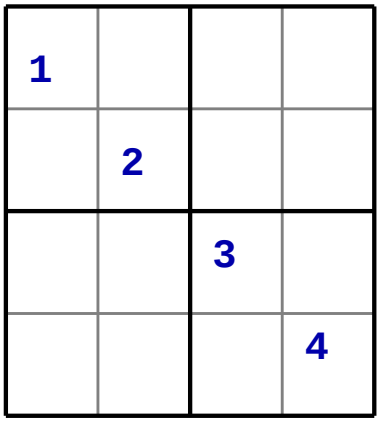

Figure 14.8. *diag(1:4)*

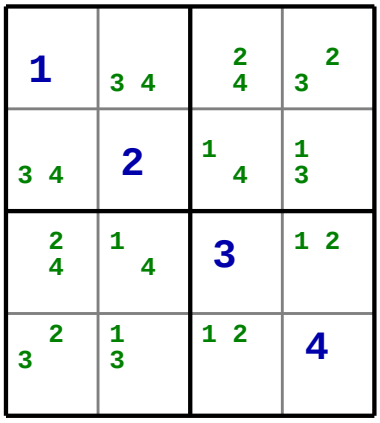

Figure 14.9. *No singletons.*

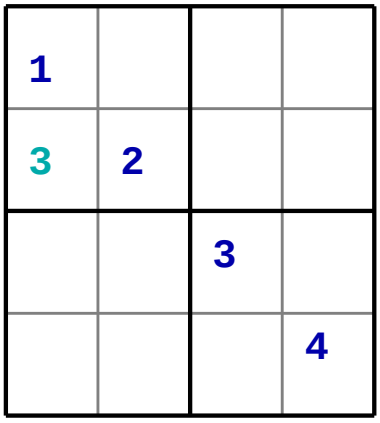

Figure 14.10. *Backtrack step.*

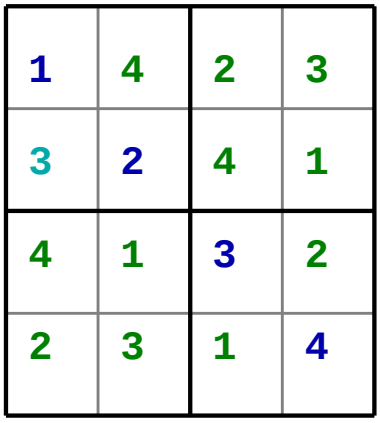

Figure 14.11. *Solution is not unique.*

The input array for figure 14.8 is generated by the MATLAB statement

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

As figure 14.9 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. All of the empty cells have two candidates, so we pick the first one and tentatively insert a 3 in cell (2,1). This creates a new puzzle, shown in figure 14.10. Our program is then called recursively. In this example, the new puzzle is easily solved and the result is shown in figure 14.11. 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
```

Existence and uniqueness

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. It would be very frustrating if a puzzle with no solution, or with more than one solution, were to show up in your daily newspaper.

Uniqueness is an elusive property. In fact, most descriptions of *Sudoku* do not specify that there has to be exactly one solution. The only way that I know to check uniqueness is to exhaustively enumerate all possibilities.

For our magic square puzzle in figure 14.1, if we were to replace the 1 in the (9,1) cell by a 5, 6 or 7, the row, column and block conditions would still be satisfied. It turns out that a 5 would produce another valid puzzle with a unique solution, a 6 would produce a puzzle with two possible solutions, and a 7 would produce a puzzle with no solution. An exercise asks you to verify these claims.

It takes `sudoku_all` a little over half an hour on my laptop to determine that the magic square puzzle generated by

```
X = kron(eye(3),magic(3))
```

without the addional entries on right in figure 14 has 283576 solutions.

The puzzle generated by

```
X = sudoku_puzzle(13)
```

has an interesting spiral pattern. But there are over 300 solutions. An exercise asks you to find out exactly how many.

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])
```

The algorithm

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.

