Sudoku

Raphaël Morsomme

2019-01-07

Contents

1 Introduction

This is a short script showing how to solve a sudoku on `R`. It turns out that by following fairly simple principles, we can solve most grids in a few steps.

2 Representations

The matrix `grid` and the array `poss` are the building blocks of the script.

2.1 `grid`

`grid` is a numerical matrix representing the sudoku grid. Its rows and columns correspond to the rows and columns of the sudoku grid. The entries of `grid` indicate the value of the corresponding cell in the sudoku. Empty cells are represented with a `0`. The function `create_grid` helps us create the object `grid` from an existing sudoku grid in a simple way.

```r
# Empty sudoku grid
grid <- matrix(0, ncol = 9, nrow = 9, dimnames = list(1:9, 1:9))
print(grid)
```

```
##   1 2 3 4 5 6 7 8 9
## 1 0 0 0 0 0 0 0 0 0
## 2 0 0 0 0 0 0 0 0 0
## 3 0 0 0 0 0 0 0 0 0
## 4 0 0 0 0 0 0 0 0 0
## 5 0 0 0 0 0 0 0 0 0
## 6 0 0 0 0 0 0 0 0 0
## 7 0 0 0 0 0 0 0 0 0
## 8 0 0 0 0 0 0 0 0 0
## 9 0 0 0 0 0 0 0 0 0
```

```r
# Function to create grid
create_grid <- function(x)  matrix(x, ncol =  9, nrow = 9, byrow = T, dimnames = list(1:9, 1:9))

# Sudoku grid from english Wikipedia page on sudoku
wiki <- c(5,3,0,0,7,0,0,0,0,
          6,0,0,1,9,5,0,0,0,
          0,9,8,0,0,0,0,6,0,
          8,0,0,0,6,0,0,0,3,
          4,0,0,8,0,3,0,0,1,
          7,0,0,0,2,0,0,0,6,
          0,6,0,0,0,0,2,8,0,
          0,0,0,4,1,9,0,0,5,
          0,0,0,0,8,0,0,7,9)

grid <- create_grid(wiki)
print(grid)
```

```
##   1 2 3 4 5 6 7 8 9
## 1 5 3 0 0 7 0 0 0 0
## 2 6 0 0 1 9 5 0 0 0
## 3 0 9 8 0 0 0 0 6 0
## 4 8 0 0 0 6 0 0 0 3
## 5 4 0 0 8 0 3 0 0 1
## 6 7 0 0 0 2 0 0 0 6
## 7 0 6 0 0 0 0 2 8 0
## 8 0 0 0 4 1 9 0 0 5
## 9 0 0 0 0 8 0 0 7 9
```

2.2 poss

poss is a 3-dimensional logical array indicating the values with which we could fill the cells of the sudoku grid. Its rows (first dimension) and columns (second dimension) correspond to the rows and column of the sudoku grid, and its layers (third dimension) to the 9 possible value (1, 2, 3, 4, 5, 6, 7, 8, 9) each cell can potential take (excluding 0). The entry $poss_{i,j,n}$ (logial) of poss indicates whether the number n could be used to fill the cell on row i and column j of the sudoku grid. We create the function create_poss for convenience.

```
create_poss <- function() array(T, dim = c(9,9,9), dimnames = list(1:9, 1:9, 1:9))
poss <- create_poss()
# First layer of `poss`: which cells could we fill with a `1`
print(poss[ , , 1])
```

```
##      1    2    3    4    5    6    7    8    9
## 1 TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
## 2 TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
## 3 TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
## 4 TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
## 5 TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
## 6 TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
## 7 TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
## 8 TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
## 9 TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

3 Solving a Sudoku

3.1 Pseudo-code

To solve a sudoku, we iteratively update grid and poss until grid has no empty cell.

