A Sudoku Solver

Sergey Gubin *

Abstract—**Article describes a Sudoku solver based on the compatibility matrix method.**

Keywords: Sudoku; Algorithm; Compatibility Matrix

1 Introduction

In this work, we use the compatibility matrix method [1, 2, 3, and others] and develop an algorithm for Sudoku (see Appendix). The compatibility matrix method may be described as follows.
For a decision/search problem, we partition the guesses in such a way that the number of the parts will be manageable with resources available and the parts' domains will be manageable as well. The partition reduces the initial problem to a detection of non-contradicting combinations of the parts' meanings. Obviously, any such *compatible combination* of the parts' meanings is a solution for the initial problem.
Basically, the compatibility matrix method reduces the initial problem to a search for non-trivial solutions of a special kind for the following Horn 2SAT instance:

$$\bigwedge_{(i_1,j_1,i_2,j_2)\in S} (\bar{x}_{i_1j_1} \vee \bar{x}_{i_2j_2}) = true$$

- where x_{ij} are Boolean variables and set S represents the given problem (variable x_{ij} is indicator for the j-th meaning of the i-th part of the guesses). So, it is no wonder that the method can be successful especially when the guess parts are *well intertwine*, and especially when the initial problem has no solutions at all (if not always then often, the guessing can be reduced to such a problem).
This article may be seen as an illustration to the compatibility matrix method. We apply the method to Classical Sudoku (see Appendix). In obvious way, it can be adjusted to other Sudoku variants and to Latin Square per se. Although, the resulting algorithm is not like [4] a "pencil-and-paper" algorithm, yet it can be realized in client side JavaScript [6].

2 Compatibility Matrix

For any given *input* (the initial partial filling of the square), *guess* is any complete filling of the square with numbers from 1 through 9. There are 81^9 guesses in total.
Testing of a guess is checking whether or not this guesses contains just one number from 1 through 9 in each row, in each column, and in each little square 3×3; and checking whether or not the guess extrapolates the given input.
We divide each of the guesses in 81 parts. The parts are the cells of the square. Each part may take value from 1 through 9: set $\{1,2,\ldots,9\}$ is domain for each of the parts.
Let's index the parts with four indexes: part P_{RCrc} is the (r,c)-th cell in the (R,C)-th little square 3×3, where

$$(R,C,r,c) \in \{1,2,3\}^4 \qquad (1)$$

For each couple of the 81 parts, let's build a *compatibility box.*.
The compatibility box for couple

$$(P_{R_1C_1r_1c_1}, P_{R_2C_2r_2c_2})$$

is a 9×9 Boolean matrix (its elements are $true$ of $false$, but we will depict the values with 1 and 0, appropriately)

$$B_{R_1C_1r_1c_1,R_2C_2r_2c_2}$$

with elements

$$B_{R_1C_1r_1c_1,R_2C_2r_2c_2,ij},\ i,j=1,2,\ldots,9$$

The value of an element is $true$ of $false$ (again, we will depict it with 1 and 0, appropriately) depending on whether or not values i and j in the cells contradict each other according to Sudoku rules (we still ignore the input):

$$B_{R_1C_1r_1c_1,R_2C_2r_2c_2,ij} = \qquad (2)$$

$$\begin{cases} 0, & i=j \ \wedge\ (R_1=R_2 \wedge r_1=r_2 \ \vee\ C_1=C_2 \wedge c_1=c_2) \\ 0, & i=j \ \wedge\ R_1=R_2 \ \wedge\ C_1=C_2 \\ 1, & \text{Otherwise} \end{cases}$$

The first and the third strings in this definition represent the Latin Square constrains (each number meets just once in each row/column); the second and third strings represent the Classical Sudoku constrains (each number meets just once in each little square 3×3).
There are 81^2 the compatibility boxes in total (by the number of couples of the cells; the cells in a couple may be the same). Let's aggregate these boxes in some consistent way in a box matrix B whose size in boxes will be 81×81. There will be $81^3 = 531,441$ scalar elements in that box matrix.
For example, we could see tuples 1 as the numbers written in base-4 and sort the cells in ascending order. Then,

*Alcatel-Lucent, Daly City, CA USA, sgubin@genesyslab.com

we could aggregate the compatibility boxes in the box matrix in accordance with the order. Anyway, we call the resulting box matrix a *compatibility matrix* for Classical Sudoku appropriate to the selected partition of the guesses:

$$B = (B_{R_1 C_1 r_1 c_1, R_2 C_2 r_2 c_2})_{81 \times 81}$$

- where size of the matrix is shown in boxes. In the matrix, all diagonal boxes are the identity matrices (in the boxes all diagonal elements equal *true* and all non-diagonal elements equal $false$). And in the non-diagonal boxes all non-diagonal elements equals *true*. In those diagonal boxes where

$$R_1 = R_2 \ \vee \ C_1 = C_2 \tag{3}$$

all diagonal elements equal $false$. And in the rest of the non-diagonal boxes, all diagonal elements equal *true*.

