Sudoku Designs

Bill Venables

2019-05-31

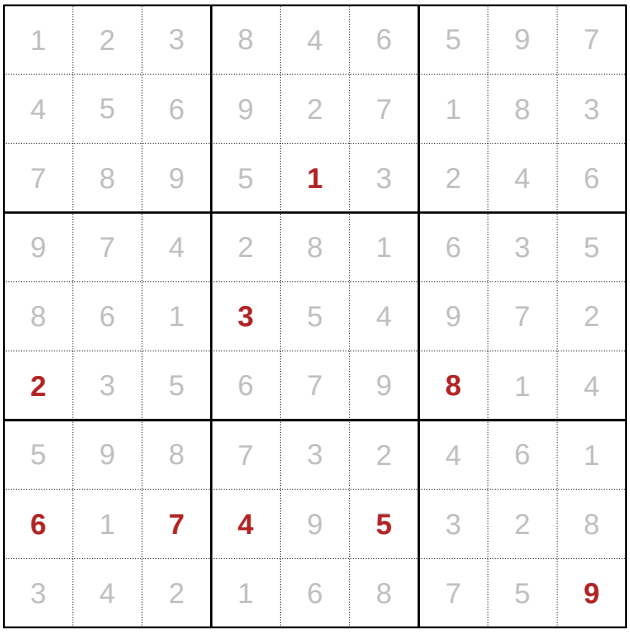
Contents

1 Introduction

A Sudoku design is an $n^2 \times n^2$ Latin square design with the additional constraint that if the pattern is further subdivided into an $n \times n$ array of smaller $n \times n$ squares, then each of the smaller squares itself has a complete replicate of the symbols used in the design. Here n is a positive integer, in practice $n > 1$ to avoid trivialities and $n < 6$ is usual.

This wordy description is more easily understood by showing an example for $n = 3$ and using the digits $1, 2, \ldots, 9$ as the symbols.

```
library(sudokuAlt)
set.seed(2019)
seedGame(3) %>% solve() %>% regulariseGame() %>% plot()
```



The example above is in a canonical form, or "regularised", by ensuring the symbols are labelled in such a way that those in the top left 3×3 sub-square are in lexicographical order by row.

In a Sudoku *puzzle* the player is given a partially completed Sudoku design and the challenge is to fill out the vacant squares in such a was that the constraints are satisfied. That is, after completion,

- Every row must contain a complete set of the n^2 symbols
- Every column must also contain a complete set of symbols, and
- Every $n \times n$ block must also contain a complete set.

The following example shows a typical puzzle where the supplied entries are shown in red and one possible completion shown in grey. This is for the typical case of $n = 3$.

```
g <- makeGame() %>% solve() %>% plot()
```

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

For larger examples, using letters for the symbols is more convenient. Here is an example for $n = 4$, regularised:

```
set.seed(2019)
g4 <- seedGame(4) %>% solve() %>% regulariseGame() %>% plot()
```

A	B	C	D	E	**H**	K	L	N	O	F	P	I	J	G	M
E	F	G	H	C	N	M	P	D	J	K	I	B	O	L	A
I	J	**K**	L	A	B	G	O	E	M	C	H	F	N	D	P
M	N	O	P	D	I	F	J	B	G	A	L	K	E	C	H
P	A	J	**N**	H	K	L	B	G	I	M	O	C	D	F	**E**
L	E	H	K	N	O	D	M	C	B	P	F	G	I	A	J
C	D	**F**	I	G	J	A	E	K	L	H	N	M	B	P	**O**
B	G	M	O	I	F	P	C	A	**D**	E	J	N	K	H	L
F	K	I	A	P	D	O	N	H	C	L	B	J	M	E	G
G	**C**	N	B	J	L	E	A	O	F	D	M	H	P	K	I
J	H	D	E	M	C	B	I	P	N	G	K	L	**A**	O	F
O	L	P	**M**	F	G	H	K	I	E	J	A	D	C	B	N
D	O	A	C	L	P	I	F	M	K	N	G	E	H	**J**	B
N	M	E	G	K	A	J	H	L	P	B	D	O	F	I	C
K	P	B	J	O	M	C	G	F	H	I	E	A	L	N	D
H	**I**	L	F	B	E	N	D	J	A	O	C	P	**G**	M	K

2 Solving Sudoku puzzles and making new ones

I have never seen the attraction of solving Sudoku puzzles *per se*, but the more abstract programming problem of devising and implementing an effective algorithm for doing so I find much more interesting.