```
# Pseudo-code for solving a sudoku
#   while(any(grid == 0)){
#     poss <- update_poss(grid, poss)
#     grid <- update_grid(grid, poss)
#   }
```

3.2 Updating poss

We start with poss which we update in 4 different ways: cell-wise, row-wise column-wise and box-wise[1].

3.2.1 Cell by cell

In a sudoku, a cell can only have one value. Therefore, if a cell $cell_{i,j}$ is filled with a value, then no other value can be used to fill it. Consequently, we set the entries of poss that correspond to $cell_{i,j}$ to FALSE.

[1]boxes are the three-by-three subgrids of the main sudoku grid.

In other words, if $cell_{i,j} \neq 0$, then $poss_{i,j,n} = FALSE, \forall n$. The function `update_poss_cell` accomplishes this. It first finds the locations of nonempty cells and assigns them to `non_empty` before updating `poss` and returning `poss`. (We use `rep()` on `non_empty` to match the dimensions of `poss`).

```
update_poss_cell <- function(grid, poss){

  # Location of nonempty cells
  non_empty       <- grid != 0
  non_empty_array <- array(rep(non_empty, 9), dim = c(9,9,9))

  # Updating `poss`
  poss <- poss & !non_empty_array

  return(poss)
}

poss <- create_poss()
poss <- update_poss_cell(grid, poss)
# Some cells have a value (different from zero)
print(grid)
```

```
##   1 2 3 4 5 6 7 8 9
## 1 5 3 0 0 7 0 0 0 0
## 2 6 0 0 1 9 5 0 0 0
## 3 0 9 8 0 0 0 0 6 0
## 4 8 0 0 0 6 0 0 0 3
## 5 4 0 0 8 0 3 0 0 1
## 6 7 0 0 0 2 0 0 0 6
## 7 0 6 0 0 0 0 2 8 0
## 8 0 0 0 4 1 9 0 0 5
## 9 0 0 0 0 8 0 0 7 9
```

```
# Entries corresponding to cells with a value are `FALSE`.
print(poss[ , , 1])
```

```
##       1     2     3     4     5     6     7     8     9
## 1 FALSE FALSE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE
## 2 FALSE  TRUE  TRUE FALSE FALSE FALSE  TRUE  TRUE  TRUE
## 3  TRUE FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE
## 4 FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE FALSE
## 5 FALSE  TRUE  TRUE FALSE  TRUE FALSE  TRUE  TRUE FALSE
## 6 FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE FALSE
## 7  TRUE FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE  TRUE
## 8  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE  TRUE FALSE
## 9  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE FALSE FALSE
```

3.2.2 Row by row, column by column and box by box

We know that a number can only appear once per row/column/box. Therefore, if a cell $cell_{i,j} = n \neq 0$, then n cannot be used to fill any other cell of $cell_{i,j}$'s row, column and box. Consequently, we set the entries of poss corresponding to $cell_{i,j}$'s row, column and box to `FALSE`. The function `update_poss_row`, `update_poss_col` and `update_poss_box` accomplish this by looping through each row/column/box of the grid. In each loop, we first isolate the row/column/box under investigation which we assign to `row_grid`, `col_grid` and `box_grid`. We then loop through the values 1, 2, 3, 4, 5, 6, 7, 8, 9. In this second loop, we check if the value under investigation n is present in any of the cells of the row/column/box under investigation. If this is the case, we update the corresponding entries of poss to FALSE.

```
#
# Row by row
update_poss_row <- function(grid, poss){
  for(row in 1:9){
    row_grid <- grid[row, ]
    for(n in 1:9){
      # Check if n is present in row
      if(any(row_grid == n)) poss[row, , n] <- FALSE
    }
  }
  return(poss)
}

poss <- create_poss()
poss <- update_poss_row(grid, poss)
# Sudoku grid
print(grid)
```

```
##   1 2 3 4 5 6 7 8 9
## 1 5 3 0 0 7 0 0 0 0
## 2 6 0 0 1 9 5 0 0 0
## 3 0 9 8 0 0 0 0 6 0
## 4 8 0 0 0 6 0 0 0 3
## 5 4 0 0 8 0 3 0 0 1
## 6 7 0 0 0 2 0 0 0 6
## 7 0 6 0 0 0 0 2 8 0
## 8 0 0 0 4 1 9 0 0 5
## 9 0 0 0 0 8 0 0 7 9
```