3 Solution Grids

Any legitimate filling of the Sudoku square is presented in the compatibility matrix with a grid of *true*-elements (the elements depicted with 1), one element per compatibility box. Any such grid of elements may be presented as a solution of the following functional equation:

$$B_{R_1 C_1 r_1 c_1, R_2 C_2 r_2 c_2, f(R_1 C_1 r_1 c_1) f(R_2 C_2 r_2 c_2)} = true \tag{4}$$

- where function f is the unknown. And visa versa, any solution of functional equation 4 represents a legitimate filling of the Sudoku square. We call any solution of equation 4 a *solution grid* for Sudoku. The solution grids are in one-to-one relation with the legitimate fillings of Sudoku square.
Any legitimate Sudoku input (the initial partial filling of the square) is a part of a solution grid (it is a partially given solution f of system 4). The input may seen as *initial conditions* for equation 4:

$$B_{R_0 C_0 r_0 c_0, R_0 C_0 r_0 c_0, i_0 i_0} = true \tag{5}$$

- where the 0-indexes are indexes of cells filled in the given initial setting. So, solving of the puzzle is just extrapolation of the given input 5 to a whole solution grid. We may use dynamic programming for the extrapolation.
In those diagonal compatibility boxes which are appropriate to the cells participating in the input, we replace all *true*-elements with value $false$ except that one element which corresponds to the number assigned to this cell in the initial setting. Then, we propagate these values $false$ along those rows and columns which come through these diagonal elements. Because there are such compatibility boxes whose indexes satisfy clauses 3, the propagation may nil some rows/columns in these non-diagonal compatibility boxes entirely. So, we may continue and propagate value $false$ along the appropriate rows/columns from these nil rows/columns into the rest of the boxes. And so on.

We call this dynamic programming procedure *depletion* of compatibility matrix. Depletion is just replacement some of the *true*-elements in the compatibility matrix with value $false$ in a way preserving at least one solution grid.
There is less than 81^3 elements to deplete in the matrix. So, sooner or latter, the matrix will be finalized. There are only three options for the outcome:

1) All elements in the final matrix equal $false$. This is a case of an incorrect input - the puzzle has no solutions. The first nilled compatibility box arising during the depletion may be called *a pattern of unsatisfiability* of the given input. When such pattern arises, the depletion can be stopped, indeed.

2) All *true*-elements in the final matrix create one solution grid. This a pure-logical case of the correct input - the puzzle has unique solution and search for it did not require any guessing.

3) There still will be uncertainty, i.e. some of the diagonal compatibility boxes will have more than one *true*-element preserved after the depletion.

The third case also could be tried with pure-logic. It just needs to *intertwine* the cells in the square, i.e. it needs to include in our guess partition some combinations of those cells - the final compatibility matrix shows which cells (and possible values) shall be combined. But, such approach would make the algorithm input-related. So, let's just guess.
In any "uncertain" diagonal compatibility box, let's select any of the *true*-elements and nil the rest of the *true*-elements in the box. Let's propagate these new $false$-values exactly as the above. If such depletion will fill the compatibility matrix with value $false$ entirely, then our guess was wrong. So, we just nil this diagonal element and propagate its value $false$. Then, we try the next "uncertain" diagonal element, and so on until the compatibility matrix will be finalized.
Let's notice that we could use in the guessing non-diagonal compatibility boxes as well: we could nil all elements in a non-diagonal box except the one; then, we would propagate those values $false$ exactly as the above and see outcome.
Again, there are three possible outcomes of the depletion:

1) All elements in the final matrix equal $false$. This is a case of an incorrect input - the puzzle has no solutions.

2) All *true*-elements in the final matrix create one solution grid. This a case of a correct input - the puzzle has unique solution.

3) There still is uncertainty, i.e. some of the diagonal compatibility boxes have more than one *true*-element left in them after the depletion. This is a

ISBN:978-988-17012-6-8

case of an incorrect input - solution of the puzzle is not unique.

The final matrix may be called a *general solution* for the given input - it is conjunction of all solution grids satisfying the puzzle.
Obviously, there are many ways to rationalize the procedure. Let's see some of them.