The first `R` package on `CRAN` to offer a programming solution is the `sudoku` package, due to David Brahm and Greg Snow, with contributions from Curt Seeliger and Henrik Bengtsson. It offered an ingenious iterative solution to the problem, (which I found difficult to follow), mainly for the standard case of $n = 3$.

This led me to consider the problem more actively. It seemed to me an explicitly *recursive* solution would offer more elegant code without too much overhead cost in computing time. I am not sure the result is all that elegant, but it does seem to work reasonably effectively.

Cases $n = 2$ or $n = 3$ are generally easy; the cases $n = 4$ or $n = 5$ are not practical to solve in general, if the game is set up in the same form as a typical puzzle, but curiously it is possible to generate games, and hence complete Sudoku *designs*, by a technique outlined below. Cases for $n > 5$ require a more sophisticated algorithm than the one given in the present `sudokuAlt` package.

2.1 The solution algorithm

In outline the solution method used here for a Sudoku game is as follows. It uses what I suspect is the standard way people do so by hand.

Given a game,

- Check to see if there are any gaps yet to be filled. If not, the task is complete, return the completed game and exit.
- For each gap in the game, establish a list of all possible symbols that may, at this stage, be considered as a possible entry.
- If any gap has no possible entries the game is insoluble. In this case return `NULL`, indicating "no solution found".
- If all gaps have possible entries, select one with fewest possible completions. Loop over these possible entries, setting each possibility in turn as the entry, checking to see if a solution can be found by calling the procedure recursively with the current temporary entry fixed. If the loop is complete without a solution being found, the game is inconsistent and `NULL` is returned. If a solution to the entire problem is found, it will be returned prior to the loop ending.

In essence, "find the possible completion symbols, fix one and look again", but working systematically in such a way as ensure the process terminates, one way or another. The problem is mainly a curiosity but the programming strategy is of some possible pedagogical value, at least.

2.2 Making Sudoku designs (or puzzles)

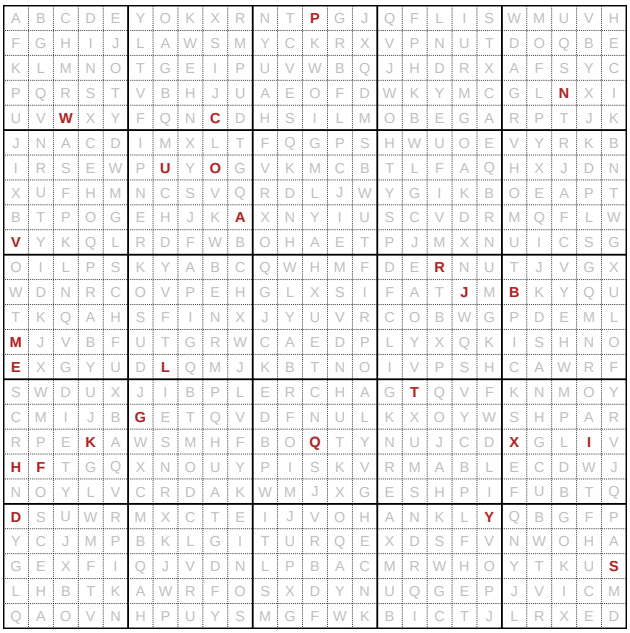
An obvious strategy to try to make a new Sudoku design (or puzzle) is

- Start with a design with all cells vacant
- Assign a single replicate of the n^2 symbols to the cells at random, giving Sudoku puzzle,
- Solve the puzzle to give a complete design.

While this is an obvious algorithm, what came as a surprise to me is just how well it works, at least for small-n cases. It works well for $n = 2, 3$, fairly well for $n = 4$ and with difficulty for $n = 5$.

Here is an example.

```
set.seed(1559707151)
g5 <- seedGame(5) %>% solve() %>% regulariseGame()
plot(g5, cex = 1)
```



The function `seedGame()` constructs the embryonic puzzle, which is denoted by the red symbols in the display.

