For office use only	Team Control Number	For office use only
T1 _____	**233666**	F1 _____
T2 _____		F2 _____
T3 _____	Problem Chosen	F3 _____
T4 _____	**B**	F4 _____

2017

Mathematical Contest in Modeling (MCM/ICM) Summary Sheet

Sudoku Analyzing

Summary

Look at any current magazine, newspaper, computer game package or handheld gaming device and you likely find sudoku, the latest puzzle game sweeping the nation. Sudoku is a number-based logic puzzle in which the numbers 1 through 9 are arranged in a 9×9 matrix, subject to the constraint that there are no repeated numbers in any row, column, or designated 3×3 square.

In addition to being entertaining, sudoku promises valuable insight into computer science and mathematical modeling. In particular, since sudoku solving is an NP-Complete problem, algorithms to generate and solve sudoku puzzles may offer new approaches to a whole class of computational problems . Moreover, we can further explore mathematical modeling techniques through generating puzzles since sudoku construction is essentially an optimization problem.

The purpose of this paper is to propose an algorithm that may be used to construct unique sudoku puzzles with four different levels of difficulty. We attempted to minimize the complexity of the algorithm while still maintaining separate difficulty levels and guaranteeing unique solutions.

Keywords: keyword1; keyword2

Sudoku Analyzing

Kai Feng, Song Lu, Yutao Zeng

January 23, 2017

Summary

Look at any current magazine, newspaper, computer game package or handheld gaming device and you likely find sudoku, the latest puzzle game sweeping the nation. Sudoku is a number-based logic puzzle in which the numbers 1 through 9 are arranged in a 9×9 matrix, subject to the constraint that there are no repeated numbers in any row, column, or designated 3×3 square.

In addition to being entertaining, sudoku promises valuable insight into computer science and mathematical modeling. In particular, since sudoku solving is an NP-Complete problem, algorithms to generate and solve sudoku puzzles may offer new approaches to a whole class of computational problems . Moreover, we can further explore mathematical modeling techniques through generating puzzles since sudoku construction is essentially an optimization problem.

The purpose of this paper is to propose an algorithm that may be used to construct unique sudoku puzzles with four different levels of difficulty. We attempted to minimize the complexity of the algorithm while still maintaining separate difficulty levels and guaranteeing unique solutions.

Keywords: keyword1; keyword2

Contents

1 Introduction

1.1 Statement of Problem

We set out to design an algorithm that would construct unique sudoku puzzles of various difficulties as well as to develop metrics by which to measure the difficulty of a given puzzle. In particular, our algorithm must admit at least four levels of difficulty while minimizing its level of complexity.

1.2 Significance of Sudoku Research

Sudoku is one of the most pupular puzzle games of all time, which is famous for its simple rules and puzzle diversity. We think that this problem is insteresting and of great significance, due to its inherently mathematical, and offers us an oppotunity to explore new mathematical techniques. After studying, we found out that this problem could be regarded as a derivation of the Latin Square puzzle, and can be solved by using the exact-cover method.

Meanwhile, both solving and building the Sudoku puzzles are proved to be NP-Complete problems, which means that there is no known efficient way to locate an exact solution in the first place. It is apparently a great challenge for us to solve the problem in the polymonial time right now. But if we can make some progress,we may also expand into other and more practical problems. However, we shall restrict our focus directly to the problem at hand, and be content to leave these reasons, along with sudoku's entertainment value, as our motivation for exploring the game.

1.3 Sudoku Introduction

Sudoku, is a logic-based,combinatorial number-placement puzzle. The objextive is to fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 subgrids that compose the grid contains all of the digits from 1 to 9. The puzzle setter provides a partially completed grid, which for a well-posed puzzle has a unique solution. Figure.1(a) is a typical example of sudoku puzzle.

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

(a) Typical sudoku puzzle

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

(b) The same puzzle with solution

Figure 1: sudoku puzzle instance

Completed games are always a type of Latin square with an additional constraint on the contents of individual regions. For example, the same single integer may not appear twice in the same row, column, or any of the nine 3×3 subregions of the 9×9 playing board.