4 Making it more feasible

To save space, let's partition our guess differently. Let's partition them just in 9 parts appropriate to the numbers from 1 trough 9. Domain for each of such parts will be the 81 cells in the Sudoku square. Also, let's factorize/partition the domain.
Let's select in the Sudoku square some coordinate lines. For example, we may select just rows and columns and present each cell as a couple of numbers - row number and column number. There are 18 such coordinate lines in total. And each of those coordinate lines is a part of our domain.
To better *intertwine* the domain's parts, i.e. to make the following depletion more efficient, we even may select some curve coordinates in the Sudoku square. The "straight" and some curve coordinates are shown below. In the shown curve coordinates, there are 36 coordinate lines in total - each row/column of the 3×3 squares is described by 6 permutations of its "little" rows/columns appropriately.
"Straight" coordinates:

				*				
				*				
				*				
				*				
*	*	*	*	+	*	*	*	*
				*				
				*				
				*				
				*				

Curve coordinates:

			*					
			*					
			*					
*	*	*		*				
			*	×	*			
				*		*	*	*
					*			
					*			
					*			

Whatever coordinates we would choose, let's combine for each of the numbers form 1 trough 9 the compatibility matrix for those coordinate lines (they are the parts of the domain for each number).
Let's notice that all "vertical" coordinates are independent among themselves. The same is true for the "horizontal" coordinates. So, the only essential part of the compatibility matrix will be its *incidence* part - the part consisting of the compatibility boxes connecting the "vertical" and "horizontal" coordinate lines. For the straight coordinates, the matrix is shown below:

n		1			2			3		
		c_1	c_2	c_3	c_1	c_2	c_3	c_1	c_2	c_3
1	r_1 r_2 r_3		$C_{n,1,1}$			$C_{n,1,2}$			$C_{n,1,3}$	
2	r_1 r_2 r_3		$C_{n,2,1}$			$C_{n,2,2}$			$C_{n,2,3}$	
3	r_1 r_2 r_3		$C_{n,3,1}$			$C_{n,3,2}$			$C_{n,3,3}$	

- where C_{nij} is the compatibility box for i-the "horizontal" coordinate line and j-th "vertical" coordinate line for number n ($n = 1, 2 \ldots, 9$). For the "straight" coordinates these boxes have size 3×3, and they will have size 6×6 for the curve coordinates shown above. Let C_n be this box matrix ror number n:

$$C_n = (C_{nij})_{3\times 3},\ n = 1, 2, \ldots, 9$$

At the beginning, before the input, matrix C_n is entirely filled with value $true$, $n = 1, 2, \ldots, 9$.
Now, when there is an input, we enter the given initial composition in the matrices C_n by the following simple *input rule*.
For example, let there be number 9 in the given input located in the $(1, 1)$-th cell of the $(1, 1)$-th little square of size 3×3. Then, in the "straight" coordinates, there will be

$$C_{9,1,1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix};\ C_{n,1,1} = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix},\ n \neq 9$$

In the curve coordinates, there will be four values $true$ and $false$ in the upper left corner, appropriately. We enter in such way all numbers from the initial composition. After the input, the values $true$ and $false$ will be distributed over matrices C_n in some order. Obviously, all the solution grid theory is true for the matrices C_n. So, to resolve the input, we need to search all 9 matrices C_n for such solution grids which would not contradict one another. For that we may use the following depletion.
We will propagate the value $false$ over matrices C_n iterating each of the four compatibility boxes

$$C_{nij},\ C_{nik},\ C_{nmj},\ C_{nmk}$$

by formulas

$$\begin{array}{lcll} C_{nij} & = & C_{nij} & \wedge \quad (C_{nik}C_{nmk}^{T}C_{nmj}) \\ C_{nik} & = & C_{nik} & \wedge \quad (C_{nij}C_{nmj}^{T}C_{nmk}) \\ C_{nmj} & = & C_{nmj} & \wedge \quad (C_{nmk}C_{nik}^{T}C_{nij}) \\ C_{nmk} & = & C_{nmk} & \wedge \quad (C_{nmj}C_{nij}^{T}C_{nik}) \end{array} \quad (6)$$

- where the parenthesis operations are Boolean matrix multiplication (see Appendix). The major benefit of these formulas - they preserve all solution grids.

We iterate all matrices C_n by formulas 6 until they get finalized. In the outcome, positions of some numbers from 1 trough 9 can be detected exactly. Then, we propagate those position for the elements in the rest of matrices C_n by the above input rule and repeat the iterations until all matrices will be finalized. In total, there are just 720 scalars to deplete in the case of "straight" coordinates, and there are just $2,916$ scalars to deplete in the case of the curve coordinates. So, it will not take long to finalize the matrices.

