Sudoku and Latin Square solvers

1. Introduction:

Sudoku is a Japanese number-placement puzzle. The most common Sudoku puzzle consists of a 9×9 grid such that each row, column and 3×3 box consists of numbers 1 to 9. The puzzle is given as a partially completed grid and the aim is to fully complete it so that it satisfies the above constraint. Latin square is very similar to Sudoku, and involves an $n \times n$ array filled with n different Latin letters, each occurring exactly once in each row and exactly once in each column (of course other symbols can be used instead of Latin letters). In fact, Sudoku is a type of Latin square with an additional constraint on the contents of individual regions (boxes).

There are several algorithms designed to solve these NP-complete problems. Today we will see how we can solve them by the Algorithm X given by Donald Knuth in the year 2000[1]. For this algorithm we need to represent Sudoku (as well as latin squares) as an exact cover problem. So now let us look at the definition of an exact cover problem and Algorithm X for solving it.

2.1 Exact cover:

Given a set of elements X and a collection S of subsets of X, an exact cover is a sub-collection S* of S such that each element in X is contained in exactly one subset of S* [5]. Formally we can say that S* should satisfy following two conditions:

1. The intersection of any two distinct subsets in S* is empty, i.e., the subsets in S* are pairwise disjoint. In other words, each element in X is contained in at most one subset in S*.

2. The union of the subsets in S* is X, i.e., the subsets in S* cover X. In other words, each element in X is contained in at least one subset in S*.

If an empty set is contained in S* it makes no difference. Thus we assume that there is no empty set in S.

2.2 Example of exact cover:

Consider a set of elements $X = \{1,2,3,4,5,6,7\}$ and a collection of subsets $S = \{A,B,C,D,E,F\}$ such that,

- $A = \{1, 4, 7\}$;
- $B = \{1, 4\}$;
- $C = \{4, 5, 7\}$;
- $D = \{3, 5, 6\}$;
- $E = \{2, 3, 6, 7\}$; and
- $F = \{2, 7\}$.

Then the collect $S^* = \{B,D,F\}$ is an exact cover. Do you see why?

Moreover, this is the only exact cover for the above problem. Because A and B are the only subsets containing 1, an exact cover must contain A or B, but not both. If an exact cover contains A, then it doesn't contain B, C, E, or F, as each of these subsets has an element in common with A. Then D is the only remaining subset, but the collection {A, D} doesn't cover the element 2. In conclusion, there is no exact cover containing A. On the other hand, if an exact cover contains B, then it doesn't contain A or C, as each of these subsets has an element in common with B. Because D is the only remaining subset containing 5, D must be part of the exact cover. If an exact cover contains D, then it doesn't contain E, as E has an element in common with D. Then F is the only remaining subset, and the collection {B, D, F} is indeed an exact cover.

3.1 Exact Hitting set:

Given a collection S of subsets of a set X, an exact hitting set X* is a subset of X such that each subset in S contains exactly one element in X* [5]. This will be clear from the example.

An exact cover problem involves selecting subsets whereas; an exact hitting set involves selecting elements.

3.2 Example of hitting set:

Consider the same example as in exact hitting set with set of elements X = {1,2,3,4,5,6,7} and a collection of subsets S = {A,B,C,D,E,F} such that,

- A = {1, 4, 7};
- B = {1, 4};
- C = {4, 5, 7};
- D = {3, 5, 6};
- E = {2, 3, 6, 7}; and
- F = {2, 7}.

Here, X* = {1, 2, 5} is an exact hitting set, since each subset in S contains exactly one element in X*.

Moreover, {1, 2, 5} is the only exact hitting set, as the following argument demonstrates: Because 2 and 7 are the only elements that hit F, an exact hitting set must contain 2 or 7, but not both. If an exact hitting set contains 7, then it doesn't contain 1, 2, 3, 4, 5, or 6, as each of these elements are contained in some subset also containing 7. Then there are no more remaining elements, but {7} is not an exactly hitting set, as it doesn't hit B or D. In conclusion, there is no exact hitting set containing 7. On the other hand, if an exact hitting set contains 2, then it doesn't contain 3, 6, or 7, as each of these elements are contained in some subset also containing 2. Because 5 is the only remaining element that hits D, the exact hitting set must contain 5. If an exact hitting set contains 5, then it doesn't contain 4, as both hit C. Because 1 is the only remaining element that hits A, the exact hitting set must contain 1. Then there are no more remaining elements, and {1, 2, 5} is indeed an exact hitting set.

