Solving Every Sudoku Puzzle

by Peter Norvig

In this essay I tackle the problem of solving every Sudoku puzzle. It turns out to be quite easy (about one page of code for the main idea and two pages for embellishments) using two ideas: **constraint propagation** and **search**.

Sudoku Notation and Preliminary Notions

First we have to agree on some notation. A Sudoku puzzle is a grid of 81 squares; the majority of enthusiasts label the columns 1-9, the rows A-I, and call a collection of nine squares (column, row, or box) a unit and the squares that share a unit the peers. A puzzle leaves some squares blank and fills others with digits, and the whole idea is:

> A puzzle is solved if the squares in each unit are filled with a permutation of the digits 1 to 9.

That is, no digit can appear twice in a unit, and every digit must appear once. This implies that each square must have a different value from any of its peers. Here are the names of the squares, a typical puzzle, and the solution to the puzzle:

```
 A1 A2 A3| A4 A5 A6| A7 A8 A9    4 . . |. . . |8 . 5     4 1 7 |3 6 9 |8 2 5
 B1 B2 B3| B4 B5 B6| B7 B8 B9    . 3 . |. . . |. . .     6 3 2 |1 5 8 |9 4 7
 C1 C2 C3| C4 C5 C6| C7 C8 C9    . . . |7 . . |. . .     9 5 8 |7 2 4 |3 1 6
---------+---------+---------    ------+------+------    ------+------+------
 D1 D2 D3| D4 D5 D6| D7 D8 D9    . 2 . |. . . |. 6 .     8 2 5 |4 3 7 |1 6 9
 E1 E2 E3| E4 E5 E6| E7 E8 E9    . . . |. 8 . |4 . .     7 9 1 |5 8 6 |4 3 2
 F1 F2 F3| F4 F5 F6| F7 F8 F9    . . . |. 1 . |. . .     3 4 6 |9 1 2 |7 5 8
---------+---------+---------    ------+------+------    ------+------+------
 G1 G2 G3| G4 G5 G6| G7 G8 G9    . . . |6 . 3 |. 7 .     2 8 9 |6 4 3 |5 7 1
 H1 H2 H3| H4 H5 H6| H7 H8 H9    5 . . |2 . . |. . .     5 7 3 |2 9 1 |6 8 4
 I1 I2 I3| I4 I5 I6| I7 I8 I9    1 . 4 |. . . |. . .     1 6 4 |8 7 5 |2 9 3
```

Every square has exactly 3 units and 20 peers. For example, here are the units and peers for the square `C2`:

```
    A2   |         |                    |         |            A1 A2 A3|         |
    B2   |         |                    |         |            B1 B2 B3|         |
    C2   |         |            C1 C2 C3| C4 C5 C6| C7 C8 C9   C1 C2 C3|         |
---------+---------+---------  ---------+---------+---------  ---------+---------+---------
    D2   |         |                    |         |                    |         |
    E2   |         |                    |         |                    |         |
    F2   |         |                    |         |                    |         |
---------+---------+---------  ---------+---------+---------  ---------+---------+---------
    G2   |         |                    |         |                    |         |
    H2   |         |                    |         |                    |         |
    I2   |         |                    |         |                    |         |
```

We can implement the notions of units, peers, and squares in the programming language Python (2.5 or later) as follows:

```
def cross(A, B):
    "Cross product of elements in A and elements in B."
    return [a+b for a in A for b in B]

digits   = '123456789'
rows     = 'ABCDEFGHI'
cols     = digits
squares  = cross(rows, cols)
unitlist = ([cross(rows, c) for c in cols] +
            [cross(r, cols) for r in rows] +
            [cross(rs, cs) for rs in ('ABC','DEF','GHI') for cs in ('123','456','789')])
units = dict((s, [u for u in unitlist if s in u])
             for s in squares)
peers = dict((s, set(sum(units[s],[]))-set([s]))
             for s in squares)
```

If you are not familiar with some of the features of Python, note that a `dict` or dictionary is Python's name for a hash table that maps each key to a value; that these are specified as a sequence of (key, value) tuples; that `dict((s, [...]) for s in squares)` creates a dictionary which maps each square `s` to a value that is the list `[...]`; and that the expression `[u for u in unitlist if s in u]` means that this value is the list of units `u` such that the square `s` is a member of `u`. So read this assignment statement as "`units` is a dictionary where each square maps to the list of units that contain the square". Similarly, read the next assignment statement as "`peers` is a dictionary where each square `s` maps to the set of squares formed by the union of the squares in the units of `s`, but not `s` itself".

It can't hurt to throw in some tests (they all pass):

```
def test():
    "A set of unit tests."
    assert len(squares) == 81
    assert len(unitlist) == 27
    assert all(len(units[s]) == 3 for s in squares)
    assert all(len(peers[s]) == 20 for s in squares)
    assert units['C2'] == [['A2', 'B2', 'C2', 'D2', 'E2', 'F2', 'G2', 'H2', 'I2'],
                           ['C1', 'C2', 'C3', 'C4', 'C5', 'C6', 'C7', 'C8', 'C9'],
                           ['A1', 'A2', 'A3', 'B1', 'B2', 'B3', 'C1', 'C2', 'C3']]
    assert peers['C2'] == set(['A2', 'B2', 'D2', 'E2', 'F2', 'G2', 'H2', 'I2',
                               'C1', 'C3', 'C4', 'C5', 'C6', 'C7', 'C8', 'C9',
                               'A1', 'A3', 'B1', 'B3'])
    print 'All tests pass.'
```

Now that we have squares, units, and peers, the next step is to define the Sudoku playing grid. Actually we need two representations: First, a textual format used to specify the initial state of a puzzle; we will reserve the name grid for this. Second, an internal representation of any state of a puzzle, partially solved or complete; this we will call a values collection because it will give all the remaining possible values for each square. For the textual format (grid) we'll allow a string of characters with 1-9 indicating a digit, and a 0 or period specifying an empty square. All other characters are ignored (including spaces, newlines, dashes, and bars). So each of the following three grid strings represent the same puzzle:

```
"4.....8.5.3..........7......2.....6.....8.4......1.......6.3.7.5..2.....1.4......"

"""
400000805
030000000
000700000
020000060
000080400
000010000
000603070
500200000
104000000"""

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

Now for values. One might think that a 9 x 9 array would be the obvious data structure. But squares have names like `'A1'`, not `(0,0)`. Therefore, values will be a dict with squares as keys. The value of each key will be the possible digits for that square: a single digit if it was given as part of the puzzle definition or if we have figured out what it must be, and a collection of several digits if we are still uncertain. This collection of digits could be represented by a Python `set` or `list`, but I chose instead to use a string of digits (we'll see why later). So a grid where `A1` is `7` and `C7` is empty would be represented as `{'A1': '7', 'C7': '123456789', ...}`.

Here is the code to parse a grid into a values dict:

```
def parse_grid(grid):
    """Convert grid to a dict of possible values, {square: digits}, or
    return False if a contradiction is detected."""
    ## To start, every square can be any digit; then assign values from the grid.
    values = dict((s, digits) for s in squares)
    for s,d in grid_values(grid).items():
        if d in digits and not assign(values, s, d):
            return False ## (Fail if we can't assign d to square s.)
    return values

def grid_values(grid):
    "Convert grid into a dict of {square: char} with '0' or '.' for empties."
    chars = [c for c in grid if c in digits or c in '0.']
```

```
    assert len(chars) == 81
    return dict(zip(squares, chars))
```

Constraint Propagation

The function `parse_grid` calls `assign(values, s, d)`. We could implement this as `values[s] = d`, but we can do more than just that. Those with experience solving Sudoku puzzles know that there are two important strategies that we can use to make progress towards filling in all the squares:

> (1) If a square has only one possible value, then eliminate that value from the square's peers.
> (2) If a unit has only one possible place for a value, then put the value there.

As an example of strategy (1) if we assign 7 to `A1`, yielding `{'A1':'7','A2':'123456789',...}`, we see that `A1` has only one value, and thus the 7 can be removed from its peer `A2` (and all other peers), giving us `{'A1':'7','A2':'12345689',...}`. As an example of strategy (2), if it turns out that none of `A3` through `A9` has a 3 as a possible value, then the 3 must belong in `A2`, and we can update to `{'A1':'7','A2':'3',...}`. These updates to `A2` may in turn cause further updates to its peers, and the peers of those peers, and so on. This process is called constraint propagation.

The function `assign(values, s, d)` will return the updated values (including the updates from constraint propagation), but if there is a contradiction--if the assignment cannot be made consistently--then `assign` returns `False`. For example, if a grid starts with the digits `'77...'` then when we try to assign the 7 to `A2`, `assign` would notice that 7 is not a possibility for `A2`, because it was eliminated by the peer, `A1`.

It turns out that the fundamental operation is not assigning a value, but rather eliminating one of the possible values for a square, which we implement with `eliminate(values, s, d)`. Once we have `eliminate`, then `assign(values, s, d)` can be defined as "eliminate all the values from `s` except `d`".

```
def assign(values, s, d):
    """Eliminate all the other values (except d) from values[s] and propagate.
    Return values, except return False if a contradiction is detected."""
    other_values = values[s].replace(d, '')
    if all(eliminate(values, s, d2) for d2 in other_values):
        return values
    else:
        return False

def eliminate(values, s, d):
    """Eliminate d from values[s]; propagate when values or places <= 2.
    Return values, except return False if a contradiction is detected."""
    if d not in values[s]:
        return values ## Already eliminated
    values[s] = values[s].replace(d,'')
    ## (1) If a square s is reduced to one value d2, then eliminate d2 from the peers.
    if len(values[s]) == 0:
      return False ## Contradiction: removed last value
    elif len(values[s]) == 1:
        d2 = values[s]
        if not all(eliminate(values, s2, d2) for s2 in peers[s]):
            return False
    ## (2) If a unit u is reduced to only one place for a value d, then put it there.
    for u in units[s]:
      dplaces = [s for s in u if d in values[s]]
      if len(dplaces) == 0:
          return False ## Contradiction: no place for this value
      elif len(dplaces) == 1:
          # d can only be in one place in unit; assign it there
            if not assign(values, dplaces[0], d):
                return False
    return values
```

Now before we can go much further, we will need to display a puzzle:

```
def display(values):
    "Display these values as a 2-D grid."
    width = 1+max(len(values[s]) for s in squares)
    line = '+'.join(['-'*(width*3)]*3)
    for r in rows:
        print ''.join(values[r+c].center(width)+('|' if c in '36' else '')
                      for c in cols)
        if r in 'CF': print line
    print
```

Now we're ready to go. I picked the first example from a list of easy puzzles from the fine Project Euler Sudoku problem and tried it:

```
>>> grid1 = '003020600900305001001806400008102900700000008006708200002609500800203009005010300'

>>> display(parse_grid(grid1))
4 8 3 |9 2 1 |6 5 7
9 6 7 |3 4 5 |8 2 1
2 5 1 |8 7 6 |4 9 3
------+------+------
5 4 8 |1 3 2 |9 7 6
7 2 9 |5 6 4 |1 3 8
1 3 6 |7 9 8 |2 4 5
------+------+------
3 7 2 |6 8 9 |5 1 4
8 1 4 |2 5 3 |7 6 9
6 9 5 |4 1 7 |3 8 2
```

In this case, the puzzle was completely solved by rote application of strategies (1) and (2)! Unfortunately, that will not always be the case. Here is the first example from a list of hard puzzles:

```
>>> grid2 = '4.....8.5.3..........7......2.....6.....8.4......1.......6.3.7.5..2.....1.4......'