```
# Rows containing a `1` are `FALSE` on layer `1` of `poss`
print(poss[ , , 1])
```

```
##       1     2     3     4     5     6     7     8     9
## 1  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## 2 FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## 3  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## 4  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## 5 FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## 6  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## 7  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## 8 FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## 9  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

```
#
# Column by column
update_poss_col <- function(grid, poss){
  for(col in 1:9){
    col_grid <- grid[ , col]
    for(n in 1:9){
      # Check if n is present in column
      if(any(col_grid == n)) poss[ , col, n] <- FALSE
    }
  }
  return(poss)
}
```

```
#
# Box by box
update_poss_box <- function(grid, poss){
  for(col_box in 1 : 3){
    for(row_box in 1 : 3){
      rows <- 1 : 3 + 3 * (row_box-1)
      cols <- 1 : 3 + 3 * (col_box-1)
      box_grid <- grid[rows, cols]
      for(n in 1 : 9){
        # Check if n is present in the box
        if(any(box_grid == n)) poss[rows, cols, n] <- FALSE
      }
    }
  }
  return(poss)
}
```

3.2.3 `update_poss()`

We encapsulate these four updating functions in `update_poss` for convenience.

```
update_poss <- function(grid, poss){
  poss <- update_poss_cell(grid, poss)
  poss <- update_poss_row(grid, poss)
  poss <- update_poss_col(grid, poss)
  poss <- update_poss_box(grid, poss)
  return(poss)
}

poss <- create_poss()
poss <- update_poss(grid, poss)
# Sudoku grid
print(grid)
```

```
##   1 2 3 4 5 6 7 8 9
## 1 5 3 0 0 7 0 0 0 0
## 2 6 0 0 1 9 5 0 0 0
## 3 0 9 8 0 0 0 0 6 0
## 4 8 0 0 0 6 0 0 0 3
## 5 4 0 0 8 0 3 0 0 1
## 6 7 0 0 0 2 0 0 0 6
## 7 0 6 0 0 0 0 2 8 0
## 8 0 0 0 4 1 9 0 0 5
## 9 0 0 0 0 8 0 0 7 9
```

```
# filled cells and rows, columns and boxes containing a `1` are `FALSE` on layer `1`
print(poss[ , , 1])
```

```
##       1     2     3     4     5     6     7     8     9
## 1 FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE  TRUE FALSE
## 2 FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## 3  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
## 4 FALSE  TRUE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE
## 5 FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## 6 FALSE  TRUE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE
## 7  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
## 8 FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## 9  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE
```

3.3 Updating `grid`

Now that we have updated the array `poss`, we can use it to update `grid`. We do this in a similar way: cell-wise, row-wise, column-wise and box-wise. The structure of the code is very similar to that of the code we use to update `poss`. In the function `update_grid_cell`, we check, for each cell (loop), how many values are still possible. If there is only one value possible, then we fill the cell with this value. In the functions `update_grid_row`, `update_grid_col` and `update_grid_box`, we identify, for each value 1, 2, 3, 4, 5, 6, 7, 8, 9 (loop), their possible locations in the row/column/box. If there is only one location possible, then we fill the corresponding cell of the grid with the value in question.