4. Inter-conversion between exact cover and exact hitting set problem:

An exact hitting set problem is the inverse of the exact cover problem involving the same set and collection of subsets. Thus, we can convert an exact hitting set problem to an exact cover problem and vice versa. Let us see this with an example.

Again consider the above example. But this time we will change the representation of the problem as follows:

Let the elements become subsets and subsets become element.

Thus we have S = {I,II,III,IV,V,VI,VII} and X={a,b,c,d,e,f}.

Here, as the subset B contained the elements 1 and 4 in the exact cover problem, the subsets I and IV contain the element b. Thus we have,

- I = {a, b}
- II = {e, f}
- III = {d, e}
- IV = {a, b, c}
- V = {c, d}
- VI = {d, e}
- VII = {a, c, e, f}

Thus X* = {b,d,f} which was also the answer of the exact cover problem with S*={B,D,F}.

5.1 Knuth's Algorithm X:

Now after looking at the exact cover problem, let us see how we can solve this problem algorithmically. First we need to represent the exact cover problem in a matrix form.

The matrix includes one row for each subset in S and one column for each element in X. The entry in a particular row and column is 1 if the corresponding subset contains the corresponding element, and is 0 otherwise.

The matrix representation for our sample problem is,

	1	2	3	4	5	6	7
A	1	0	0	1	0	0	1
B	1	0	0	1	0	0	0
C	0	0	0	1	1	0	1
D	0	0	1	0	1	1	0
E	0	1	1	0	0	1	1
F	0	1	0	0	0	0	1

The Algorithm was named "X" for the lack of a better name! This is a recursive, nondeterministic, depth first, backtracking algorithm. Algorithm X is the statement of obvious trial and error approach. [1][7]

The pseudo code for the algorithm is as follows:

Here matrix A is the exact cover problem.

1. If the matrix A is empty, the problem is solved; terminate successfully.
2. Otherwise choose a column c (deterministically).
3. Choose a row r such that Ar, c = 1 (non-deterministically).
4. Include row r in the partial solution.
5. For each column j such that Ar, j = 1,
   for each row i such that Ai, j = 1,
   delete row i from matrix A;
   delete column j from matrix A.
6. Repeat this algorithm recursively on the reduced matrix A.

The nondeterministic choice of r means that the algorithm essentially clones itself into independent sub algorithms; each sub algorithm inherits the current matrix A, but reduces it with respect to a different row r. If column c is entirely zero, there are no subalgorithms and the process terminates unsuccessfully.

The subalgorithms form a search tree in a natural way, with the original problem at the root and with level k containing each subalgorithm that corresponds to k chosen rows. Backtracking is the process of traversing the tree in preorder, "depth first."

Any systematic way of choosing column c will find all the solutions. But the method of choosing columns with minimum number of ones leads to fewest branches.

5.2 Step by step implementation of Algorithm X:

Step by step implementation of Algorithm X for the above example (matrix) is as follows:

Level 0

Step 1—The matrix is not empty, so the algorithm proceeds.

Step 2—The lowest number of 1s in any column is two. Column 1 is the first column with two 1s and thus is selected (deterministically):

	1	2	3	4	5	6	7
A	1	0	0	1	0	0	1
B	1	0	0	1	0	0	0
C	0	0	0	1	1	0	1
D	0	0	1	0	1	1	0
E	0	1	1	0	0	1	1
F	0	1	0	0	0	0	1

Step 3—Rows *A* and *B* each have a 1 in column 1 and thus are selected (nondeterministically).

The algorithm moves to the first branch at level 1…

Level 1: Select Row *A*

Step 4—Row *A* is included in the partial solution.