>>> display(parse_grid(grid2))
   4      1679   12679  |  139     2369    269   |   8      1239     5
 26789     3    1256789 | 14589   24569   245689 | 12679    1249   124679
  2689   15689   125689 |   7     234569  245689 | 12369   12349   123469
------------------------+------------------------+------------------------
  3789     2     15789  |  3459   34579    4579  | 13579     6     13789
  3679   15679   15679  |  359      8     25679  |   4     12359   12379
 36789     4     56789  |  359      1     25679  | 23579   23589   23789
------------------------+------------------------+------------------------
  289      89     289   |   6      459      3    |  1259     7     12489
   5      6789     3    |   2      479      1    |   69     489     4689
   1      6789     4    |  589     579     5789  | 23569   23589   23689
```

In this case, we are still a long way from solving the puzzle--61 squares remain uncertain. What next? We could try to code more sophisticated strategies. For example, the naked twins strategy looks for two squares in the same unit that both have the same two possible digits. Given `{'A5': '26', 'A6':'26', ...}`, we can conclude that 2 and 6 must be in `A5` and `A6` (although we don't know which is where), and we can therefore eliminate 2 and 6 from every other square in the `A` row unit. We could code that strategy in a few lines by adding an `elif len(values[s]) == 2` test to `eliminate`.

Coding up strategies like this is a possible route, but would require hundreds of lines of code (there are dozens of these strategies), and we'd never be sure if we could solve every puzzle.

Search

The other route is to search for a solution: to systematically try all possibilities until we hit one that works. The code for this is less than a dozen lines, but we run another risk: that it might take forever to run. Consider that in the `grid2` above, `A2` has 4 possibilities (`1679`) and `A3` has 5 possibilities (`12679`); together that's 20, and if we keep multiplying, we get $4.62838344192 \times 10^{38}$ possibilities for the whole puzzle. How can we cope with that? There are two choices.

First, we could try a brute force approach. Suppose we have a very efficient program that takes only one instruction to evaluate a position, and that we have access to the next-generation computing technology, let's say a 10GHz processor with 1024 cores, and let's say we could afford a million of them, and while we're shopping, let's say we also pick up a time machine and go back 13 billion years to the origin of the universe and start our program running. We can then compute that we'd be almost 1% done with this one puzzle by now.

The second choice is to somehow process much more than one possibility per machine instruction. That seems impossible, but fortunately it is exactly what constraint propagation does for us. We don't have to try all 4×10^{38} possibilities because as soon as we try one we immediately eliminate many other possibilities. For example, square H7 of this puzzle has two possibilities, 6 and 9. We can try 9 and quickly see that there is a contradiction. That means we've eliminated not just one possibility, but fully half of the 4×10^{38} choices.

In fact, it turns out that to solve this particular puzzle we need to look at only 25 possibilities and we only have to explicitly search through 9 of the 61 unfilled squares; constraint propagation does the rest. For the list of 95 hard puzzles, on average we need to consider 64 possibilities per puzzle, and in no case do we have to search more than 16 squares.

What is the search algorithm? Simple: first make sure we haven't already found a solution or a contradiction, and if not, choose one unfilled square and consider all its possible values. One at a time, try assigning the square each value, and searching from the resulting position. In other words, we search for a value `d` such that we can successfully search for a solution from the result of assigning square `s` to `d`. If the search leads to an failed position, go back and consider another value of `d`. This is a recursive search, and we call it a depth-first search because we (recursively) consider all possibilities under `values[s] = d` before we consider a different value for `s`.

To avoid bookkeeping complications, we create a new copy of `values` for each recursive call to `search`. This way each branch of the search tree is independent, and doesn't confuse another branch. (This is why I chose to implement the set of possible values for a square as a string: I can copy `values` with `values.copy()` which is simple and efficient. If I implemented a possibility as a Python `set` or `list` I would need to use `copy.deepcopy(values)`, which is less efficient.) The alternative is to keep track of each change to `values` and undo the change when we hit a dead end. This is known as backtracking search. It makes sense when each step in the search is a single change to a large data structure, but is complicated when each assignment can lead to many other changes via constraint propagation.

There are two choices we have to make in implementing the search: variable ordering (which square do we try first?) and value ordering (which digit do we try first for the square?). For variable ordering, we will use a common heuristic called minimum remaining values, which means that we choose the (or one of the) square with the minimum number of possible values. Why? Consider `grid2` above. Suppose we chose `B3` first. It has 7 possibilities (`1256789`), so we'd expect to guess wrong with probability 6/7. If instead we chose `G2`, which only has 2 possibilities (`89`), we'd expect to be wrong with probability only 1/2. Thus we choose the square with the fewest possibilities and the best chance of guessing right. For value ordering we won't do anything special; we'll consider the digits in numeric order.

Now we're ready to define the `solve` function in terms of the `search` function:

```
def solve(grid): return search(parse_grid(grid))

def search(values):
    "Using depth-first search and propagation, try all possible values."
    if values is False:
        return False ## Failed earlier
    if all(len(values[s]) == 1 for s in squares):
        return values ## Solved!
    ## Chose the unfilled square s with the fewest possibilities
    n,s = min((len(values[s]), s) for s in squares if len(values[s]) > 1)
    return some(search(assign(values.copy(), s, d))
                for d in values[s])

def some(seq):
    "Return some element of seq that is true."
    for e in seq:
        if e: return e
    return False
```

That's it! We're done; it only took one page of code, and we can now solve any Sudoku puzzle.

Results

You can view the complete program. Below is the output from running the program at the command line; it solves the two files of 50 easy and 95 hard puzzles (see also the 95 solutions), eleven puzzles I found under a search for [hardest sudoku], and a selection of random puzzles:

```
% python sudo.py
All tests pass.
Solved 50 of 50 easy puzzles (avg 0.01 secs (86 Hz), max 0.03 secs).
Solved 95 of 95 hard puzzles (avg 0.04 secs (24 Hz), max 0.18 secs).
Solved 11 of 11 hardest puzzles (avg 0.01 secs (71 Hz), max 0.02 secs).
Solved 99 of 99 random puzzles (avg 0.01 secs (85 Hz), max 0.02 secs).
```

Analysis

Each of the puzzles above was solved in less than a fifth of a second. What about really hard puzzles? Finnish mathematician Arto Inkala described his 2006 puzzle as "the most difficult sudoku-puzzle known so far" and his 2010 puzzle as "the most difficult puzzle I've ever created." My program solves them in 0.01 seconds each (`solve_all` will be defined below):

```
>>> solve_all(from_file("hardest.txt")[0:2], 'Inkala')
8 5 . |. . 2 |4 . .
7 2 . |. . . |. . 9
. . 4 |. . . |. . .
------+------+------
. . . |1 . 7 |. . 2
3 . 5 |. . . |9 . .
. 4 . |. . . |. . .
------+------+------
. . . |. 8 . |. 7 .
. 1 7 |. . . |. . .
. . . |. 3 6 |. 4 .

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

(0.01 seconds)

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

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

(0.01 seconds)