3.3.1 Cell by cell

```
update_grid_cell <- function(grid, poss){
  for(row in 1 : 9){
    for(col in 1 : 9){
      # Check if cell is empty
      if(grid[row, col] == 0){
        values_left <- poss[row, col, ]
        # check if only one value left to complete empty cell
        if(sum(values_left) == 1)  grid[row, col] <- c(1:9)[values_left]
      }
    }
  }
  return(grid)
}

grid_update <- update_grid_cell(grid, poss)
# grid before update
print(grid)
```

```
##   1 2 3 4 5 6 7 8 9
## 1 5 3 0 0 7 0 0 0 0
## 2 6 0 0 1 9 5 0 0 0
## 3 0 9 8 0 0 0 0 6 0
## 4 8 0 0 0 6 0 0 0 3
## 5 4 0 0 8 0 3 0 0 1
## 6 7 0 0 0 2 0 0 0 6
## 7 0 6 0 0 0 0 2 8 0
## 8 0 0 0 4 1 9 0 0 5
## 9 0 0 0 0 8 0 0 7 9
```

```
# grid after update
print(grid_update)
```

```
##   1 2 3 4 5 6 7 8 9
## 1 5 3 0 0 7 0 0 0 0
## 2 6 0 0 1 9 5 0 0 0
## 3 0 9 8 0 0 0 0 6 0
## 4 8 0 0 0 6 0 0 0 3
## 5 4 0 0 8 5 3 0 0 1
## 6 7 0 0 0 2 0 0 0 6
## 7 0 6 0 0 0 7 2 8 4
## 8 0 0 0 4 1 9 0 3 5
## 9 0 0 0 0 8 0 0 7 9
```

```
# 4 entries are updated {(5,5), (7,6), (7,9), (8,8)}
sum(grid_update != grid)
```

```
## [1] 4
```

3.3.2 Row by row, column by column and box by box

```
update_grid_row <- function(grid, poss){
  for(row in 1 : 9){
    for(n in 1 : 9){
      locations <- poss[row, , n]
      # Check if only one possible location for value `n` in row
      if(sum(locations) == 1) grid[row, locations] <- n
    }
  }
  return(grid)
}
```

```
update_grid_col <- function(grid, poss){
  for(col in 1 : 9){
    for(n in 1 : 9){
      locations <- poss[ , col, n]
      # Check if only one possible location for value `n` in column
      if(sum(locations) == 1) grid[locations, col] <- n
    }
  }
  return(grid)
}
```

```
update_grid_box <- function(grid, poss){
  for(col_box in 1 : 3){
    for(row_box in 1 : 3){
      rows <- 1 : 3 + 3 * (row_box-1)
      cols <- 1 : 3 + 3 * (col_box-1)
      box  <- grid[rows, cols]
      for(n in 1 : 9){
        locations <- poss[rows, cols, n]
        # Check if only one possible location for value `n` in box
        if(sum(locations) == 1) grid[rows, cols][locations] <- n
      }
    }
  }
  return(grid)
}
```

3.3.3 `update_grid()`

We encapsulate these four updating functions in `update_grid` for convenience.

```r
update_grid <- function(grid, poss){
  grid <- update_grid_cell(grid, poss)
  grid <- update_grid_row(grid, poss)
  grid <- update_grid_col(grid, poss)
  grid <- update_grid_box(grid, poss)
  return(grid)
}

grid_update <- update_grid(grid, poss)
# grid before update
print(grid)
```

```
##   1 2 3 4 5 6 7 8 9
## 1 5 3 0 0 7 0 0 0 0
## 2 6 0 0 1 9 5 0 0 0
## 3 0 9 8 0 0 0 0 6 0
## 4 8 0 0 0 6 0 0 0 3
## 5 4 0 0 8 0 3 0 0 1
## 6 7 0 0 0 2 0 0 0 6
## 7 0 6 0 0 0 0 2 8 0
## 8 0 0 0 4 1 9 0 0 5
## 9 0 0 0 0 8 0 0 7 9
```

```r
# grid after update
print(grid_update)
```

```
##   1 2 3 4 5 6 7 8 9
## 1 5 3 0 0 7 8 0 1 0
## 2 6 0 0 1 9 5 0 0 0
## 3 0 9 8 0 4 0 5 6 0
## 4 8 0 0 0 6 0 0 0 3
## 5 4 0 6 8 5 3 7 0 1
## 6 7 0 3 0 2 0 8 0 6
## 7 9 6 0 0 0 7 2 8 4
## 8 0 8 0 4 1 9 6 3 5
## 9 0 0 0 0 8 0 1 7 9
```

```r
# 16 cells are updated
sum(grid != grid_update)
```

```
## [1] 16
```

```r
# There are still 35 empty cells...
sum(grid_update == 0)
```

```
## [1] 35
```