We don't use here any Sudoku "governing dynamics". So, the depletion can have three outcomes:

1) All elements in some matrix C_n equal $false$. This is a case of an incorrect input - the puzzle has no solutions (there is no place for that number n).

2) All $true$-elements in each of the final matrices C_n create one solution grid. This a pure-logical case of the correct input (formulas 6 are just a form of deduction) - the puzzle has unique solution and search for it did not require any guessing (would these solution grids incompatible, we would get in case 1).

3) There still will be uncertainty, i.e. some of the compatibility boxes will have more than one *true*-element preserved after the depletion. This case requires additional work.

Let's notice that there are at most $3^6 = 729$ solution grids in each matrix C_n in the case of "straight" coordinates, and there are at most $6^6 = 46,656$ solution grids in the case of the curve coordinates. So, it will not take long to find and test all of them as follows.

For each of the matrices C_n, we may use brute force and find all solution grids in it. Each such grid presents all 9 positions for number n in Sudoku square. We use the above input rule and enter all these 9 positions in all matrices C_n. Then, we deploy the above depletion. If a solution grid nils some of the matrices, then we rid of it. The resulting version of matrix C_n is disjunction of some solution grids. We repeat this procedure for all matrices C_n until they all will get finalized.

Again, the procedure can have three outcomes listed above. But now, the third outcome would mean that the given input allows many solutions.

5 Making it smarter

There are 27 "geographical features" in Sudoku square (9 little squares 3×3, 9 rows, and 9 columns) where each number may be met just once. But, the same is true for any combination of the numbers from 1 trough 9. There are just $2^9 - 1 = 511$ such combinations. So, we may include in the algorithm the allocation of those combinations as well. But, there is trade-off: complexity of the logic can slow down the depletion.

6 Conclusion

We described as the compatibility matrix method can be deployed to Sudoku. The method reduces the puzzle to solution of functional equation 4 with initial conditions 5. The equation can be solved by depleting the compatibility matrix. The result of the depletion is the general solution of the puzzle.

The compatibility matrix method may be seen as the following framework for NP-problems.

The goal of the method is to organize parallel testing of all guesses. For that, we partition the guesses. Formally, the partition is just several factorizations of the set of all guesses. Each of the factorizations is a part. Domains of the parts are the appropriate factor sets. The only requirements is the number of the factorizations and the total number of the cossets shall be *manageable* with resources available; and the whole guesses could be restored from their parts.

The most natural partition is the real one. Suppose, the guesses are some strings in a language. Then, we may equalize those strings which coincide in certain positions. Effectively, the parts in this case will be the shorter strings. So, goal of such partition will be to divide those long strings in these shorter strings in such a way that the number of those parts and the number of those parts' meanings both will be manageable. If we could find the parts belonging to a solution, then we would glue the whole solution from those shorter strings.

Then, we "translate" the given checking condition in a relation of *compatibility* (non-contradiction) on those parts. For each two parts the relation is a Boolean matrix - the compatibility box.

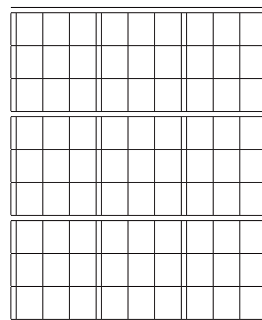
Let's return to our language example. Why a guess is not a solution? Most likely, it contains wrong substring. And we can detect this substring - it's a NP-problem. Thus, those meanings of parts will be incompatible (contradicting one another) which produce that wrong substring when glued.

Then, we combine the compatibility boxes in a box matrix - the compatibility matrix. The matrix encodes the problem in contradictions between parts of the guesses. Any solution creates in the matrix a pattern - the solution grid. And vise versa, any solution grid delivers a solution for the problem. So, solving of the problem is searching of the compatibility matrix for the solution grids. Set of