Solved 2 of 2 Inkala puzzles (avg 0.01 secs (99 Hz), max 0.01 secs).
```

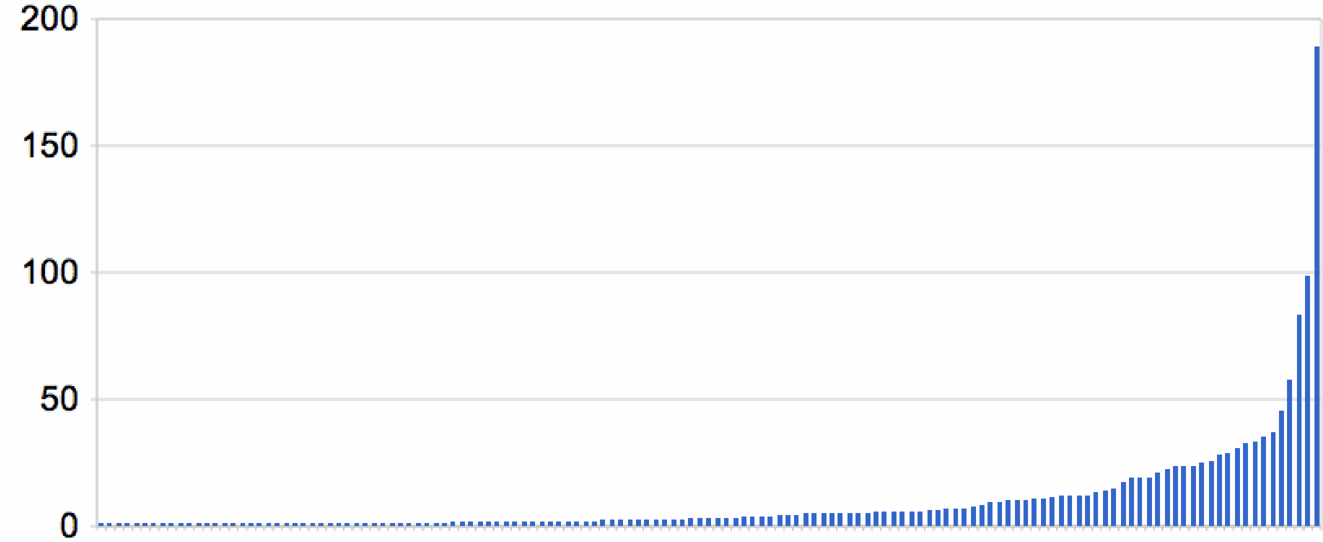
I guess if I want a really hard puzzle I'll have to make it myself. I don't know how to make hard puzzles, so I generated a million random puzzles. My algorithm for making a random puzzle is simple: first, randomly shuffle the order of the squares. One by one, fill in each square with a random digit, respecting the possible digit choices. If a contradiction is reached, start over. If we fill at least 17 squares with at least 8 different digits then we are done. (Note: with less than 17 squares filled in or less than 8 different digits it is known that there will be duplicate solutions. Thanks to Olivier Grégoire for the fine suggestion about 8 different digits.) Even with these checks, my random puzzles are not guaranteed to have one unique solution. Many have multiple solutions, and a few (about 0.2%) have no solution. Puzzles that appear in books and newspapers always have one unique solution.

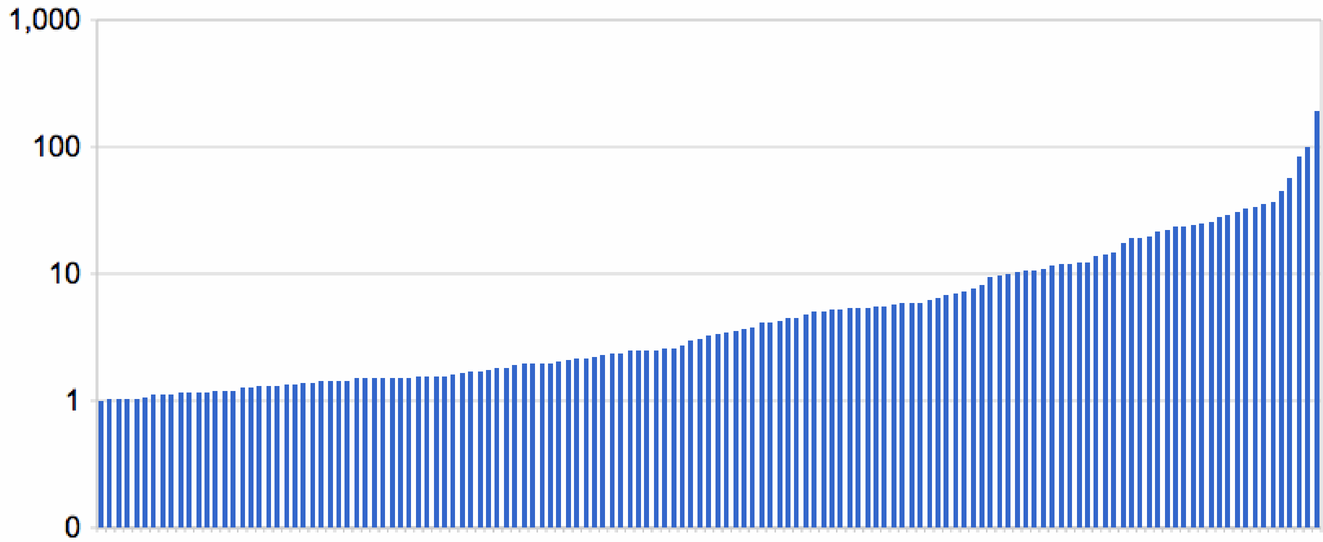
The average time to solve a random puzzle is 0.01 seconds, and more than 99.95% took less than 0.1 seconds, but a few took much longer:

0.032%	(1 in 3,000) took more than 0.1 seconds
0.014%	(1 in 7,000) took more than 1 second
0.003%	(1 in 30,000) took more than 10 seconds

0.0001% (1 in 1,000,000) took more than 100 seconds

Here are the times in seconds for the 139 out of a million puzzles that took more than a second, sorted, on linear and log scales:





It is hard to draw conclusions from this. Is the uptick in the last few values significant? If I generated 10 million puzzles, would one take 1000 seconds? Here's the hardest (for my program) of the million random puzzles:

```
>>> hard1  = '.....6....59.....82....8....45........3........6..3.54...325..6..................'
>>> solve_all([hard1])
. . . |. . 6 |. . . 
. 5 9 |. . . |. . 8 
2 . . |. . 8 |. . . 
------+------+------
. 4 5 |. . . |. . . 
. . 3 |. . . |. . . 
. . 6 |. . 3 |. 5 4 
------+------+------
. . . |3 2 5 |. . 6 
. . . |. . . |. . . 
. . . |. . . |. . . 

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

```
(188.79 seconds)
```

Unfortunately, this is not a true Sudoku puzzle because it has multiple solutions. (It was generated before I incorporated Olivier Grégoire's suggestion about checking for 8 digits, so note that any solution to this puzzle leads to another solution where the 1s and 7s are swapped.) But is this an intrinsically hard puzzle? Or is the difficulty an artifact of the particular variable- and value-ordering scheme used by my `search` routine? To test I randomized the value ordering (I changed `for d in values[s]` in the last line of `search` to be `for d in shuffled(values[s])` and implemented `shuffled` using `random.shuffle`). The results were starkly bimodal: in 27 out of 30 trials the puzzle took less than 0.02 seconds, while each of the other 3 trials took just about 190 seconds (about 10,000 times longer). There are multiple solutions to this puzzle, and the randomized `search` found 13 different solutions. My guess is that somewhere early in the search there is a sequence of squares (probably two) such that if we choose the exact wrong combination of values to fill the squares, it takes about 190 seconds to discover that there is a contradiction. But if we make any other choice, we very quickly either find a solution or find a contradiction and move on to another choice. So the speed of the algorithm is determined by whether it can avoid the deadly combination of value choices.

Randomization works most of the time (27 out of 30), but perhaps we could do even better by considering a better value ordering (one popular heuristic is least-constraining value, which chooses first the value that imposes the fewest constraints on peers), or by trying a smarter variable ordering.

More experimentation would be needed before I could give a good characterization of the hard puzzles. I decided to experiment on another million random puzzles, this time keeping statistics on the mean, 50th (median), 90th and 99th percentiles, maximum and standard deviation of run times. The results were similar, except this time I got two puzzles that took over 100 seconds, and one took quite a bit longer: 1439 seconds. It turns out this puzzle is one of the 0.2% that has no solution, so maybe it doesn't count. But the main message is that the mean and median stay about the same even as we sample more, but the maximum keeps going up--dramatically. The standard deviation edges up too, but mostly because of the very few very long times that are way out beyond the 99th percentile. This is a heavy-tailed distribution, not a normal one.