Step 5—Row *A* has a 1 in columns 1, 4, and 7:

	1	2	3	4	5	6	7
A	1	0	0	1	0	0	1
B	1	0	0	1	0	0	0
C	0	0	0	1	1	0	1
D	0	0	1	0	1	1	0
E	0	1	1	0	0	1	1
F	0	1	0	0	0	0	1

Column 1 has a 1 in rows *A* and *B*; column 4 has a 1 in rows *A*, *B*, and *C*; and column 7 has a 1 in rows *A*, *C*, *E*, and *F*. Thus rows *A*, *B*, *C*, *E*, and *F* are to be removed and columns 1, 4 and 7 are to be removed:

	1	2	3	4	5	6	7
A	1	0	0	1	0	0	1
B	1	0	0	1	0	0	0
C	0	0	0	1	1	0	1
D	0	0	1	0	1	1	0
E	0	1	1	0	0	1	1
F	0	1	0	0	0	0	1

Row *D* remains and columns 2, 3, 5, and 6 remain:

	2	3	5	6
D	0	1	1	1

Step 1—The matrix is not empty, so the algorithm proceeds.

Step 2—The lowest number of 1s in any column is zero and column 2 is the first column with zero 1s:

	2	3	5	6
D	0	1	1	1

Thus this branch of the algorithm terminates unsuccessfully.

The algorithm moves to the next branch at level 1…

Level 1: Select Row *B*

Step 4—Row *B* is included in the partial solution.

Row *B* has a 1 in columns 1 and 4:

	1	2	3	4	5	6	7
A	1	0	0	1	0	0	1
B	1	0	0	1	0	0	0
C	0	0	0	1	1	0	1
D	0	0	1	0	1	1	0
E	0	1	1	0	0	1	1
F	0	1	0	0	0	0	1

Column 1 has a 1 in rows *A* and *B*; and column 4 has a 1 in rows *A*, *B*, and *C*. Thus rows *A*, *B*, and *C* are to be removed and columns 1 and 4 are to be removed:

	1	2	3	4	5	6	7
A	1	0	0	1	0	0	1
B	1	0	0	1	0	0	0
C	0	0	0	1	1	0	1
D	0	0	1	0	1	1	0
E	0	1	1	0	0	1	1
F	0	1	0	0	0	0	1

Rows *D*, *E*, and *F* remain and columns 2, 3, 5, 6, and 7 remain:

	2	3	5	6	7
D	0	1	1	1	0
E	1	1	0	1	1
F	1	0	0	0	1

Step 1—The matrix is not empty, so the algorithm proceeds.

Step 2—The lowest number of 1s in any column is one. Column 5 is the first column with one 1 and thus is selected (deterministically):

	2	3	5	6	7
D	0	1	1	1	0
E	1	1	0	1	1
F	1	0	0	0	1

Step 3—Row *D* has a 1 in column 5 and thus is selected (nondeterministically).

The algorithm moves to the first branch at level 2…

Level 2: Select Row *D*

Step 4—Row *D* is included in the partial solution.

Step 5—Row *D* has a 1 in columns 3, 5, and 6:

	2	3	5	6	7
D	0	1	1	1	0
E	1	1	0	1	1
F	1	0	0	0	1

Column 3 has a 1 in rows *D* and *E*; column 5 has a 1 in row *D*; and column 6 has a 1 in rows *D* and *E*. Thus rows *D* and *E* are to be removed and columns 3, 5, and 6 are to be removed:

	2	3	5	6	7
D	0	1	1	1	0
E	1	1	0	1	1
F	1	0	0	0	1

Row *F* remains and columns 2 and 7 remain:

	2	7
F	1	1

Step 1—The matrix is not empty, so the algorithm proceeds.

Step 2—The lowest number of 1s in any column is one. Column 2 is the first column with one 1 and thus is selected (deterministically).

Row *F* has a 1 in column 2 and thus is selected (nondeterministically).

The algorithm moves to the first branch at level 3…

Level 3: Select Row *F*

Step 4—Row *F* is included in the partial solution.

Row *F* has a 1 in columns 2 and 7:

	2	7
F	1	1

Column 2 has a 1 in row *F*; and column 7 has a 1 in row *F*. Thus row *F* is to be removed and columns 2 and 7 are to be removed:

	2	7
F	1	1

Step 1—The matrix is empty, thus this branch of the algorithm terminates successfully.