If the Sudoku game is to be used formally as an experimental design, it is convenient to have the information in `data.frame` form rather than as a `"sudoku"` object. This conversion is achieved by the `designGame()` function:

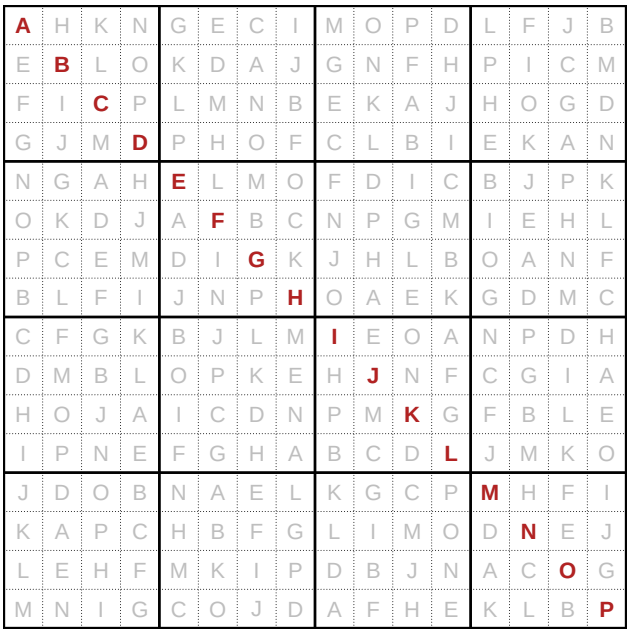
```
d5 <- designGame(g5)
head(d5); tail(d5)
    Row Col Square Symbol
001 R01 C01    S11      A
002 R01 C02    S11      B
003 R01 C03    S11      C
004 R01 C04    S11      D
005 R01 C05    S11      E
006 R02 C01    S11      F
    Row Col Square Symbol
620 R24 C25    S55      M
621 R25 C21    S55      L
622 R25 C22    S55      R
623 R25 C23    S55      X
624 R25 C24    S55      E
625 R25 C25    S55      D
```

2.3 Prescribed patterns

The package also contains a function `emptyGame(n)` that supplies a Sudoku object with all entries unfilled. This has been convenient for exploring Sudoku designs with prescribed additional constraints or patterns.

For example, it is possible to devise a 4^4 Sudoku pattern with the additional property that the leading diagonal is also a complete replicate of the 16 symbols:

```
g <- emptyGame(4)
diag(g) <- LETTERS[1:16]
g %>%
  solve() %>%
  plot() -> sg
```



There are no functions for the input of unsolved puzzles, but this `emptyGame` facility might help with manual input.

2.4 `print` and `plot` methods

The package gives some attention to reasonably comprehensible presentation of Sudoku puzzles and designs.

The `plot` method is shown above. It works reasonably well but may require some adjustment of the `cex` graphics parameter. It has the advantage of being able to show the puzzle and its solution separately on the one diagram.

The `print` method is shown below. It can only show either the puzzle or the solution, of course.

```
g <- emptyGame(3)
g[1:3, 1:3] <- matrix(1:9, nrow = 3, byrow = TRUE)
solve(g)
```

```
 + - - - + - - - + - - - +
 | 1 2 3 | 4 9 5 | 6 7 8 |
 | 4 5 6 | 7 8 2 | 3 9 1 |
 | 7 8 9 | 1 6 3 | 4 2 5 |
 + - - - + - - - + - - - +
 | 2 6 1 | 8 7 9 | 5 3 4 |
 | 3 7 4 | 5 1 6 | 9 8 2 |
 | 5 9 8 | 2 3 4 | 1 6 7 |
 + - - - + - - - + - - - +
 | 6 1 2 | 9 4 7 | 8 5 3 |
 | 8 3 5 | 6 2 1 | 7 4 9 |
 | 9 4 7 | 3 5 8 | 2 1 6 |
 + - - - + - - - + - - - +
```

Both `plot` and `print` methods return the game itself (invisibly).

2.5 Input and frivolities

As mentioned above, not much effort has been given to input of particular Sudoku puzzles, but the coercion function, `as.sudoku()` can be useful in this regard. It takes a square matrix as its input and returns a Sudoku object that the methods of the package can recognize.

The `emptyGame()` function could also be used for input, as mentioned above.

There are two functions which can access public websites to get daily Sudoku puzzles. These are

- `fetchAUGame()` which uses an Australian web site, and
- `fetchUKGame()` which uses a British web site.

This possibly explains why the package is listed as being for ***spoiling*** Sudoku puzzles.