1.4 Notations and Terminologies

It is difficult to discuss our solution to the proposed problem without understanding some common terminology. Moreover, since we will apply more mathematical formalism here than in most documents dealing with sudoku, it will be helpful to introduce notational conventions.

- **Cell.** The basic unit of Sudoku puzzle. A square in the grid which may contain one digit(1-9). The grid is composed of 81 cells.
- **Block.** A 3×3 array of cells. Normally, the boundaries of the blocks are marked by slightly darker or thicker lines than the lines separating the cells. The grid is composed of 9 non-overlapping blocks. Each block must contain all the digits(form 1 to 9) and may not contain more than one of each digit.
- **Column.** A verticle line of 9 cells. The grid is composed of 9 columns. Each column must contain all the digits(1-9) and may not contain more than one of each digit.
- **Row.** A horizontal line of 9 cells. The grid is composed of 9 rows. Each row must contain all the digits (1-9) and may not contain more than one of each digit.
- **Grid.** The 9×9 array of cells that compose a Sudoku puzzle. The grid contains 9 rows, 9 columns and 9 blocks.
- **Puzzle.** A 9×9 matrix of cells, with at least one empty and at least one filled cell. For our purposes, we impose the additional requirement that all puzzles have exactly one solution.
- **House.** The column, row and block are collectively called the House.
- **Peer.** If two cells are in the same house (same row, same column, or same block) they are said to see each other, or to be peers.
- **Candidate.** Any digit that may be placed in an empty cell based on current state of the puzzle. If a digit is present in one or more of a cell's buddies, it cannot be a candidate for that cell. Analysis may further reduce the candidate set to a signle candidate, that candidate must be the sulution for that cell.
- **Analysis.** Any technique that eliminates candidates. Techniques of analysis do ultimately lead to solutions for cells, but it may take the application of multiple techniques or multiple applications of the same technique to reach a solution for a single cell. The point of analysis then, is to eliminate candidates, not look for solutions. Looking for solutions is scanning.

1.5 Common Solving Strategy

1.5.1 Full House/Last Digit

A Full House is simply the last digit that can be placed in a row,column or block. If it is the last digit for the whole grid, it is sometimes called "Last Digit".

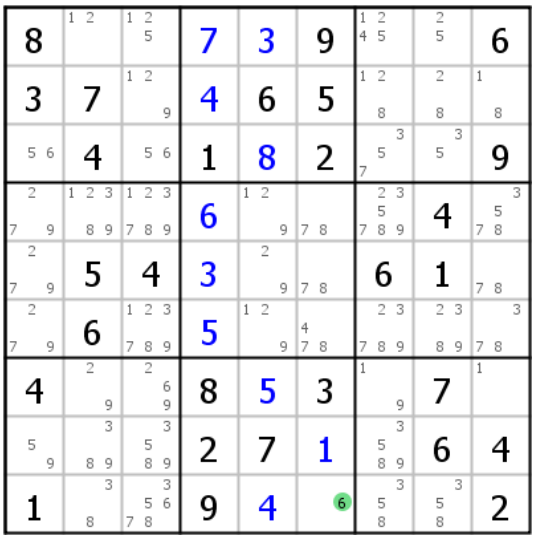
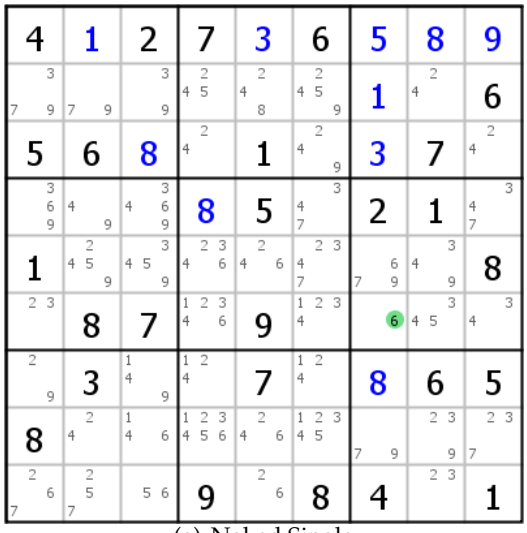


