Sudoku and Recursive Algorithms
Chris Fallin

Sudoku

The reader may be familiar with the popular Sudoku number puzzle. In truth, Sudoku is not a number puzzle so much as a symbol placement puzzle; any symbols may be used. However, the common instance of the puzzle uses Arabic numerals 1-9.

The goal of a Sudoku puzzle, constructed as a 9x9 grid, one digit per cell, with various digits given, is to fill in the entire grid while satisfying a simple constraint. The single constraint is that no digit may occur more than once per 3x3 subgrid (of which there are 9), per column, or per row. A sample Sudoku puzzle is given in Figure 1.

Upon examination, one quickly develops a few common techniques. Firstly, it is possible to eliminate possibilities for many cells. The top left cell, for example, cannot contain any of 2, 8, 7, 5, 6, 1, 4 – all digits in the cell's row, its column, and its 3x3 area. This leaves only 3 and 9 as possibilities. Continuing through the rest of the problem, one may construct similar lists.

However, this technique suffices only for the simplest of puzzle setups. As seen in the example above, multiple possibilities will exist, and the solver must make a decision on one possibility, exploring the roads that this choice opens and keeping in mind the possibility of the choice's impossibility – thus requiring the solver to back out and try another possibility.

In summary, we have developed the following very simple and generic technique: from

```
6 1 4 5
8 3 5 6
2
8 4 7 6
6
7 9 1 4
5
7 2 6 9
4 5 8 7
```
the known knowledge, derive all that is possible, perhaps gaining more knowledge as a
result; then, when the possibilities have been narrowed as far as possible, choose one
possibility and try to solve the resulting grid; if this fails, choose another possibility and
try again. We will use this procedure to develop a formal algorithm.

Algorithms

What we have described thus far is a very informal approach, with very general
wording: “narrow the possibilities,” “try to solve the resulting grid,” etc. When a
human solver is attempting a problem, this is usually enough – humans possess
intelligence and the capacity to learn, and thus are able to derive small, seemingly
inconsequential details of the procedure from an understanding of the general method.
Computers, however, possess no such intelligence. A computer program is a procedure
in its most rigorous, literal form; an algorithm is a specific procedure used by a
computer program to solve a problem.

To understand algorithms, we must first understand the mechanisms by which they are
executed. A computer holds information in storage – variables, in common
programming terminology. A program, at least in a procedural programming language,
consists of a series of statements that are executed in sequence in order to modify
variables based on computations, as well as to perform various other tasks, such as
reading data from and writing data to the outside world. Various special statements,
called “flow control statements,” may modify the sequential flow of execution. Such
statements include loops, which may execute a sequence of statements multiple times
depending on various conditions, and if-statements, which conditionally execute blocks
of statements depending on various conditions.

Many such programming languages exist; in the course of developing a precise
algorithm to solve this problem, we will use Python, a well-known dynamic procedural
language.

The Sudoku Algorithm

Recall that we have defined a Sudoku puzzle as a 9x9 grid of cells, each of which may
contain a digit. Recall also that in the course of describing our general procedure to
solve the puzzle, we found lists of possibilities for each cell. We will start our precise
definition of the solver algorithm by defining our data structures – the exact ways in
which our variables hold our data. Rather than keeping simply a grid of cells holding
single digits in memory, we will keep a grid of lists; each list will contain the
possibilities for that cell. In addition, we will keep a grid of possibility counts – the
number of possibilities for each cell – and a single number, the count of “knowns,”
where a “known” is defined as a cell with exactly one possibility.