3.4 Solving our first sudoku

Now that we are equipped with `update_poss` and `update_grid`, we can write a function that iteratively updates `poss` and `grid` until the grid is complete. We add a safeguard in our function to avoid the loop to continue for ever in case `update_grid` fails to find a cell to update (which can happen for difficult grids, in which case we need to one of the numbers, see following section)

```
solve_sudoku <- function(grid){

  poss <- update_poss(grid, create_poss())

  # Loop
  while(any(grid == 0)){
    print(paste(sum(grid == 0), "empty cells left." ))
    grid_update <- update_grid(grid, poss)
    # safeguard: if algorithm stuck, stop it
    if(all(grid == grid_update)) return(print("Algorithm stuck"))
    grid <- grid_update
    poss <- update_poss(grid, poss)
  }

  # Output
  print("Grid solved")
  return(grid)

}
```

Let us solve the sudoku from the Wikipedia page with our newly created function `solve_sudoku`.

```
# Solving our first grid
grid           <- create_grid(wiki)
solve_sudoku(grid)
```

```
## [1] "51 empty cells left."
## [1] "35 empty cells left."
## [1] "14 empty cells left."
## [1] "5 empty cells left."
## [1] "1 empty cells left."
## [1] "Grid solved"
```

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

Let us try a more complex case: a sudoku with only 17 starting numbers (minimum number of starting cues to have a unique solution).

```
# Sparsest sudoku possible? No problem!
x <- c(0,0,0,0,0,0,0,1,0,
       0,0,0,0,0,2,0,0,3,
       0,0,0,4,0,0,0,0,0,
       0,0,0,0,0,0,5,0,0,
       4,0,1,6,0,0,0,0,0,
       0,0,7,1,0,0,0,0,0,
       0,5,0,0,0,0,2,0,0,
       0,0,0,0,8,0,0,4,0,
       0,3,0,9,1,0,0,0,0)
# Only 17 starting numbers
sum(x != 0)
```

```
## [1] 17
```

```
grid_sparse <- create_grid(x)
solve_sudoku(grid_sparse)
```

```
## [1] "64 empty cells left."
## [1] "59 empty cells left."
## [1] "55 empty cells left."
## [1] "51 empty cells left."
## [1] "46 empty cells left."
## [1] "43 empty cells left."
## [1] "42 empty cells left."
## [1] "39 empty cells left."
## [1] "33 empty cells left."
## [1] "27 empty cells left."
## [1] "23 empty cells left."
## [1] "18 empty cells left."
## [1] "14 empty cells left."
## [1] "8 empty cells left."
## [1] "3 empty cells left."
## [1] "Grid solved"
```

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

It takes more iterations, but our relatively simple algorithm solves the grid nonetheless. Impressive!

4 Algorithm stuck? Take a guess!

For difficult grids, the function `update_grid` mays fail to find a cell to fill with a value. In such case, we need to *guess* the value of one of the cells.

```
# Grid with "evil" level from https://www.websudoku.com/?level=4&set_id=4360842130
x <- c(0,1,0,0,4,5,0,0,0,
       0,0,0,0,0,0,7,0,6,
       0,0,5,2,0,0,0,0,4,
       0,9,0,0,7,0,0,8,2,
       0,0,0,6,0,1,0,0,0,
       4,8,0,0,3,0,0,7,0,
       8,0,0,0,0,9,4,0,0,
       7,0,9,0,0,0,0,0,0,
       0,0,0,3,5,0,0,6,0)
grid_evil <- create_grid(x)
# Algorithm fails to update any cell on the 5th iteration.
solve_sudoku(grid_evil)
```

```
## [1] "55 empty cells left."
## [1] "50 empty cells left."
## [1] "47 empty cells left."
## [1] "46 empty cells left."
## [1] "Algorithm stuck"
```