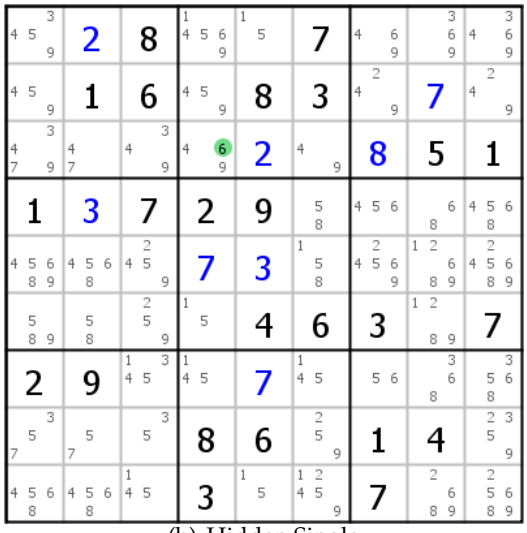
Figure 2: an example of full house

1.5.2 Naked Single

Naked Single means that in a specific cell only one digit remains possible (the last remaining candidate has no other candidates to hide behind and is thus naked). The digit must then go into that cell.



(a) Naked Single



(b) Hidden Single

Figure 3: Single Strategy

1.5.3 Hidden Single

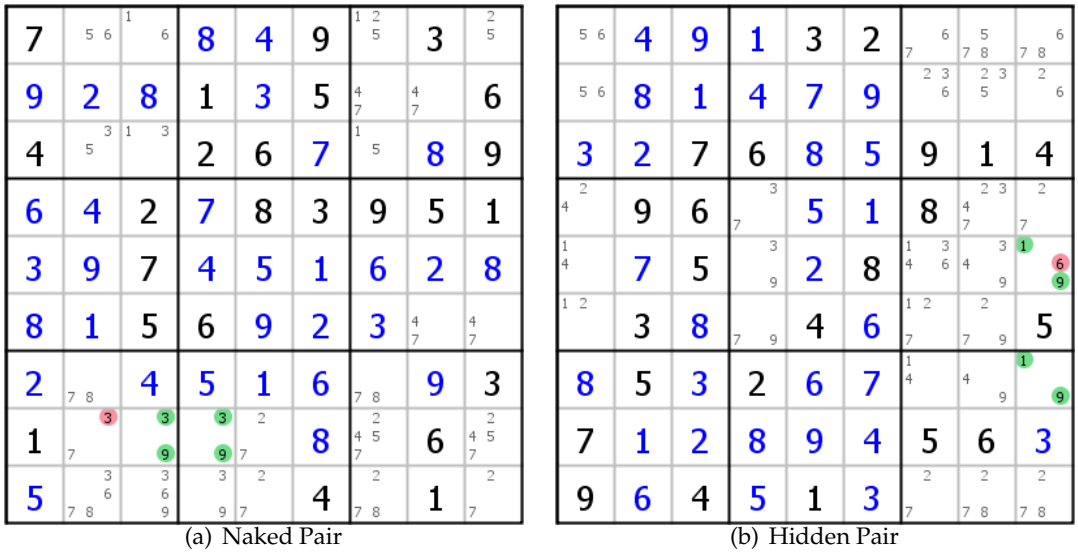
Hidden Single means that for a given digit and house only one cell is left to place that digit. The cell itself has more than one candidate left, the correct digit is thus hidden amongst the rest.

1.5.4 Naked Pair

If you can find two cells, both in the same house, that have only the same two candidates left, you can eliminate that two candidates from all other cells in that house.