For comparison, the tables below give the statistics for puzzle-solving run times on the left, and for samples from a normal (Gaussian) distribution with mean 0.014 and standard deviation 1.4794 on the right. Note that with a million samples, the max of the Gaussian is 5 standard deviations above the mean (roughly what you'd expect from a Gaussian), while the maximum puzzle run time is 1000 standard deviations above the mean.

Samples of Puzzle Run Time

N	mean	50%	90%	99%	max	std. dev
10	0.012	0.01	0.01	0.01	0.02	0.0034
100	0.011	0.01	0.01	0.02	0.02	0.0029
1,000	0.011	0.01	0.01	0.02	0.02	0.0025
10,000	0.011	0.01	0.01	0.02	0.68	0.0093
100,000	0.012	0.01	0.01	0.02	29.07	0.1336
1,000,000	0.014	0.01	0.01	0.02	1439.81	1.4794

Samples of N(0.014, 1.4794)

N	mean	50%	90%	99%	max	std. dev
10	0.312	1.24	1.62	1.62	1.62	1.4061
100	0.278	0.18	2.33	4.15	4.15	1.4985
1,000	0.072	0.10	1.94	3.38	6.18	1.4973
10,000	0.025	0.05	1.94	3.45	6.18	1.4983
100,000	0.017	0.02	1.91	3.47	7.07	1.4820
1,000,000	0.014	0.01	1.91	3.46	7.80	1.4802

Here is the impossible puzzle that took 1439 seconds:

```
. . . |. . 5 |. 8 .
. . . |6 . 1 |. 4 3
. . . |. . . |. . .
------+------+------
. 1 . |5 . . |. . .
. . . |1 . 6 |. . .
3 . . |. . . |. . 5
------+------+------
5 3 . |. . . |. 6 1
. . . |. . . |. . 4
. . . |. . . |. . .
```

Here is the code that defines `solve_all` and uses it to verify puzzles from a file as well as random puzzles:

```
import time, random

def solve_all(grids, name='', showif=0.0):
    """Attempt to solve a sequence of grids. Report results.
    When showif is a number of seconds, display puzzles that take longer.
    When showif is None, don't display any puzzles."""
    def time_solve(grid):
        start = time.clock()
        values = solve(grid)
        t = time.clock()-start
        ## Display puzzles that take long enough
        if showif is not None and t > showif:
            display(grid_values(grid))
            if values: display(values)
            print '(%.2f seconds)\n' % t
        return (t, solved(values))
    times, results = zip(*[time_solve(grid) for grid in grids])
    N = len(grids)
    if N > 1:
        print "Solved %d of %d %s puzzles (avg %.2f secs (%d Hz), max %.2f secs)." % (
            sum(results), N, name, sum(times)/N, N/sum(times), max(times))

def solved(values):
    "A puzzle is solved if each unit is a permutation of the digits 1 to 9."
    def unitsolved(unit): return set(values[s] for s in unit) == set(digits)
    return values is not False and all(unitsolved(unit) for unit in unitlist)

def from_file(filename, sep='\n'):
    "Parse a file into a list of strings, separated by sep."
    return file(filename).read().strip().split(sep)

def random_puzzle(N=17):
    """Make a random puzzle with N or more assignments. Restart on contradictions.
    Note the resulting puzzle is not guaranteed to be solvable, but empirically
    about 99.8% of them are solvable. Some have multiple solutions."""
    values = dict((s, digits) for s in squares)
    for s in shuffled(squares):
        if not assign(values, s, random.choice(values[s])):
            break
        ds = [values[s] for s in squares if len(values[s]) == 1]
        if len(ds) >= N and len(set(ds)) >= 8:
            return ''.join(values[s] if len(values[s])==1 else '.' for s in squares)
    return random_puzzle(N) ## Give up and make a new puzzle

def shuffled(seq):
    "Return a randomly shuffled copy of the input sequence."
    seq = list(seq)
    random.shuffle(seq)
    return seq

grid1  = '003020600900305001001806400008102900700000008006708200002609500800203009005010300'
grid2  = '4.....8.5.3..........7......2.....6.....8.4......1.......6.3.7.5..2.....1.4......'
hard1  = '.....6....59.....82....8....45........3........6..3.54...325..6..................'

if __name__ == '__main__':
    test()
    solve_all(from_file("easy50.txt", '========'), "easy", None)
    solve_all(from_file("top95.txt"), "hard", None)
    solve_all(from_file("hardest.txt"), "hardest", None)
    solve_all([random_puzzle() for _ in range(99)], "random", 100.0)
```

Why?

Why did I do this? As computer security expert Ben Laurie has stated, Sudoku is "a denial of service attack on human intellect". Several people I know (including my wife) were infected by the virus, and I thought maybe this would demonstrate that they didn't need to spend any more time on Sudoku. It didn't work for my friends (although my wife has since independently kicked the habit without my help), but at least one stranger wrote and said this page worked for him, so I've made the world more productive. And perhaps along the way I've taught something about Python, constraint propagation, and search.

Translations

This code has been reimplemented by several people in several languages:

- C++ by Pau Fernandez
- C# with LINQ version by Richard Birkby
- Clojure version by Justin Kramer

- Erlang version by Andreas Pauley
- Haskell version by Emmanuel Delaborde
- Java version by Johannes Brodwall
- Javascript version by Brendan Eich
- Javascript version by Pankaj Kumar
- Ruby version by Luddite Geek
- Ruby version by Martin-Louis Bright

You can see a Korean translation of this article by JongMan Koo, or use the translation widget below:

Peter Norvig

Alan • 7 years ago

1) The code as it stands will overwrite squares in the initial puzzle; these squares need to be frozen.
This problem was exhibited by parse_grid, which sometimes changed the starting puzzle.

The solution is to create a list squares0 of frozen squares when initializing the puzzle.

I prefer to replace parse_grid by

def init_grid_as_dict(grid): ## Assumption: grid is a string, not a list
"""Convert grid to a dict of possible values, {square: digits}, or return False if a contradiction is detected."""

To start, every square can be any digit; then assign values from the grid.

grid = grid_to_values(grid)
values = dict((s, digits) for s in squares)
squares0 = []
for s in squares:
if grid[s] in digits: squares0.append(s); values[s] = grid[s]
Sq = [s for s in squares if not s in squares0]
for s in Sq :
for u in peers[s]:
if u in squares0:
values[s] = values[s].replace(values[u][0], "")
return values
Modify assign and eliminate as follows:
if s in squares0:
assert len(values[s]) == 1
if values[s] == d : return values
return False ## else

2) Of course, the smallest unsolvable grid has size N = 7. Just put

2) Of course, the smallest unsolvable grid has size N = 7. Just put
1, 2, 3, 4, 5, 6 in the first column and 7, 8, or 9 in the last block in columns 2 or 3. A better question to ask is: what is the smallest possible uniquely solvable grid?