sudoku and sudoku_basic

Here is the header for `sudoku_basic`.

```
function [X,steps] = sudoku_basic(X,steps)
% SUDOKU_BASIC  Solve the Sudoku puzzle using recursive backtracking.
%   sudoku_basic(X), for a 9-by-9 array X, solves the Sudoku puzzle for X
%   without providing the graphics user interface from sudoku.m
%   [X,steps] = sudoku_basic(X) also returns the number of steps.
%   See also sudoku, sudoku_all, sudoku_assist, sudoku_puzzle.
```

This first bit of code checks `nargin`, the number of input arguments, and initializes the step counter on the first entry. Any subsequent recursive calls will have a second argument.

```
   if nargin < 2
      steps = 0;
   end
```

These comments describe the primary variables in the program.

```
% C is the array of candidate values for each cell.
% N is the vector of the number of candidates for each cell.
% s is the index of the first cell with the fewest candidates.
```

The program fills in any singletons, one at a time. The candidates are recomputed after each step. This section of code, by itself, can solve puzzles that do not require backtracking. Such puzzles can be classified as "easy".

```
[C,N] = candidates(X);
while all(N>0) & any(N==1)
   s = find(N==1,1);
   X(s) = C{s};
   steps = steps + 1;
   [C,N] = candidates(X);
end
```

If filling in the singletons does not solve the puzzle, we reach the following section of code that implements backtracking. The backtracking generates many impossible configurations. The recursion is terminated by encountering a puzzle with no solution.

```
if all(N>0)
   Y = X;
   s = find(N==min(N),1);
   for t = [C{s}]                          % Iterate over the candidates.
      X = Y;
      X(s) = t;                            % Insert a tentative value.
      steps = steps + 1;

      [X,steps] = sudoku_basic(X,steps);   % Recursive call.

      if all(X(:) > 0)                     % Found a solution.
         break
      end
   end
end
```

All of the bookkeeping required by backtracking is handled by the recursive call mechanism in MATLAB and the underling operating system.

The key internal function for `sudoku` is `candidates`. Here is the preamble.

```
function [C,N] = candidates(X)
   % C = candidates(X) is a 9-by-9 cell array of vectors.
   % C{i,j} is the vector of allowable values for X(i,j).
   % N is a row vector of the number of candidates for each cell.
   % N(k) = Inf for cells that already have values.
```

The following statement creates an internal function named `tri` that computes the indices for blocks. For example, `tri(1)`, `tri(2)` and `tri(3)` are all equal to `1:3` because the block that includes a cell with row or column index equal to `1`, `2` or `3` has row or column indices `1:3`.

```
tri = @(k) 3*ceil(k/3-1) + (1:3);
```

Here is the core of `candidates`.

```
C = cell(9,9);
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
```

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 14.1. 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]`.

The `candidates` function concludes with the following statements. The number of candidates, `N(i,j)` for cell `(i,j)` is the length of `C{i,j}`. If cell `(i,j)` already has a value, then `X(i,j)` is nonzero, `C{i,j}` is empty and so `N(i,j)` is zero. But we make these `N(i,j)` infinite to distinguish them from the cells that signal an impossible puzzle.

```
N = cellfun(@length,C);
N(X>0) = Inf;
N = N(:)';
```

An example, puzzle number one

Figure 14.12 shows the key steps in the solution of our Lo-Shu puzzle. The initial candidates include just one singleton, the red 5 in position (3,9). Five steps with five singletons allow us to fill in the cells with green 5's and 8's. We now consider the first cell with two candidates, in position (6,1). The first step in the backtracking puts a tentative 6 in position (6,1). Blue values are the initial clues, cyan values are generated by the backtracking and green values are implied by the others. After five steps we reach an impossible situation because there are no candidates for position (6,2). Try again, with a 7 in position (6,1). This puzzle is easily completed by filling in singletons. The title on the final solutions shows that it took 50 steps, including the abortive ones in the first backtrack, to reach the solution.