1.5.5 Hidden Pair

All Hidden Subsets work the same way, the only thing that changes is the number of cells and candidates affected by the move. Take Hidden Pair: If you can find two cells within a house such as that two candidates appear nowhere outside those cells in that house, those two candidates must be placed in the two cells. All other candidates can therefore be eliminated.



(a) Naked Pair (b) Hidden Pair

Figure 4: Pair Strategy

1.5.6 Hidden Triple

Hidden Triples work in the same way as Hidden Pairs only with three cells and three candidates.

All the strategies above will be used as the benchmark to test the difficulty of new-created sudoku puzzles.

2 Analysis of the Problem

2.1 Background Knowledge

2.1.1 Exact Cover

In mathematics, given a collection $\mathcal{S}$ of subsets of a set $\mathcal{X}$, an **exact vocer** is a subcollection $\mathcal{S}^*$ of $\mathcal{S}$ that satisfies two conditions:

- The intersection of any two sistinct subsets in $\mathcal{S}^*$ is empty, i.e., the subsets in $\mathcal{S}^*$ are pairwise disjoint. In other words, each element in $\mathcal{X}$ is contained in at most one subset in $\mathcal{S}^*$.
- The union of the subsets in $\mathcal{S}^*$ is $\mathcal{X}$, i.e., the subsets in $\mathcal{S}^*$ cover $\mathcal{X}$. In other words, each element in $\mathcal{X}$ is contained in at least one subset in $\mathcal{S}^*$.

In short, an exact cover is "exact" in the sense that each element in $\mathcal{X}$ is contained in exactly one subset in $\mathcal{S}^*$. Figure 5 is an example of exact cover. As we can see the set(row 1,4 and 5) is the exact cover of the origin matrix.

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

Figure 5: an example of exact cover

2.1.2 Algorithm X

To solve the exact cover problem, we need to introduce the methodology called Algorithm X. "Algorithm X" is the name D.E Knuth used in his paper "Dancing Links" to refer to "the most obvious-trail-and-error approach" for finding all solutions to the exact cover problem. Technically, Algorithm X is a recursive, nondeterministic, depth-first, backtracking algorithm. While Algorithm X is generally useful as a succinct explanation of how the exact cover problem may be solved, Knuth's intent in presenting it was merely to demonstrate the utility of the dancing links technique via an efficient implementation he called DLX.

The exact cover problem is represented in Algorithm X using a matrix A consisting of 0s and 1s. The goal is to select a subset of the rows so that the digit 1 appears in each column exactly once. Algorithm X functions as follows:

Algorithm X

1. If the matrix A has no columns, the current partial solution is a valid solution; terminate successfully.

2. Otherwise choose a column *c* (deterministically).
3. Choose a row *r* such that $A_{r,c} = 1$ (nondeterministically).
4. For each column *j* such that $A_{r,j} = 1$,
   for each row *i* such that $A_{i,j} = 1$,
   delete row *i* from matrix *A*,
   delete column *j* from matrix *A*.
5. Repeat this algorithm recursively on the reduced matrix *A*.

For better understanding the procedure of Algorithm X, let's see the following example. Consider the exact problem specified by the universe U = {1, 2, 3, 4, 5, 6, 7} and the collection of sets $\mathcal{S}$ = {*A, B, C, D, E, F*}, where:

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

This problem is presented by the matrix:

	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

Table 1: Origin Matrix

For the least recursive time, we often select the column with the least 1s to start. Let's go on and solve this problem.

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):

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

	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

After several steps, we can get the matrix:

	2	7
F	1	1

Notice that the next step we can clear all the rest 1s in the matrix. So it is a correct answer to this exact cover problem. 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

2.2 Model Building

Now we have known how to solve the exact cover problem with Algorithm X, so what's the relationship between Sudoku puzzle and exact cover problem? In fact, generating and solving Sudoku are both exact cover problems. But apparently, Algorithm X are used to solve the 0-1 problem which is much different from the Sudoku grid. So the next thing we need to do is to convert Sudoku problem into the exact cover problem. Let me introduce some indices we might use later.