As rows *B*, *D*, and *F* are selected, the final solution is:

	1	2	3	4	5	6	7
B	1	0	0	1	0	0	0
D	0	0	1	0	1	1	0
F	0	1	0	0	0	0	1

In other words, the subcollection $\{B, D, F\}$ is an exact cover, since every element is contained in exactly one of the sets $B = \{1, 4\}$, $D = \{3, 5, 6\}$, or $F = \{2, 7\}$.

There are no more selected rows at level 3, thus the algorithm moves to the next branch at level 2…

There are no more selected rows at level 2, thus the algorithm moves to the next branch at level 1…

There are no more selected rows at level 1, thus the algorithm moves to the next branch at level 0…

There are no branches at level 0, thus the algorithm terminates.

In summary, the algorithm determines there is only one exact cover: S* = {*B*, *D*, *F*}.

6. Complexity of Knuth's Algorithm X:

The exact cover problem is an example of NP-complete problem and hence the algorithm X has an exponential runtime. [8]

Let us prove for upper bounds on this algorithm.

6.1 A few notations

Let X be the universal set of all the elements.

Let S be the family of subsets of X.

Let C denote the subfamily of sets in S that contain the currently chosen element $x \in X$

[i.e. when we select a column and then look for the rows having a '1' in it, the set of these rows is denoted by C].

Let S' denote the remaining subsets i.e. $|S'| = |S| - |C|$

6.2 Explanation

At every step the algorithm will choose a column deterministically i.e. the algorithm chooses an element $x \in X$.

Then the idea is to select any one of the rows in the final solution, which implies that only one of the set containing x will be chosen (which is what we want, right?).

But then on choosing any one of these sets (non-deterministically of course), we see if the chosen set leads to the final solution i.e. making the current choice we have to recursively apply the algorithm to see if we get final solution i.e. an exact cover.

Thus we have to apply the algorithm for all the sets in C.

But each of the |C| iterations now have |S'| sets where $|S'| = |S| - |C|$.

It may happen that we get 2 or 3 or 4…..or m sets having the element x, which makes |C| = 2 or 3 or 4 …..or m.

The worst of these cases is the one with three recursive calls, i.e. when |C| = 3, leading to a time bound of $O(3^{n/3}) \approx O(1.44225^n)$.

Therefore, even the stupidest version of Algorithm X, one that avoids any chance to backtrack and always chooses its pivots in such a way as to maximize its running time, takes at most $\boldsymbol{O(3^{n/3})}$ time.