4.1 Taking a guess

The function `guess` guesses the value of an empty cell. We use the function when our algorithm is stuck. To minimze the risk of guessing a wrong value, we find a cell with the smallest number of possible values (ideally only 2 possible values). First, we apply `rowSums` on `poss` to obtain the number of possible values for each cell which we assign to `n_possible`. We then assign the minimum of `n_possible` (excluding non empty cells, since they have zero possible values) to `n_min`. Next, we loop through the cells until we find one that is empty and has a number of possible values equal to `n_min`. When we find such cell, we identify the possible values (`values_possible`), choose one at random (`guess`) and fill the cell with it. We then immediatly return the grid to avoid making additional guesses which would increase the risk of making an error.

```
guess <- function(grid, poss){

  # Number of possible values per cell
  n_possible <- rowSums(poss, dims = 2)
  # find minimum number of possible value among empty cells
  n_possible_empty <- n_possible[grid == 0]
  n_min            <- min(n_possible_empty)

  for(row in 1 : 9){
    for(col in 1 : 9){
      # find an empty cell with the minimum number of possible values
      if(grid[row, col] == 0  &  sum(poss[row, col, ]) == n_min){
        values_possible <- c(1:9)[poss[row, col, ]]
        guess           <- sample(values_possible, size = 1)
        grid[row, col]  <- guess
        return(grid)
      }
    }
  }
}
```

4.2 Checking a grid

Guessing a number in the grid opens the door to errors. After we guess, we must check that the grid does not contain an error. Let us design a function `check_grid` that checks if a sudoku grid is correct, using two criteria:

1. each empty cell must have at least one possible value
2. each number that is not present in a row/column/box must have at least one possible location in the row/column/box

The code of `check_grid` is very similar to code previously used. If the grid satisfies the two conditions, then the function returns a `TRUE`, otherwise, a `FALSE`.

```
check_grid <- function(grid, poss){

  # Check that each empty cell has at least one possible value
  for(row in 1 : 9){
    for(col in 1 : 9){
      # If cell is empty and there is no possible value, then grid contains an error.
      if(grid[row, col] == 0  &  all(poss[row, col, ] == FALSE))  return(FALSE)
    }
  }

  # Check that each number absent from a row/column/box
  # has at least one possible location in the row/column/box
  for(row in 1:9){ # row
    row_grid <- grid[row, ]
    for(n in 1:9){
      # If value absent from row and there is no location left, then grid contains an error.
      if(all(row_grid != n)  &  all(poss[row, , n] == FALSE))  return(FALSE)
    }
  }

  for(col in 1:9){ # col
    col_grid <- grid[ , col]
    for(n in 1:9){
      # If value absent from column and there is no location left, then grid contains an error.
      if(all(col_grid != n)  &  all(poss[ , col, n] == FALSE))  return(FALSE)
    }
  }

  for(col_box in 1 : 3){ # box
    for(row_box in 1 : 3){
      rows     <- 1 : 3 + 3 * (row_box-1)
      cols     <- 1 : 3 + 3 * (col_box-1)
      box_grid <- grid[rows, cols]
      for(n in 1:9){
        # If value absent from box and there is no location left, then grid contains an error.
        if(all(box_grid != n)  &  all(poss[rows, cols, n] == FALSE))  return(FALSE)
      }
    }
  }

  # if all conditions are satisfied, the grid is ok
  return(TRUE)
}
```

4.3 Solving Sudoku (with guess)

Let us include the functions `guess` and `check_grid` to our function `solve_sudoku` and try it on the "evil" grid.

```
solve_sudoku <- function(grid){
```

```r
  # Setup
  poss          <- update_poss(grid, poss = create_poss())
  grid_original <- grid # save grid in case we make a wrong guess
  n_wrong_guess <- 0    # keep track of number of wrong guesses
  has_guessed   <- FALSE

  # Loop
  while(any(grid == 0)){

    # Update grid and poss
    grid_update <- update_grid(grid, poss)
    if(all(grid == grid_update)){# If no cell is updated, make a guess
      grid_update <- guess(grid, poss)
      has_guessed <- TRUE
    }
    poss <- update_poss(grid_update, poss)

    # if grid_update contains an error, start over; otherwise, continue with next iteration
    if(!check_grid(grid_update, poss)){
      grid <- grid_original
      poss <- update_poss(grid, poss = create_poss())
      n_wrong_guess <- n_wrong_guess + 1
    }else{
      grid <- grid_update
    }

    # Safeguard
    if(n_wrong_guess >= 100) return(print("Too many guesses: impossible grid"))

  } # end while-loop

  # Output
  if(has_guessed) print(paste("Grid solved after", n_wrong_guess, "wrong guesses."))
  else            print("Grid solved without guess")
  return(grid)

}
```