all solution grids is a *general solution* of the problem.
The search for solution grids always can be done efficiently. Actually, the search is a linear problem whose size is polynomial over the size of the compatibility matrix [1, 2, 3]. So, we could solve Sudoku solving a system of linear equations and inequalities.
Really, the partition effectively creates in space of all guesses a system of coordinates. In the coordinates, set of all solutions is a set of points. For a NP-problem, the number of points in the set may be exponential and even factorial. Nevertheless, there is bijection of the set on the vertices of such a (convex) polytope which can be described with a polynomial number of linear equations and inequalities[1]. And the compatibility matrix is very instrumental in finding of such polytope.
Depletion is another general approach to the search for solution grids. Formally, it is just replacement of some *true* elements in the compatibility matrix with value *false*. The goal is to transform the compatibility matrix in disjunction of all solution grids - the general solution of the problem. The obvious benefit of the depletion - it is more efficient than solution of the linear system. Most painful drawback - it is not always clear which elements to inverse. The Boolean matrix multiplication 6 (and its analogs) preserves all solution grids. But, it can miss some elements which shall be inversed. In the terms of the language example, we do not know any relation between the guess' parts and the wrong substrings. A priory, there can be situation when no two parts will cover any wrong substring. We describe such foul cases as cases when the guess' parts are not *well intertwine*. The general recipes here may include mixing of the parts, creating another partition and taking conjunction of the outcomes, etc.
Mechanically, the depletion misses some elements when there are too much *true* elements in the compatibility boxes. So, another approach is to try and test each element in the compatibility boxes separately. The elements shall be tested on belonging to a solution grid.
Above for Sudoku, we have fixed such partition which allowed the testing of large fragments of the would-be solution grid.

Appendix

Latin Square. There is given a square matrix $n \times n$. The matrix is partially filled with some numbers from 1 through n. The problem is to entirely fill the matrix with the numbers in such a way that each number meets just once in each row and in each column.
Being centuries old, Latin Square is not just a puzzle. Completing of Latin Square is a NP-complete problem [5]. So, many practical and theoretical problems are reducible to it.
Sudoku variants. The most of the puzzles may be seen as a Latin Square with additional constrains: the numbers meet just once in certain areas of the square. There are many such puzzles. They differ by the size of the square and by the shapes of the areas.
The *input* (the initial partial filling of the square) is sad to be *correct* if it provides just one solution for the puzzle.
Again, many practical and theoretical problems can be modeled with Sudoku variants.
Classical Sudoku. The puzzle is Latin Square 9×9 divided in 9 evenly distributed squares 3×3:



The problem is to complete the square in such a way that each number meets just once in each of the 3×3 squares, i.e. it meets just once in each row, each column, and each little square.
The puzzle was invented by Howard Garns in 1979, and it became very popular. Today, there are Sudoku communities, tournaments, TV shows, etc. Among literature, let's us mention [4, 7, 8, 9, and many others]. Much more information and references may be found in Wikipedia.
Boolean matrices. Boolean matrices are matrices whose elements are *true* or *false*. Conjunction and disjunction of such matrices is just a matrix of the conjunctions and disjunctions of the appropriate elements (the matrices have to be of the same size). Multiplication of such matrices can be defined exactly as multiplication of the numerical matrices with replacement of operations + and $\times$ with the appropriate Boolean operations. Here, we use the following definition:

$$XY = (\bigvee_{\mu} x_{i\mu} \wedge y_{\mu j})$$

- where X and Y are Boolean matrices of the appropriate sizes (the number of columns in X equal the number of rows in Y).

[1] A well known example of such polytopes is Birkhoff polytope. It has $n!$ vertices, yet it is described with $2n$ linear equations and n^2 linear inequalities.

References

[1] Sergey Gubin, *Polynomial size asymmetric linear model for Subgraph Isomorphism*, Proc. of WCECS 2008, ISBN:978-988-98671-0-2, pp. 241 - 246 (see arXiv:0802.2612v2 [cs.DM])

[2] Sergey Gubin, *Complementary to Yannakakis' theorem*, 22-nd MCCCC, Program and Abstracts, UNLV 2008, p. 8 (see arXiv:cs/0610042v3 [cs.DM])

[3] Sergey Gubin, *Polynomial size asymmetric linear model for SAT*, In Proc. of the Advances in Electrical and Electronics Engineering - IAENG Special Edition of the World Congress on Engineering and Computer Science 2008 (WCECS 2008). (accepted)

[4] J. F. Crook, *A Pencil-and-Paper Algorithm for Solving Sudoku Puzzles*, Notices of the AMS, Volume 56, Number 4 pp.460 - 468, 2009

[5] C. Colbourn, *The complexity of completing partial latin squares*, Discrete Applied Mathematics 8: pp. 2530, 1984

[6] *Sudoku Solver*, www.timescube.com (pending)

[7] Michael Mepham, *Solving Sudoku*, Crosswords Ltd., Frome, England. 2005, (see `http://www.sudoku.org.uk/PDF/Solving_Sudoku.pdf.`)

[8] Denis Berthier, *The Hidden Logic of Sudoku*, May 2007, Lulu.com, ISBN : 978-1-84753-472-9

[9] Agnes M. Herzberg and M. Ram Murty, *Sudoku squares and chromatic polynomials* , Notices Amer. Math. Soc. 54(6), pp. 708717, 2007