3) The problem of uniqueness is important if you want to generate puzzles for public consumption.

Here is some code that makes a start on the problem.

=====================================

########################### UNIQUENESS
######################

def hasUniqueSolution(grid):
solns = findMultipleSolutions(grid)
return len(solns) == 1

def findMultipleSolutions(grid, bQuick = True): ## Returns a list solns, solns[i] = <solution grid,="" square="" s,="" j="values[s]">
len(solns) > 100 is common and the list of solutions we return is not exhaustive.

It's not hard to see that if this code returns solns with len(solns) = 1, then grid has exactly one solution
and it's the one that we found. Here is the idea. Suppose grid has two solutions, S1 and S2. Then they must
differ in some square s, which holds v1 and v2, respectively. When the code tests this square the legal values for
the dot at s necessarily include v1 and v2, and the code necessarily finds a solution T1 and T2 for each (and adds T1 and
T2 to solns if they are not already there). However, there is no guarantee that T1 = S1 and T2 = S2, since the solution
from s for the two values may not be unique; the details depend on the choices used by search. To deal with
this we would have to radiate outward recursively, calling findMultipleSolutions again for each grid in solns, which is
not practical. If N = grid_len(grid) is the grid size, i.e., the number of non-dot entries in grid, then the code looks at
all possible legal grids of size N + 1 that contain grid.

If bQuick is True, we return as soon as nonuniqueness is detected. If bQuick is False, this code will return a sample of the
available solutions.

The main point of this code is that if solns with len(solns) = 1 is returned, grid is guaranteed to be uniquely solvable.

values0 = solve_all([grid], "", False)
if not values0:
return []
values = init_grid_as_dict(grid)
grid0 = grid
solns = []
nums = shuffled(range(81)) ## randomize so as to avoid building up exclusively at the beginning. We seek uniquely solvable
puzzles with small N and well-distributed filled squares
for i in nums:
if grid0[i] != '.':
continue
s = grid_index_to_square(i)
vals = values[s]
if bQuick:
solns = [] ## Check multiplicity for each given square s only

```
for j in vals:
grid = grid0 ## breadth first
grid = grid[:i] + j + grid[i+1:]
L = solve_all([grid], "", False) ## L = list of tuples <time,
solngrid="">
if not L: ## no solution
continue
G = L[0][1] ## We called solve_all with [grid], so there is at most
one item in the returned list
## namely L[0], and we take the solution grid part L[0][1]
t = G, s, j
if all(t[0] != t2[0] for t2 in solns):
solns.append(t) ## t is not in our list of solutions found so far, so
add it
else:
pass ## debug

if bQuick and len(solns) > 1: ## We just want to know if there is
more than one solution, so return now
return solns
return solns ## solns[i] is a tuple <solngrid, s,="" val="j">

def enlarge_until_unique(grid): ## If grid has multiple solutions,
keep adding assigned squares to it until there is only one solution
## Note: There is no guarantee that enlarging in this way will
produce a uniquely solvable grid -- we could go from multiple
solutions
## in the previous step to no solutions at all in the current step. If
this happens, enlarge_until_unique returns False. I don't yet
## have an example of this. The digit entries of grid are frozen
and cannot be changed.
display_grid(grid)
solns = findMultipleSolutions(grid, True) ## True: bQuick -- stop as
soon as nonuniqueness is detected
if len(solns) == 1:
return grid
elif len(solns) == 0: ## The null case mentioned above
return False
f.write("\n\nTwo solutions:\n\n")
display_grid(solns[0][0])
display_grid(solns[1][0])
f.write("\n\n----------------------------\n\n")
j = shuffled(range(len(solns)))[0]
s = solns[j][1]; d = solns[j][2]; i = index(s)
grid = grid[:i] + str(d) + grid[i+1:]
return enlarge_until_unique(grid) ## returns a uniquely solvable
grid string containing grid, or else False if there is none

def is_minimal_unique(uniq, grid_core): ## arguments are strings.
uniq contains the starting puzzle grid_core whose digit entries
# are frozen and cannot be changed. Recall: The size N of a grid
or puzzle is the number of non-dot entries, N = grid_len(grid). A
fully
# solved puzzle has size 81.
# uniq typically comes from a call to
enlarge_until_unique(grid_core). We may be able to cut uniq back
if some of its square values turn out to
# be consequences of its other values. This may seem surprising.
The explanation is that enlarge_until_unique(grid) freezes all digit
values of
# grid (i.e., the entire expanding grid is treated as a puzzle). By
contrast, the present code freezes only the digit squares of
```

grid_core.
grids = []
Ind = []
for i in range(81): ## Find the places that are not frozen
if grid_core[i] not in digits: ## cannot change any digit in grid_core, but will assign to dots
Ind.append(int(i))
for i in Ind:
if uniq[i] in digits: ## relax the constraint on uniq by making uniq[i] a dot, store as grid
grid = uniq[:i] + '.' + uniq[i+1:]
grids.append(grid)
for grid in grids: ## the singly unfrozen grids
if hasUniqueSolution(grid):
return False, grid ## Like uniq, grid has a unique solution but it is smaller, choose it
return True, uniq

def to_minimal_unique(grid, grid0):
if not hasUniqueSolution(grid):
return False
imu = is_minimal_unique(grid, grid0)
if imu[0]:
return grid
return to_minimal_unique(imu[1], grid0)

##

##To exercise this code, try:

puzzles = [random_puzzle() for _ in range(1)]
L = solve_all(puzzles, "Test", True, True)
display_grid(puzzles[0])
uniq = enlarge_until_unique(puzzles[0]) ## uniq is a grid string
L = solve_all([uniq], "uniq", True, True)
uniq2 = to_minimal_unique(uniq, puzzles[0])
L = solve_all([uniq2], "Minimal unique", True, True)
f.close()

##-----------------------

##Here f is a file opened for debugging.

32 ^ | ⌄ • Reply • Share ›

naeg • 10 years ago

This article should be called "how to win sudoku".

29 ^ | ⌄ • Reply • Share ›

Michel Lespinasse • 10 years ago

I like your writeup & your concise implementation.

When doing my own sudoku solver, I used a similar approach but with a twist in the constraint propagation. I had constraints not just on cell contents like in yours, but also in values that MAY or MUST be seen within blocks of 3 consecutive cells (horizontally or vertically). I found that this allowed my algorithm to make quick progress in cases where the search method would not - for example, I can find out that your 1439 seconds puzzle has no solution without having to invoke search even once.