References

[1] Wikipedia's Mathematics of Sudoku.
http://en.wikipedia.org/wiki/Mathematics_of_Sudoku
[2] Gordon Royle, Minimum Sudoku.
http://mapleta.maths.uwa.edu.au/~gordon/sudokumin.php
[3] Nikoli.
http://www.nikoli.co.jp/en
[4] Sudoku Squares and Chromatic Polynomials.
http://www.ams.org/notices/200706/tx070600708p.pdf
[5] Sudoku web site modeled after Wikipedia.
http://www.sudopedia.org/wiki
[6] Ruud's Guide, SudoCue.
http://www.sudocue.net/guide.php

Recap

```
%% Sudoku Chapter Recap
% This is an executable program that illustrates the statements
% introduced in the Sudoku Chapter of "Experiments in MATLAB".
% You can access it with
%
%    sudoku_recap
%    edit sudoku_recap
%    publish sudoku_recap
%
%  Related EXM programs
%
%    sudoku
%    sudoku_all
%    sudoku_assist
%    sudoku_basic
%    sudoku_puzzle
```

```
%% Disclaimer
% Our Sudoku chapter and Sudoku program depend heavily
% upon recursion, which cannot be done by this script.

%% Sudoku puzzle incorporating the Lo-Shu magic square.
   X = kron(eye(3),magic(3))
   C = full(sparse([9 8 4 3 7 6 2 1], [1:4 6:9], [1 2 3 1 3 1 8 3]))
   X = X + C

% Also available as
   X = sudoku_puzzle(1);

%% Transforms of a Sudoku puzzle
   T = X;
   p = randperm(9);
   z = find(X > 0);
   T(z) = p(X(z))
   X'
   rot90(X,-1)
   flipud(X)
   fliplr(X)
   X([4:9 1:3],:)
   X(:,[randperm(3) 4:9])

%% Candidates

% C = candidates(X) is a cell array of vectors.
% C{i,j} is the set of allowable values for X(i,j).

   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
   C

%% First singleton and first empty.
```

```
% N = number of candidates in each cell.
% s = first cell with only one candidate.
% e = first cell with no candidates.

   N = cellfun(@length,C)
   s = find(X==0 & N==1,1)
   e = find(X==0 & N==0,1)

%% Sudoku puzzles

   help sudoku_puzzle

   for p = 1:16
      sudoku_puzzle(p)
   end
```

Exercises

14.1 *xkcd.* Solve the binary Sudoku puzzle by Randal Munroe in the Web comic strip `xkcd` at

```
http://xkcd.com/74
```

14.2 *Solve.* Use `sudoku` to solve a puzzle from a newspaper, magazine, or puzzle book. Use `sudoku_assist` to solve the same puzzle by hand.

14.3 *sudoku_puzzle.* The `exm` program `sudoku_puzzle` generates 16 different puzzles. The comments in the program describe the origins of the puzzles. How many steps are required by `sudoku_basic` to solve each of the puzzles?

14.4 *By hand.* Use `sudoku_assist` to solve our Lo-Shu based puzzle by hand.

```
X = sudoku_puzzle(1);
sudoku_assist(X)
```

14.5 *Modify puzzle #1.* The following program modifies the lower left hand clue in our Lo-Shu based puzzle. What happens with each of the resulting puzzles?

```
X = sudoku_puzzle(1);
for t = [0 1 5 6 7]
   X(9,1) = t;
   sudoku_all(X)
end
```

14.6 *Try this.* Try to use `sudoku_assist` to solve this puzzle by hand.

```
X = sudoku_puzzle(1);
X(9,1) = 7;
sudoku_assist(X)
```

14.7 *Puzzle #3.* How many solutions does `sudoku_puzzle(3)` have? How do they differ from each other?

14.8 *Puzzle #13.* How many solutions does `sudoku_puzzle(13)` have? In addition to the initial clues, what values are the same in all the solutions?

14.9 *All solutions.* What are the differences between `sudoku_basic` and `sudoku_all`?

14.10 *Combine.* Combine `sudoku_all` and `sudoku_basic` to create a program that returns all possible solutions and the number of steps required to find each one. Try your program on `sudoku_puzzle(13)`. What is the average number of steps required? Hints: if `S` is a cell array where each cell contains one numeric value, then `s = [S{:}]` creates the corresponding vector. And, `help mean`.

14.11 *Search strategy* In `sudoku_basic`, the statement

```
s = find(N==min(N),1);
```

determines the search strategy. What happens when you change this to

```
s = find(N==min(N),'last');
```

or

```
s = find(X==0,'first');
```

14.12 *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.