The upper bound can be obtained by the tree method as follows,

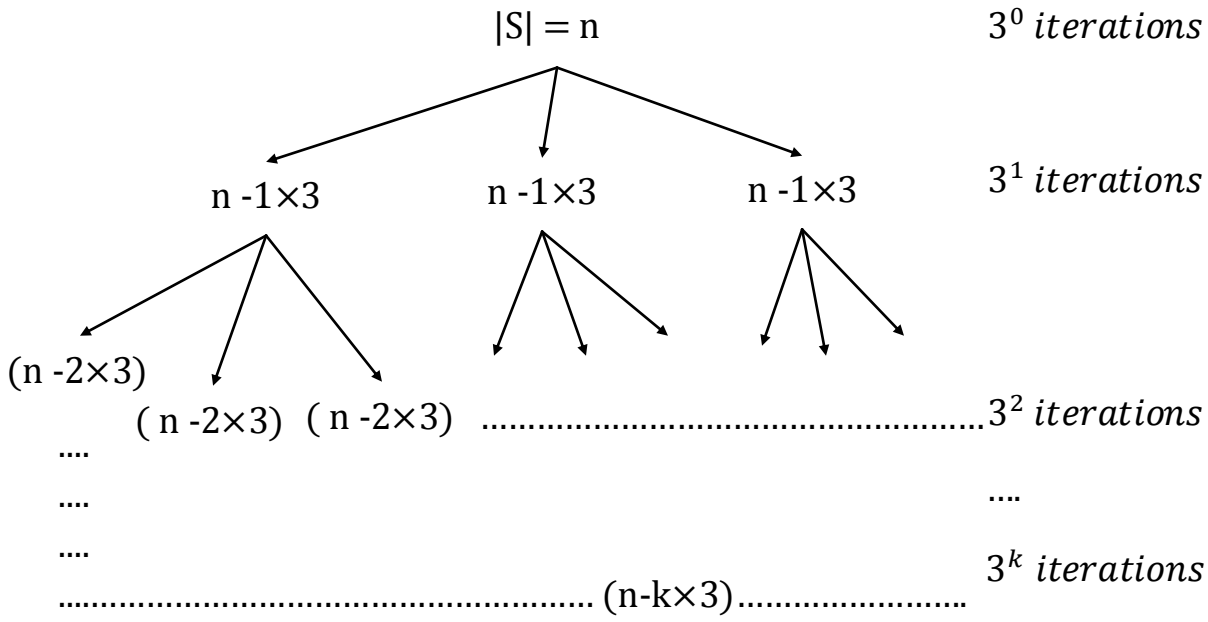

Figure 1: Tree method to calculate upper bound on Algorithm X

We will define the upper bound for the algorithm in terms of the number of subsets |S|. Initially the number of sets are |S| = n.

From figure 1 above, if 3 of the sets or rows are being chosen at every stage, i.e. |C| = 3 at every stage, then we would have 3^k Iterations at every k^{th} stage.

Thus the runtime for the algorithm would be given as,

$\sum_{i=0}^{k} 3^i$

$= 3^0 + 3^1 + 3^2 + \cdots \ldots + 3^k$

$= \frac{(3^{k+1}-1)}{2}$

Thus the run time becomes $O(3^k)$.

The maximum depth for the tree would be when $k = n/3$, that is when the number of remaining rows equal to zero.

Thus we get the **upper bound of $O(3^{n/3})$**

6.3 Why is the worst case for |C| =3?

The reason why |C| case is worst of all the cases is,

$[of\ all\ of\ these,\ 2^{1/2}, 3^{1/3}, 4^{1/4}, 5^{1/5}, \ldots\ldots, m^{1/m}]$ the value for $3^{1/3}$ is maximum.

It goes to show that by choosing a column which has a '1' in 3 rows every time, the depth and the branching of the recursion tree becomes such that we have the most number of lookups.

7. Sudoku as an exact cover problem:

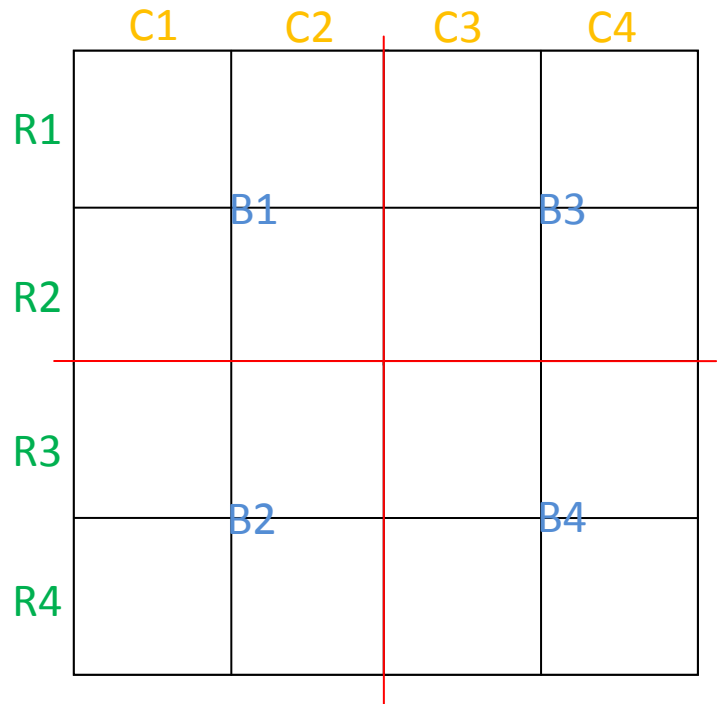

Figure 2: A 4 × 4 SUDOKU WITH ROW, COLUMN AND BOX NUMBERS

7.1 Intuition, the idea!

We will use the above 4 × 4 Sudoku example for understanding why Sudoku is an example of an exact hitting set problem.