As an example of how this works:
If the possible values for 3 consecutive cells don't include a given value, one can note that the corresponding cell block may not

include that value.
if two aligned cell blocks may not include a given value, then the third one must include that value (even if we don't know which of the 3 cells will have the value in).
Since we know the value will be present in that third cell block, it may not be present in any of the other 6 cells in the corresponding 3x3 block.
This is a good heuristic to have because it's very complementary with the search - it works well in cases where the search does not.

18 ^ | ⌄ • Reply • Share ›

Ravi Annaswamy • 10 years ago

Genius is fluency. Genius is going to a distance where no one has gone and then going further and further away.

Peter Norvig teases me. As though if you give him an hour, he can take any issue and make a giant step. Wow! I memorized the first version of your sudoku.py to learn Python idioms. I spent a couple of months with that code and it led me to functional programming, lisp and so many beautiful things. Proof that if you go deep, you encompass wide. Thank you.
Ravi

16 ^ | ⌄ • Reply • Share ›

swapnil ➦ Ravi Annaswamy • 3 years ago

wow

^ | ⌄ • Reply • Share ›

Anonymous • 9 years ago

Ahh, but there is an advantage to playing sudoku (as opposed to just completing the puzzle with a computer program). That advantage is making your brain think in a logical fashion, not just work from rote. And it is this logical thought process that gets the blood flowing to sometimes under-utilized portions of the brain, and can help stave off both boredom and possibly even dementia.

7 ^ | ⌄ • Reply • Share ›

Harald Schilly ➦ Anonymous • 9 years ago

this is right, i think, but i also read that it only
helps against dementia as long as you are "learning" the rules. once you managed to learn sudoku to the point where you are into it and can solve it ... means you know some patterns and it feels nice to live in this world of rules ... you are already beyond the point where it helps. at this point, you need to search for a new puzzle and don't get sucked into this one too much :-)

6 ^ | ⌄ • Reply • Share ›

Dave_Bernazani • 7 years ago

Mr. Norvig,
I am neither a mathematician nor a programmer, but I am an avid sudoku player, and your article was so well written it was understandable and interesting even to me! Thanks for sharing this fascinating info.
David Bernazani, Lafayette, CA

6 ^ | ⌄ • Reply • Share ›

Dong Peter ➦ Dave_Bernazani • a year ago

Me too. Thank Mr NorVig,

^ | ⌄ • Reply • Share ›

Saurabh Mangal ➦ Dave_Bernazani • 4 years ago

What I fail to understand (but want to know) is that, what is the motivation behind solving this puzzle when you can clearly see that its a backtracking type of problem, suitable for computers and not humans... in other words, is it possible that a person's intuition can make him a better agent of solving this then the 'mechanical' algorithm a computer uses (considering only variable ordering and value ordering choices and the like). After all everybody would eventually try to formulate an algorithm in their heads, right?

^ | ˅ • Reply • Share ›

Dave_Bernazani ➦ Saurabh Mangal • 4 years ago

Saurabh, I don't know, it's just kinda fun and satisfying to finish one. It's like doing crossword puzzles or playing solitaire: it's a minor brain challenge that is both relaxing and stimulating at the same time. You don't need a "motivation", just try them and see for yourself.

^ | ˅ • Reply • Share ›

Mike S ➦ Dave_Bernazani • 3 years ago • edited

Agreed, I like my book, "The Original Sudoku" which includes 300 hand-designed puzzles. They progress from easy to medium to hard to very hard. It's really fun finding a new technique with new puzzles where all of the previous techniques no longer work. After you have done several dozen and gotten comfortable with them (at least that's what it took me) you can see the author introducing added complexity requiring you to learn something new, it's fun and frustrating and tantalizing all at once.

1 ^ | ˅ • Reply • Share ›

Abel Mengistu • 7 years ago • edited

A simple suggestion:

This last weekend I solved this puzzle on Project Euler, after which I read a comment in the forum that led me here.
I had used the same constraint propagation strategies (1) and (2), and did the search in a similar way starting with squares that have the fewest options. However, with the same reasoning for choosing the squares with fewest possibilities, I was ordering the options themselves based on their frequency in the whole grid. I had bookkeeping code, that kept the frequencies. I tested it on the **hard1** puzzle from above, and it ran in less than 0.06 seconds. :)

5 ^ | ˅ • Reply • Share ›

Idk ➦ Abel Mengistu • 6 years ago • edited

Could you share your code for this approach?
I'm currently learning python an am thinking about how one could speed this thing up.Thanks!

1 ^ | ˅ • Reply • Share ›

Thomas Lambrecht • 9 years ago • edited

working on the C implementation right now (i have got to make this for an exam)
It's really helpfull to have some guidlines to follow, thank you for the guide.

Step (1) and (2) work already... starting on the search part

5 ^ | v • Reply • Share ›

Bill Zwicky • 10 years ago

Your link for "easy 50" has the wrong file name and wrong format, and the program no longer works with them. To fix, use this from_file which supports regular expressions:

```
def from_file(filename, sep='\n'):
"Parse a file into a list of strings, separated by sep."
s = file(filename).read().strip()
l = re.split(sep, s)
l = filter(lambda z: len(z)>0, l)
return l
```

and this test line:

```
solve_all(from_file("easy50.txt", "Grid [0-9]+"), "easy", None)
```

5 ^ | v • Reply • Share ›

Peter Norvig Mod → Bill Zwicky • 10 years ago

Bill, you are exactly right: I tried to give more credit to project Euler by linking directly to their file, when I should have linked to my local version of the file, in which "Grid ..." is replaced by a constant '======' marker. Your solution using re.split is a good one and more general than my solution, but I wanted to keep it simple.

6 ^ | v • Reply • Share ›

Ed • 7 years ago

I am a novice python programmer that happened to stumble on this page by trying to solve the Euler sudoku puzzle. I have to say I have learned so much from the beauty and simplicity of your code. My code for equivalent tasks is so bloated!!! Going through your code and trying to rewrite parts of my own algorithm has really been a great lesson in python for me. Thanks so much.

I am wondering... will I ever be able to write such concise code? Its a little discouraging to think about it, actually.

4 ^ | v • Reply • Share ›

Mike S → Ed • 3 years ago

How long have you been programming? When you look at your first efforts, do you see that you've made any improvements?

^ | v • Reply • Share ›

Alan • 8 years ago

Peter,

Can you please post the grid for one of your "unsolvable" puzzles? How do you know it's truly unsolvable? By repeatedly running the program with randomized choices in search:

```
for d in shuffled(values[s])),
```

and then waiting for ten minutes, say, to see if a solution is found? If none is found after a dozen runs (i.e., 12 independent random

If none is found after a dozen runs (i.e., 12 independent random brute-force strategies) that would be pretty convincing that it's not solvable (but still not proof). I believe that 17 initial legally filled squares will always be solvable; more squares can be added in real puzzles to guarantee a unique solution (the puzzle maker would choose the "hardest" of the ones available for a typical "human" strategy).