Our updated version of `solve_sudoku` solves the “evil” grid seamlessly. In fact, I have tried to solve numerous grids with the algorithm and always succeeded. In the worst case, the algorithm makes a few wrong guesses (at most five) before solving the grid.

```r
set.seed(123)
x <- replicate(n = 10, solve_sudoku(grid_evil))
```

```
## [1] "Grid solved after 1 wrong guesses."
## [1] "Grid solved after 1 wrong guesses."
## [1] "Grid solved after 0 wrong guesses."
## [1] "Grid solved after 1 wrong guesses."
## [1] "Grid solved after 0 wrong guesses."
## [1] "Grid solved after 0 wrong guesses."
## [1] "Grid solved after 1 wrong guesses."
## [1] "Grid solved after 1 wrong guesses."
## [1] "Grid solved after 0 wrong guesses."
## [1] "Grid solved after 1 wrong guesses."
```

4.4 Testing the algorithm: extreme cases

Surprisingly, even if the grid is empty, our algorithm manages to generate a valid solution very quickly. This indicates that the algorithm does not struggle to solve soduko grids that are sparse.

```
# Empty grid
grid_empty <- create_grid(0)
set.seed(123)
x <- replicate(n = 10, solve_sudoku(grid_empty))
```

```
## [1] "Grid solved after 0 wrong guesses."
## [1] "Grid solved after 0 wrong guesses."
## [1] "Grid solved after 0 wrong guesses."
## [1] "Grid solved after 0 wrong guesses."
## [1] "Grid solved after 0 wrong guesses."
## [1] "Grid solved after 0 wrong guesses."
## [1] "Grid solved after 0 wrong guesses."
## [1] "Grid solved after 0 wrong guesses."
## [1] "Grid solved after 0 wrong guesses."
## [1] "Grid solved after 0 wrong guesses."
```

In case we feed an impossible grid to the algorithm, the safeguard stops the loop.

```
# Impossible grid
grid_impossible <- create_grid(c(1,1, rep(0,79)))
print(grid_impossible)
```

```
##   1 2 3 4 5 6 7 8 9
## 1 1 1 0 0 0 0 0 0 0
## 2 0 0 0 0 0 0 0 0 0
## 3 0 0 0 0 0 0 0 0 0
## 4 0 0 0 0 0 0 0 0 0
## 5 0 0 0 0 0 0 0 0 0
## 6 0 0 0 0 0 0 0 0 0
## 7 0 0 0 0 0 0 0 0 0
## 8 0 0 0 0 0 0 0 0 0
## 9 0 0 0 0 0 0 0 0 0
```

```
set.seed(123)
solve_sudoku(grid_impossible)
```

```
## [1] "Too many guesses: impossible grid"
```

5 Conclusion

In this script, I develop an algorithm to solve sudoku grids based on two principles: (i) if an empty cell has only one possible value left, write the value in the cell, and (ii) if a row/column/box has only one cell left for a value, write the value in the cell. Despite its simplicity, the algorithm can solve most sudoku grids, even the most sparse one!

Yet, for some grids, the algorithm gets stuck, in which case we need to guess the value of a cell. I updated the original algorithm so that it makes a guess when it gets stuck and checks the validity of the grid after the guess. This version of the algorithm solves even the most difficult grid after at most a few iterations.

This shows that sudoku grids can be solved with very simple principles; in the worst cases, we just have to effectuate one or two guesses. Who knew sudoku was so simple?