And since an exact hitting set is equivalent to an exact cover problem (as explained above), once we prove that Sudoku is an exact hitting set problem, we can have its dual exact cover problem and use the Knuth's Algorithm X to solve it.

7.2 Understanding Constraints

Before proving for exact hitting set, we need to understand the underlying constraints for the Sudoku puzzle.

The problem in Sudoku is to assign numbers (or digits, values, symbols) to cells (or squares) in a grid so as to satisfy certain constraints. [4]

There are **4 kinds of constraints**:

Row-Column: Each intersection of a row and column, i.e., each cell, must contain exactly one number.

For the example given in figure 2 above,

$R1C1 = \{R1C1\#1, R1C1\#2, R1C1\#3, R1C1\#4\}$

[Implies that cell R1C1 can contain any of the four numbers from 1-4]

$R1C2 = \{R1C2\#1, R1C2\#2, R1C2\#3, R1C2\#4\}$

……

$R4C4 = \{R4C4\#1, R4C4\#2, R4C4\#3, R4C4\#4\}$

Thus in all we will have **16 constraint sets, one for each cell in the grid**.[1]

Row-Number: Each row must contain each number exactly once

$R1\#1 = \{R1C1\#1, R1C2\#1, R1C3\#1, R1C4\#1\}$

[Implies that row R1 can contain 1 in any of the 4 columns C1-C4]

Similarly we will have sets $R1\#2, R1\#3, R1\#4$.

And we will have **4 such sets for each of the 4 rows**, thus making **16 constraint sets in all** for this type of constraint as well.

Column-Number: Each column must contain each number exactly once.

$C1\#1 = \{R1C1\#1, R2C1\#1, R3C1\#1, R4C1\#1\}$

[Implies that column C1 can contain 1 in any of the four rows R1-R4]

Again, we will have **16 such constraint sets in all**.

Box-Number: Each box must contain each number exactly once.

$B1\#1 = \{R1C1\#1, R1C2\#1, R2C1\#1, R2C2\#1\}$

[Implies that box B1 can have one in any of the R_C_ locations]

16 sets in all.

[1] Reading Constraint Set Values: Example, Constraint set value R1C1#1 is read as, "cell located at row=R1 and column=C1 has entry=1 "(i.e. the number which appears after #).

Thus for a 4 by 4 matrix, the number of constraint sets become $16 \times 4 = 64$.

[Note: For the typical 9 by 9 Sudoku, we would have (9×9)81 constraint sets for each of the 4 types of constraints. Thus in all we would have $81 \times 4 = 324$ constraint sets.

The matrix representation would have 729×324 entries in all, and the matrix would be a sparse matrix [9]. Here 729 are the total number of possibilities considering that there are 81 cells in all with each cell having either of 1-9 numbers thus making $81 \times 9 = 729$ possibilities]

[Before moving to the next step, make sure that you understand the above representation of four types of constraints (understanding with figure2 in mind, would help here!)]

7.3 Understanding Sudoku as exact hitting set with an example

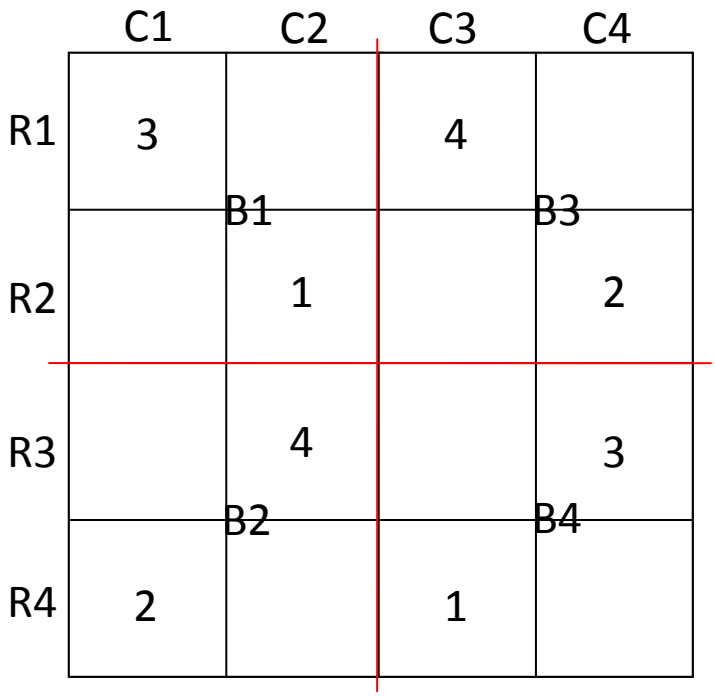

Figure 3: Example of 4 by 4 Sudoku

Let us consider the problem of placing a 1 in the box B3.