Symbols	Meanings
$\mathcal{X}$	The set of candidates
$\mathcal{Y}$	The set of constrains sets
$\mathcal{R}$	Binary relation matrix with 324 columns and 729 rows at most,representing the relationship "is contained in".
$c_{i,j}$	The element in R matrix, located in row i and column j.

Table 2: Symbol Table

Intuitively, we need to find a transformation *T*, which could achieve:

$$T_{\mathcal{Y}}(X) = R$$

Each possible assignment of a particular number to a particular cell is a candidate. When Sudoku is played with pencil and paper, candidates are often called penceil marks.

In the standard 9×9 Sudoku variant, in which each of 9×9 cells is assigned one of 9 numbers, there are $9 \times 9 \times 9 = 729$ possibilities. The key to convert Sudoku into exact cover problem is to find out the constrains sets $\mathcal{Y}$. According to the rules of Sudoku, we can get following constains:

- Each cell **must** and **only** have one digit.
- No duplicate number in the same row.
- No duplicate number in the same column.
- No duplicate number in the same block.

All the constaints are listed above, let's build the matrix R step by step. As we all Sudoku grid has 81 cells and a completed Sudoku has one digit in each of its cell. So in order to represent this constraint, we need 81 rows in our matrix $\mathcal{R}$. The elements in each row are not digit(1-9), but a abstract bool value(0 or 1). 1 means this cell has a number, otherwise it should be 0. Here we assign the number to the cell from left to right, starting from the top row. So if $c_{i,j}$ is not null, the $(9 \times i + j)th$ row should be 1. We call these 81 lines combined as "cell field" Apparently, a correct solution of Sudoku should be the exact cover of cell field.

We need no duplicate numbers in the same row, which means each number can only be placed once in a single row. There are 9 digits and 9 rows in a grid, so we also need 81 rows to express this constraint, called "row field". The column 1 to column 9 of the row field shows that the Sudoku has digit 1 to 9 in the first row, column 10 to 18 are correspond to the second row and so on. If we have placed digit 8 in the seventh row, then the $6 \times 9 + 8 = 62nd$ column should be marked as 1. A corrent solution could also be the exact cover of row field.

Just like "row field", "column field" are quite the same,which represents whether one digit has shown up in one column.If you have digit 8 in the 7th column, then the $6 \times 9 + 8 = 62nd$ column should be marked as 1.

Let's come to the last constraints. There are 9 digits and 9 blocks in a Sudoku, which means we need another 81 columns to express the relationship, called "block field". The index are count from the left-top to the right-bottom, rows first.

Now we can join all the four fields above and get the matrix with 324 columns. A Sudoku problem should have atmost 729 rows(for all possibilities). This matrix is the matrix $\mathcal{R}$ and the job to solve the Sudoku puzzle now has changed to find a exact cover of this matrix.

3 Matrix Design

To generate Sudoku puzzles of varying difficulty in a effcient way, we decide to split it into two parts.The first part is to generate a complete Sudoku grid filled with 9×9 figures. And the second part is to dig a few figures from the complete Sudoku grid. Of course, we need to gurantee that the Sudoku has a unique solution.

3.1 Generate complete Sudoku grids

There is a fact that the number of different complete Sudoku grids was computed to be $6670903752021072936960 \approx 6.671 \times 10^{21}$. And the essentially different Sudoku grids, which can not be transformed by another Sudoku grid in some symmetrical ways, is proved to be 2297902829591040 by Ed Russell and Frazer Jarvis in 2006. That's to say, the possiblity of building a random Sudoku grid is high enough for computers. Based on this, we adopt a Las Vegas-liked algorithm to generate them.

First of all, we create a random array filled with $1 \sim 9$ and put it in the first line of a empty Sudoku grid. Then we continue to create a array, take the figures out of it in turns to fill in cells of the follwing line, and check whether the constraints are met. If not, we put the figures back and check next figure until the line is completely filled. If this process fails, a new random array is needed to be created. Then repeat the procedure.