The first fundamental operation we will define is the “reduce” operation. Intuitively,
reducing the grid is simply discovering what we can without specifically choosing single
possibilities out of lists of multiple choices. The reduce operation, in Python, is given in
Listing 1.
# finds possible elements for each position

```python
def reduce(self):
    self.known = 0
    # for each cell:
    for i in range(9):
        for j in range(9):
            # skip this cell if it's already known
            if self.count[i][j] == 1:
                self.known = self.known + 1
                continue
            # start with all possible
            possible = {1: 1, 2: 1, 3: 1, 4: 1, 5: 1, 6: 1, 7: 1, 8: 1, 9: 1}
            # check row
            for n in range(9):
                if self.count[i][n] == 1:
                    x = self.cells[i][n].keys()[0]
                    if possible.has_key(x):
                        del possible[x]
            # check column
            for n in range(9):
                if self.count[n][j] == 1:
                    x = self.cells[n][j].keys()[0]
                    if possible.has_key(x):
                        del possible[x]
            # check inner box
            box_i = i - (i % 3)
            box_j = j - (j % 3)
            for a in range(box_i, box_i + 3):
                for b in range(box_j, box_j + 3):
                    if self.count[a][b] == 1:
                        x = self.cells[a][b].keys()[0]
                        if possible.has_key(x):
                            del possible[x]
            # now we deal based on 'possible'
            self.count[i][j] = len(possible.keys())
            # none possible: no solutions
            if self.count[i][j] == 0:
                return 0
            # one possible: increase number of knowns
            if self.count[i][j] == 1:
                self.known = self.known + 1
            # store in 'cells' matrix
            self.cells[i][j] = possible
    # done: successful (ie, no impossible entries)
    return 1
```

Listing 1. reduce() method (syntax highlighting using GNU enscript)

Several things may be noted. Firstly, the “self.<variable>” idiom occurs because the solver is implemented as a class in Python, thus all instance variables are stored within “self”, the class instance. self.cells is a two-dimensional array, of which each entry is a Python dictionary – created on line 12; this simply means that it is easy for the code to
tell whether a specific digit is included in the list of possibilities, and delete a single
digit, in better than linear time. self.counts holds the possibility counts in a similar two-
dimensional array, and self.known is a simple integer variable counting knowns.

The general algorithm of this function is fairly easily described in English. First, we set
the count of knowns to zero. Then, we iterate through all cells in the grid. At each cell,
we first check to see if it is a known – that is, if there is only one possibility – and if
so, we increment the count of knowns and continue directly to the next cell, skipping
the following processing. If the cell is not a known, we start with a list of all nine digits
as possibilities, and iterate through all cells in the current cell's row, column, and 3x3
box, deleting possibilities as we encounter digits in these cells. Finally, we store the
remaining possibilities back in the cell. If there are no possibilities left, we return zero,
indicating that the given grid reduced to an impossibility. If there is exactly one
possibility, we count the new known. If all cells undergo this processing without
returning a zero (indicating an impossibility), we return a result of one, indicating that
processing completed successfully.

We must now implement the overall algorithm. Recall from earlier discussion that our
method, put simply, is to first derive all knowledge possible, and then venture
tentatively down a single path chosen from the multiple possibilities that exist. We
have precisely specified the algorithm to derive all knowledge possible; we must now
specify the precise method to try possibilities until a solution is found.

It is at this point that a well-used technique known as recursion enters the picture.
Recursion may be defined informally as repetition of or reference to an algorithm or
method from within itself. For example, one may specify the process of performing an
operation on a sequence of objects as performing the operation on the first object, then
performing the operation on the rest of the objects. This at first may seem redundant,
trivial, and useless, and entirely equivalent to simply specifying “perform the operation
on each object, one at a time.” However, a subtle distinction exists; while the latter
method of specification requires a loop, the former does not. The method is very
simple, consisting of only two operations; it just so happens that one of the operations
in the method itself. One must take care when using recursion, however, not to create
an infinite loop; at some point, the method-within-a-method stack must “bottom out.”
In the example of performing an operation on a sequence of objects, the method could
include a conditional statement to return immediately if the specified sequence is the
empty (or null) sequence; thus, when the last object is passed to the method, “the rest
of the objects” is the null sequence, and the subsequent call to the method returns
immediately. This technique is powerful enough that in the Scheme programming
language, a dialect of Lisp, recursion is preferred over all other forms of iteration.

Our algorithm, too, will use recursion. We define the overall algorithm, very simply, as
“reduce, find the first undecided cell, choose one possibility, and attempt to solve the
resulting grid.” The implementation of this is presented in Listing 2.
In order to understand this method, let us separate it into sections. First, from line 3 to line 9, we reduce the grid until we gain no new knowledge from consecutive invocations. It is necessary to try multiple times because, for example, a cell closer to
the top-left of the grid may be reducible to a fewer number of possibilities based on a
new known found closer to the bottom-right of the grid, processed after the top-right.
Thus, later passes may build on knowledge gained in previous passes. As an alternate
case, if a call to the reduce method fails, indicating an impossibility, the entire solve
method fails as a consequence.