14.13 *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 14.13. Does this make it easier or harder to solve puzzles by hand.

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

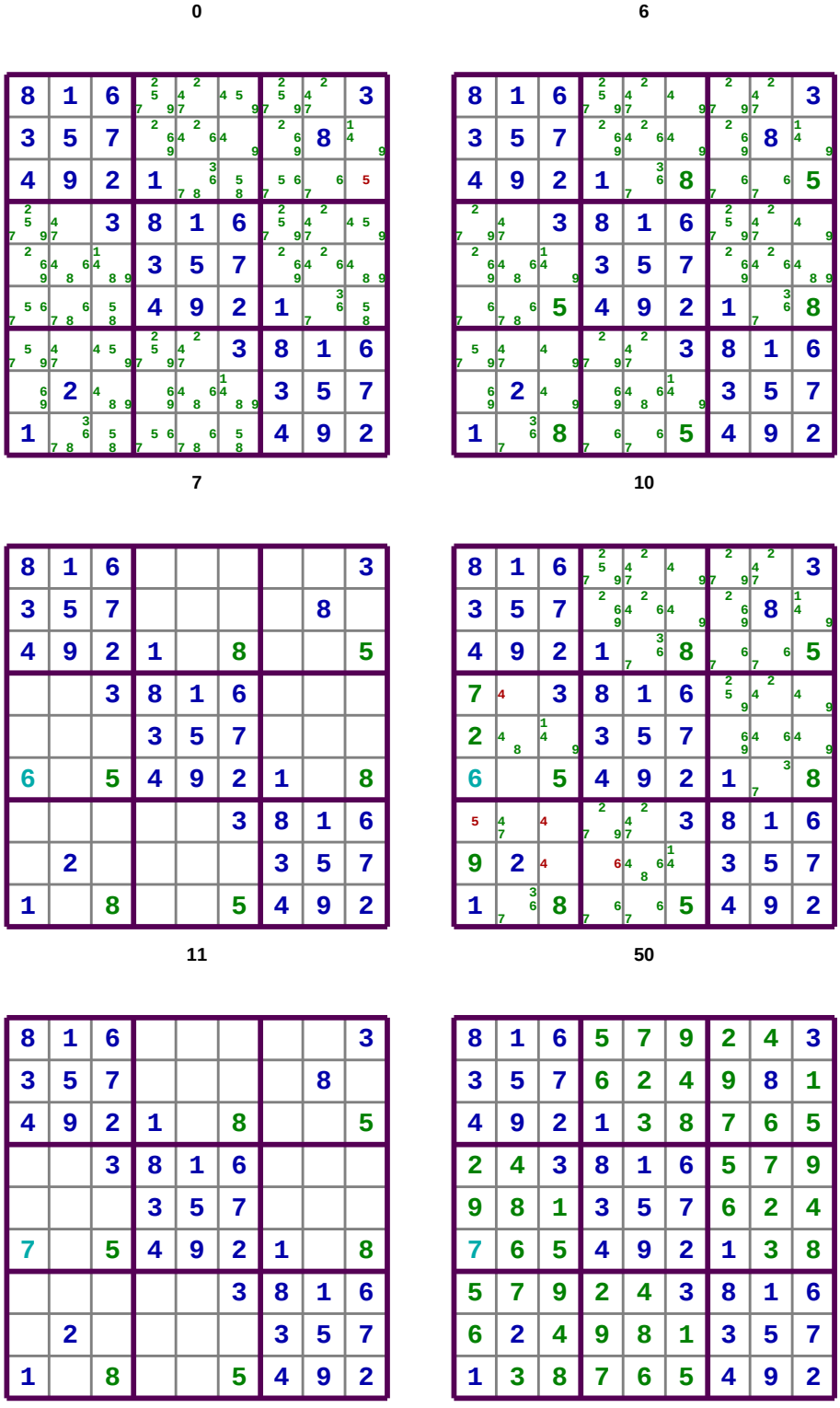

Figure 14.12. *The key steps in the solution of our Lo-Shu puzzle.*

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

Figure 14.13. *Use the letters 'A' through 'I' instead of the digits 1 through 9.*