Here is the pseudocode:

Generate Complete Grids Algorithm

1. Define a array A which is filled with $1 \sim 9$ in turns.
2. Choose a random figure in this array and swap it with A[0]. Repeat the procedure for a few (in most cases, the number is 20) times. Then the array A becomes a random one.
3. For each line *i* in a empty Sudoku *S*,
   if it's the first line($i = 0$),
   put the array in without any check,
   else if $i \neq 0$,
   randomize the array A with Step 2,
   for each figure *n* in A,
   fill the cells in S_i with *n* that can be put in,
   if S_i can't be filled up,
   randomize array A,
   if the randomization times *t* reach a threshold *MaxTimes*,
   remove all figures in *S* and restart(procedure for *i* in *S*)
   else,
   remove all the figures in S_i and repeat procedure($i \neq 0$) ,
4. return the complete Sudoku *S*.

Obviously, by trying for enough times, we will get a complete Sudoku grid which meets all the constraints. In fact, we can generate such grids in several milliseconds, in other words, 100000 grids in one minute. By the way, digging different figures in different orders from a complete Sudoku grid will generate a large amount of Sudoku puzzles. Therefore, this Las Vegas-liked algorithm is quite efficient and our require to construct various Sudoku puzzles can be met quite well.

3.2 Generate a Sudoku puzzle

3.2.1 Solve Sudoku puzzle with DLX algorithm

Since the Sudoku puzzles generated should be solved with a unique solution, we must develop a algorithm to confirm that the Sudoku puzzle is constraint-satisfied while taking figures out of it. Fortunately, we have turned this into a exact cover problem. So, let's focus on solving it.

An elegant algorithm to solve the exact cover problem is DLX, which is one good way to implement algorithm X using a data structure called **Dancing Links**. To transform the matrix $\mathcal{R}$ into a dancing links, we represent each 1 in the matrix $\mathcal{R}$ as a *data object x* with five fields $x.left, x.right, x.down$, $x.up, x.columnHead$. Rows of the matrix are doubly linked as circular lists via $x.left$ and $x.right$ fields; columns are doubly linked as circular lists via $x.up$ and $x.down$ fields. Each column list also includes a special data object called its *list header*.

The list headers are instantiations of a larger object called a *column object*. Each column object y contains the fields $y.left, y.right, y.up, y.down$, and $y.columnHead$ of a data object and two additional fields, $y.size$ and $y.name$; the size is the number of 1s in the column, and the name is a symbolic identifier for showing the answer. The $columnHead$ field of each object points to the column object at the head of the relevant column.

The $left$ and $right$ fields of the list headers link together all columns that still need to be covered. This circular list also includes a special column object called the *root*, h, which serves as a master header for all the active headers. The fields $up, down, columnHead, size$ and $name$ are not used.

To implement the above-mentioned data structures, we use Java and first simply create a class called SudokuNode representing the data object then create a class called ColumnHead representing the column object by extending the SudokuNode, that is to say that the ColumnHead is a subclass of the SudokuNode.

The algorithm X can now be cast in the following explicit, deterministic form as a recursive procedure $search(k)$, which is invoked initially with $k = 0$. $Array$ is a linear list used to keep current solution.

$Search(k)$
If $h.right == h$, save the current solution($Save\_Current\_Solution()$) and return.
choose a column object c($Choose\_Column\_Object()$).
$Cover\_Column(c)$;
$r = c$;
For each $r = r.down$, while $r \neq c$,
    set $Array[k] = r$;
    $j = r$;
    for each $j = j.right$, while $j \neq r$,
        $Cover\_Column(j.columnHead)$;
    $search(k + 1)$;
    set $r = Array[k]$ and $c = r.columnHead$;
    $j = r$;

for each $j = j.left$, while $j \neq r$,
$Uncover\_Column(j.columnHead)$
$Uncover\_Column(c)$ and return.