There is clearly a largest number N such that all legal initial Sudoku configurations with N filled squares are solvable. If I'm wrong and N != 17, then N must be < 17. Does anyone know what N is? Attempts at a theoretical attack on this question seem to run into insuperable difficulties.

Presumably, repeated runs on the same puzzle with random shuffling will produce all solutions if multiple solutions exist, with some very high probability. Does anyone know what this probability is (if not theoretically, then at least empirically)? What about generalizations (e.g., hexadecimal Sudoku with a 15 x 15 grid). If there is an applicable theory somewhere, it should work equally well with all generalizations of this type. The most interesting outcome for me would be that there is no such theory. Then Sudoku could be telling us something profound about our universe (change Einstein's "God does not play dice." to "God does not play Sudoku.", both assertions, of course, being wrong)

2 ^ | v • Reply • Share ›

Martin d'Anjou • 9 years ago • edited

Something is strange with this one, it cannot be solved, yet it has 17 values in it:

```
>>>
qs='...8.1..........435............7.8........1...2..3....6......75..34........2..6..'
>>> solve_all([qs])
>>> display(parse_grid(qs))
23479 3679 4679 | 8 249 1 | 5 269 269
1289 1689 1689 | 7 29 5 | 29 4 3
5 189 1489 | 3 249 6 | 7 1289 1289
------------------+------------------+------------------
1349 1369 14569 | 156 7 24 | 8 23569 269
3479 3679 45679 | 56 8 24 | 1 23569 2679
178 2 15678 | 156 3 9 | 4 56 67
------------------+------------------+------------------
6 4 2 | 9 1 8 | 3 7 5
189 5 3 | 4 6 7 | 29 1289 1289
1789 1789 1789 | 2 5 3 | 6 189 4
```

We need to give it a hand:

```
>>>
qs='...8.1..........435............7.8........1...2..3....6......75..34........2..68.'
>>> solve_all([qs])
>>> display(parse_grid(qs))
2 3 7 |8 4 1 |5 6 9
1 8 6 |7 9 5 |2 4 3
5 9 4 |3 2 6 |7 1 8
------+------+------
3 1 5 |6 7 4 |8 9 2
4 6 9 |5 8 2 |1 3 7
7 2 8 |1 3 9 |4 5 6
------+------+------
6 4 2 |9 1 8 |3 7 5
8 5 3 |4 6 7 |9 2 1
9 7 1 |2 5 3 |6 8 4
```

2 ^ | v • Reply • Share ›

Mark Lawrence ➦ Martin d'Anjou • 9 years ago

You are mistaken. I've tried this with both Peter's solution and with the excellent Sudoku Explainer and they tally exactly, there is no need to give this solver a helping hand.

2 ^ | v 1 • Reply • Share ›

Martin d'Anjou ➦ Mark Lawrence • 9 years ago

Please show me the commands. I'd like to know what I am missing.

2 ^ | v • Reply • Share ›

Peter Gfader • 2 years ago

Thanks Peter for this writeup.
I especially enjoyed your thinking, your wise choose of language and your additional things you did and mentioned like (runtime on a processor over time and the "Analysis" section)
and not just "programming sudoku".

Thanks!

1 ^ | v • Reply • Share ›

jssk • 4 years ago • edited

Hi Peter,

I'm an amateur programmer reading around the web looking for ideas to improve on my sudoku solving algorithm or grids that would help expose its bugs, when I came across your page. It is very interesting and your program was so nicely written. Mine was about 1000 lines!

I understand the last grid was impossible, but it probably shouldn't have taken 1400 seconds to figure out it was actually an unsolvable grid. In fact, right off the bat, it becomes apparent that, an instance of number 1 , 5 , and 6 must be placed in either H5 or I5 (under your indexing method) locations, i.e. we have to fit 3 numbers in 2 spots, which immediately makes it an invalid grid.

It took my program 4 iterations to reach that conclusion, which took about 5 seconds. It was probably because of my old slow computer or something was wrong with my implementation in Python, because in my original Matlab code anything under 30 iterations took a fraction of a second.

Anyway, thanks for sharing your thoughts and code. Cheers!

Ning

1 ^ | v • Reply • Share ›

Dr. Mike • 5 years ago

Peter, is this discussion still open? If so I would like to make a comment or two. I'm an old Fortran hacker and I thought it might be amusing to program Sudoku. Years ago, I programmed chess so that I could get a good game
and the program got good enough to beat some very good players. It used alpha-beta pruning to cut off the search
trees, and worked pretty well. I had just been through Knuth's discussion of how to make and test random number generators, and another paper of his discussing searching by using "dancing links". The sudoku solving program
was to be my "homework" about learning this technique. I got the

program to work, but it seemed to me that
it was spending too much time looking at incorrect possibilities
whose error was found fairly deep in the search tree.
So I made an idiot's program by picking entries at random, and
just starting over when I found an inconsistency, using no search
or backtracking.

When I added constraint propagation by recursively eliminating
naked and hidden singles, I was astonished to find
that the time to solve any puzzle was a few milliseconds or less. A
little study showed that the constraint clean-up was
eliminating over 90% of the moves, and I only had to find three to
five correct ones in a row to solve the puzzle.
Since there were usually about 5 possibilities in a cell, at worst
one only had to try about 100-150 random
sequences to solve the puzzle. Since each sequence took a few
microseconds, the game was solved
quickly.

I got a data base of over 1700 sudoku games ranging from easy
to the hardest ones ever found, and the
average solution time was 4 milliseconds. The easy ones were
solved by the constrain propagation alone
and the longest time was a problem with the name of Platinum
Blonde which took 45 milliseconds.

Since each solution used a different sequence, the entire data
base was solved over 1000 times and
averaged. The standard deviation and the solution times were
about the same for each of the puzzles
indicating an exponential distribution.

Since the entries were being guessed, the chance of being right
using no heuristics was greatest
in the squares with the fewest possibilities, so I picked the trial
square with a probability of the
inverse of the number of possibilities that it contained. Any square
could be picked at any point, but
the ones with fewest possibilities were most likely. This simple
addition speeded up the game by a factor
of two to three. Anyway its been fun. And reading your writeup, I
was amused to find that we were
using pretty much the same data structure. Thanks for an
interesting read.

1 ^ | ˅ • Reply • Share ›

Jimmy Gauvin • 5 years ago

This is great stuff for learning how to program searching and
backtracking.
What I would like to find is a technique for solving a sudoku puzzle
without backtracking.

1 ^ | ˅ • Reply • Share ›

beka • 6 years ago

nice article. Peter, do you have more puzzles like the impossible
one? I generated about 4 million random puzzles (without "at least
different 8" rule), but they are all solved in less than 1s (including
yours).

1 ^ | ˅ • Reply • Share ›

Rushikesh • 7 years ago

Excellent way to solve the Sudoku

Excellent way to solve the Sudoku
without having to discover the various rules with which the number
can
be deterministically determined!!!