Lines 11-13 check to see if we have 81 knowns – that is, if the grid is completely
known. This is our check on the recursion to keep it from proceeding indefinitely – or
one of them, at least. The control flow may bottom out either here, at line 49, if a
recursive call returns a completely solved grid, or at line 52, if the control flow falls
through all recursive calls on possibilities and none work – thus returning an
unsuccessful return code.

Lines 15-30 scan the cells to find one with more than one possibility. Of special note is
the logic involving the “start” parameter; this logic is to prevent this loop from re-
scanning possibilities in recursive calls. A recursive call to the solve method will specify
a “start” of the possibility that is being tried, plus one.

Lines 32-36 are simply a sanity check, to ensure the consistency of what our data
structures are telling us. Although this sanity check failed several times – thus giving
useful information! - during the development of the algorithm, it should never fail with
the current hopefully completely correct implementation. Redundant techniques such as
this, in attempts to failsafe code, are important in practical software engineering,
although not strictly necessary in the theoretical development of algorithms and the
implementations thereof.

Finally, we come to the core of the recursion, at lines 38-49. We iterate through all
possibilities for the cell that we chose previously, and for each one we create a tentative
copy of our data, set the cell in question to the chosen possibility, and recursively call
the solve method, specifying the “start” parameter as one more than the location of the
cell in question. If we obtain a solution given the chosen possibility, we set our own
data to the now-solved tentative data copy and return a successful return code.

Finally, at lines 51-52, if no possibility yields a solution, we fall through to an
unsuccessful return code.

Let us now take a moment to reflect: why, exactly, does this work? A thoughtful
reader may think to him or herself, “We are assuming things, are we not, when we
choose a possibility. How do we know that this assumption is valid?”

The key to understanding this is to realize that we do not know whether the
assumption is valid. We determine its validity either positively, if a solution yielded by
the chosen possibility affirms it, or negatively, if the lack of solutions proves it invalid.
The beauty of the recursive approach is that all possibilities for the entire grid are tried
implicitly, without an explicit search of the entire problem domain, and dead ends are
eliminated quickly, as soon as the reduce method determines an impossibility.
A tie-in may be made at this point to *Godel, Escher, Bach: An Eternal Golden Braid* by Douglas R. Hofstadter. In his chapter on Propositional Calculus, Hofstadter presents the idea of “fantasies.” A fantasy is an assumption leading to a conclusion; logically, one may not take the conclusion as truth outside the context of the assumption. However, one may take a sentence stating “if the assumption is true, then the conclusion is true” as truth, even outside this context. Similarly, the recursive solve method makes an assumption – trying a possibility – and derives a conclusion. This conclusion may be either positive – a completely solved grid – or negative. If the conclusion is positive, then one may take the statement “If the given possibility is chosen, then a specific known solution exists.” More than likely several of these if-then statements will be nested within each other; an example might be “If cell 1 is 5, then if cell 2 is 3, then if cell 4 is 8, then if cell 6 is 9, then if cell 7 is 4, then if cell 8 is 2, ... then the solution <specific solution grid> exists” (note that certain cells will not have a corresponding if-clause because they are given in the problem).

On the surface, such a statement might not appear to provide much value. However, the if-clauses may be trivially stripped from the statement using only a bit of logical thinking. The statement above says, essentially, that if each of the cells is set to a specific value, then a solution exists with those specific values. However, the problem definition leaves the solver completely free to assign any value whatsoever to the blank cells, given that they fit the constraints of the puzzle. Because the solution grid exists with the numbers in the cells, the numbers fit the constraints; thus, we realize that the statement reduces to “if the grid is populated with this valid configuration, the resulting configuration is a solution.” This is precisely what we are looking for: a solution. Intuitively, one may also strip away these if-clauses by reasoning that a solution is valid no matter the context – the validity depends in no way on the assumptions given in the if-clauses, thus the solution is universally valid.

Truly, recursion is a powerful technique, usable to solve a large portion of simple puzzles such as Sudoku.

The author's complete solver implementation, and its online interface, may be found at [http://www.c1f.net/misc/sudoku.cgi](http://www.c1f.net/misc/sudoku.cgi).