Function $Save\_Current\_Solution()$ simply clone the $Array$ and save the copy into a two dimensional table:

$Save\_Current\_Solution()$
$Table.add(Array.clone)$;

To minimize the branching factor, we want to find out a column object c with the smallest number of 1s, in other words, with the smallest field $size$, we could set $s = \infty$ and then

$Choose\_Column\_Object$
$r = h$;
for each $r = r.right$, while $r \neq h$
if $r.size < s$
set $c = r$ and $s = r.size$;

Function $Cover_Column(c)$ removes c from the header list and removes all rows in c's own list from the other column lists they are in, that is to say, removes all data objects linked with a data object in column c in a row from the column lists they are in.

$Cover\_Column(c)$
Set $c.right.left = c.left$ and $c.left.right = c.right$;
$i = c.down$;
For each $i = i.right$, while $i \neq c$,
$j = i$;
for each $j = j.right$, while $j \neq i$,
set $j.down.up = j.up$ and $j.up.down = j.down$;
set $j.columnHead.size = j.columHead.size - 1$

The Function $Uncover_Column(c)$ add the data objects removed by $Cover_Column(c)$ into the original column lists they are in.

$Uncover\_Column(c)$
$i = c$;
For each $i = i.up$, while $i \neq c$,
    $j = i$;
        for each $j = j.left$, while $j \neq i$,
        set $j.columnHead.size = j.columnHead.size + 1$;
        set $j.down.up = j$ and $j.up.down = j$;
$c.right.left = c$ and $c.left.right = c$;

3.2.2 Generate a genuine Sudoku puzzle

To generate a Sudoku puzzle with an unique solution, we implement a Las Vegas algorithm based on the above-mentioned DLX algorithm. This generate algorithm first generate a complete Sudoku puzzle and then try to randomly remove numbers in the Sudoku puzzle. Each time a number is removed, use DLX algorithm to solve the puzzle, if there is only one solution, try to remove another number, if we get several solutions, add the number to its original position. If the algorithm fails to remove number for a limited time, it backtracks. We implement the backtracking using data structure stack. We use n to represent the asked number of vacancies. If we successfully remove n numbers in the Sudoku puzzle in limited times of loops, then return true, otherwise return false. We use data structure $Array$ to store the position of removed numbers and their values. $count$ is a global variable to record the current number of vacancies. s is used to represent a Sudoku puzzle. The algorithm can be cast in the following form:

$get(n)$
create a two-dimensional boolean array $sign$ to record whether the numbers in the Sudoku puzzle is removed or not.
set $times = 0$ and $count = 0$
$tryToRemoveACell(sign)$;
while $times < TIMES$,
    if $count == n$ return $true$;
    for $i = 0$ to $i = LIMIT$,
        if $tryToRemoveACell(sign) == true$ break;
    if $i == LIMIT$,
        $Array array = stack.pop()$;
        $sign[array.x][array.y] = true$;
        $s[array.x][array.y] = array.value$;
    $times = times + 1$;
return $false$;

Function $tryToRemoveACell(sign)$ is used to randomly remove a number from the

Sudoku puzzle.

$tryToRemoveACell(sign)$
randomly choose a position in two-dimensional array $sign$ which is not a vacancy.
$sign[position.x][position.y] = false$;
$stcak.push(newArray(position.x, position.y, s[position.x][position.y]))$;
$s[position.x][s.position.y] = 0$;
$count = count + 1$;
solve the s using DLX algorithm.
if get more then one solution,
    $Arrayarray = stack.pop()$;
    $s[array.x][array.y] = array.value$;
    $sign[array.x][array.y] = true$;
    $count = count - 1$;
    return $false$;
return $true$;

Finally, we can wrap the above-mentioned algorithm to certainly generate a Sudoku puzzle each time.

$Get(n)$
while $get(n) == false$ do nothing.