The 4 types of constraints for the above situation of the 4 by 4 Sudoku puzzle would be as follows.

1) Row-Column or Cell constraint:

$R1C3 = \{R1C3\#1, R1C3\#2, R1C3\#3, R1C3\#4\}$

$R2C4 = \{R2C4\#1, R2C4\#2, R2C4\#3, R2C4\#4\}$

2) Row-Number:

$R2\#1 = \{R2C1\#1, R2C2\#1, R2C3\#1, R2C4\#1\}$

3) Box-Number:

$B3\#1 = \{R1C3\#1, R1C4\#1, R2C3\#1, R2C4\#1\}$

If we want to find the exact hitting set over the constraint sets listed above, selecting the element 4 in the R1C3 cell (**shown by marking** the constraint value **R1C3#4 as green**) means that we cannot select any other element in the set R1C3, i.e. we cannot select any of these, $R1C3\#1, R1C3\#2, R1C3\#3$.

Thus for $B3\#1$ set, element $R1C3\#1$ cannot be selected.

Considering the selection of $R2C4\#2$ in the set $R2C4$, we cannot have $R2C4\#1$ (since we are looking for exact hitting set) and hence the possibility of $R2C4\#1$ in $B3\#1$ goes away!

Next $R2\#1$ has $R2C2\#1$ as selected, making the choice of $R2C3\#1$ not possible for B3#1 due to the properties of exact hitting set again.

Hence the only element that can be selected in B3#1 now is, ***R1C4*#1** *and thus* ***we have to place a* '1' *in the cell R1C4***

Thus the **constraint sets** are **made such that if one of the elements in the set is selected** (marked green), **then the remaining elements of that set cannot be selected** (i.e. cannot be marked green) as it would violate rules of Sudoku. (Do you see why?)

Since only one element in each set can be selected, we can see that the Sudoku problem is an example of exact hitting set problem.

Since it is an example of an exact hitting set problem, we can have its dual as an exact cover problem and solve it using Knuth's Algorithm X.

(One way to look at solving Sudoku, is that the 81 cells in case of 9×9 Sudoku, would be represented as vertices of a graph, and then the task would be to color the vertices such that adjacent vertices would be of different color. The adjacencies in this case would be defined by the row, column and box constraints. Do you see how?)

8. Discussion of Uniqueness of Sudoku

What determines the hardness of a Sudoku? To get an insight into this, we have to consider whether the given Sudoku has a unique solution. The intuition here is that even if we have less number of "givens" while solving a Sudoku puzzle, if the solution is unique, then the Sudoku puzzle might not be a hard one. On the contrary, even if the number of 'givens' are more than what you would expect for a particular Sudoku, there may be cases where you would end up guessing and backtracking to solve the Sudoku! This could be because the Sudoku has many possible solutions. [2]

This gives us a motivation to look into uniqueness of a Sudoku puzzle.

In this discussion we will consider the problem of Sudoku solving as a graph coloring problem.

8.1 Chromatic Number

A λ-coloring of a graph G is a map f from the vertex set of G to {1, 2… λ}. Such a map is called a proper coloring if f (x) $\neq$ f (y) whenever x and y are adjacent in G. The minimal number of colors required to properly color the vertices of a graph G is called the chromatic number of G and denoted χ (G).

If the Sudoku puzzle is of size $n^2 \times n^2$ (for 9×9 puzzle $n = 3$), then the degree of every vertex becomes,

$3n^2 - 2n - 1$

[The way this can be calculated is, every vertex would have $n^2 - 1$ adjacent vertices on the same row, $n^2 - 1$ adjacencies on the same column and $(n-1)^2$ adjacencies **remaining** in the same box]

8.2 Theorem:

Let G be a graph with chromatic number $\chi(G)$ and C be a partial coloring of G using only $\chi(G) - 2$ colors. If the partial coloring can be completed to a total proper coloring of G, then there are at least two ways of extending the coloring.

Proof:

Since two colors have not been used in the initial partial coloring, these two colors can be interchanged in the final proper coloring to get another proper extension.

What the theorem does not mean:

If we consider the case of 4×4 Sudoku puzzle, if 2 colors are given then we definitely don't have unique solution (by given theorem). But say we have 3 colors given as in the case below. Try to see if it results in unique solution.

1			
		4	
	3		

Figure 3: Example Sudoku, 3 colors as given, but still no unique solution.

8.3 Implication of the Theorem

This implies that if C is a partial coloring of G that can be completed uniquely to a total coloring of G, then C must use at least $\chi(G) - 1$ colors. In particular, we have that in any 9×9 Sudoku puzzle; at least eight of the colors must be used in the "given" cells. In general, for the $n^2 \times n^2$ Sudoku puzzle, at least $n^2 - 1$ colors must be used in the "given" partial coloring in order that the puzzle has a unique solution.

Thus initially if we have only (9-2) that's 7 colors for a 9×9 Sudoku puzzle; then we can say with full confidence that the Sudoku will not have a unique solution. [Though, converse will not be true as shown above]

9. Concluding Remarks

So what have we learned from all this?

Firstly, considering that to solve a Sudoku, we are going to use some kind of backtracking scheme, no matter what approach we take.

The **general idea** for Sudoku being hard seems to be that there are **few "givens"**. However if we are given an empty Sudoku, we have too much freedom of choice which makes solving easy. In the search tree for such a type of Sudoku, we could pick any of the valid leaves.

Also if there are too many "givens" or constraints, we could effectively cut off most branches of the search tree again making the Sudoku fairly easy.

If the search tree for the given Sudoku instance is such that few leaves are giving the correct solution, then we may reach at junctions in the search tree where we have to make guesses. Note that if the solution for a given Sudoku instance is unique, the number of such junctions might be lesser.

However, uniqueness alone would not be enough to determine the hardness of the puzzle. We can consider the case where we are given **all of** the numbers from **1 to 7** for a 9×9 Sudoku. This means that we need to place only 8 and 9. By the theorem explained above, we would have one solution and then we could extend it replacing the colors i.e. replacing 8 and 9. Thus in this case we don't have unique solution but still the given instance of Sudoku would be fairly easy to solve.

If we have to make guesses in the initial stages of the search tree and if that guess did not work when we reached latter stages of the search tree, we would end up backtracking a lot making the puzzle a hard one. Also, more the number of guesses or such junctions, more difficult the Sudoku would be.

As seen, solving Sudoku or Latin Squares is an np-complete problem. Another such problem is factorization problem in mathematics. In encryption schemes, such as RSA, we use the fact that finding factors for a large number is computationally hard. On the same lines, since Sudoku solving can also be computationally hard, it is being used in encryption schemes [4] and for Steganography [3] (information hiding scheme, and so the attacker is not aware that data is being sent from sender to receiver).

References:

[1]. Donald Knuth (2000) – *Dancing Links* **arXiv:cs/0011047v1**

[2]. Agnes M. Herzberg and M. Ram Murty - *Sudoku Squares and Chromatic Polynomials* http://www.ams.org/notices/200706/tx070600708p.pdf

[3] Wien Hong, Tung-Shou Chen, Chih-Wei Shiu - *Steganography Using Sudoku Revisited*

[4] Vue Wu, Sos Agaian - *Binary Data Encryption using the Sudoku Block Cipher*

Websites:

[5] Exact cover: http://en.wikipedia.org/wiki/Exact_cover

[6] Knuth's Algorithm X: http://en.wikipedia.org/wiki/Knuth%27s_Algorithm_X

[7] Sudoku Algorithms: http://en.wikipedia.org/wiki/Sudoku_algorithms

[8] Analyzing Algorithm X: http://11011110.livejournal.com/128249.html

[9] Exact Cover Matrix: http://www.stolaf.edu/people/hansonr/sudoku/exactcovermatrix.htm