4 The Model Results

5 Evaluate of the Model

5.1 Advantage of the Model

5.2 Disadvantage of the Model

6 Validating the Model

7 Conclusions

in short but accurate

8 A Summary

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

lipsum latex

9 Strengths and weaknesses

Etiam euismod. Fusce facilisis lacinia dui. Suspendisse potenti. In mi erat, cursus id, nonummy sed, ullamcorper eget, sapien. Praesent pretium, magna in eleifend egestas, pede pede pretium lorem, quis consectetuer tortor sapien facilisis magna. Mauris quis magna varius nulla scelerisque imperdiet. Aliquam non quam. Aliquam porttitor quam a lacus. Praesent vel arcu ut tortor cursus volutpat. In vitae pede quis diam bibendum placerat. Fusce elementum convallis neque. Sed dolor orci, scelerisque ac, dapibus nec, ultricies ut, mi. Duis nec dui quis leo sagittis commodo.

9.1 Strengths

- **Applies widely**
  This system can be used for many types of airplanes, and it also solves the interference during the procedure of the boarding airplane,as described above we can get to the optimization boarding time.We also know that all the service is automate.
- **Improve the quality of the airport service**
  Balancing the cost of the cost and the benefit, it will bring in more convenient for airport and passengers.It also saves many human resources for the airline.
-

References

[1] Donald E. Knuth: Dancing Links, Oxford-Microsoft Symposium on Computer Science, 2000

[2] Wikipedia: Sudoku

[3] `http://blog.gssxgss.me/use-dlx-to-solve-sudoku-1/`

Appendices

Appendix A First appendix

Here are simulation programmes we used in our model as follow.

Listing 1 : Implement of Sudoku Creation

```
package Sudoku;

public class Array {
    public int x;
    public int y;
    public int value;

    public Array(int x, int y, int value){
        this.x = x;
        this.y = y;
        this.value = value;
    }
}
```

```
package Sudoku;

public class ColumnHead extends SudokuNode{
    public int sum;
    public int name;

    public ColumnHead(SudokuNode upNode, SudokuNode downNode, SudokuNode leftNode, SudokuNode r
        super(upNode, downNode, leftNode, rightNode, columnHead);
        this.name = name;
    }
}
```

```
package Sudoku;

import java.util.Random;

class GenerateByLasVegas {
    public int GridNumber = 3;
    public int Number = 9;
    public int MaxCompareNum = 500000;
    public int Sudoku[][] = new int[Number][Number];
    public CandidateNumberOriginal Candinum = new CandidateNumberOriginal(Number);

    // public static void main(String args[]){
        // int times = 0;
        // while(times < 100){
            // if(Generate() == true){
                // times++;
            // }
        // }

    // }

    public int[][] Generate(){
```

```
        //Initialize
        for(int i = 0; i < Number; i++){
            for(int j = 0; j < Number; j++){
                Sudoku[i][j] = 0;
            }
        }
        //Generate
        int WrongTimes = 0;
        for(int i = 0; i < Number; i++){
            if(i == 0){
                Candinum.ChangeOrder();
                Sudoku[i] = Candinum.Candidate.clone();
            } else {
                do{
                    Candinum.ChangeOrder();
                    WrongTimes++;
                    if(WrongTimes > MaxCompareNum){
                        return null;
                    }
                }while(CanFillInArray(Candinum.Candidate, Number, i) == false);
                Sudoku[i] = Candinum.Candidate.clone();
            }
        }

        //Print
        /*for(int i = 0; i < Number; i++){
            for(int j = 0; j < Number; j++){
                System.out.print(Sudoku[i][j] + " ");
            }
            System.out.println();
        }*/
        //System.out.println("------------------");
        int result[][] = new int[Number+1][Number+1];
        for(int i = 1; i <= Number; i++){
            for(int j = 1; j <= Number; j++){
                result[i][j] = Sudoku[i - 1][j - 1];
            }
        }
        for(int i = 1; i <= Number; i++){
            for(int j= 1; j <= Number; j++){
                System.out.print(result[i][j] + " ");
            }
            System.out.println();
        }
        System.out.println("------------------");
        return result;
    }

    public boolean CanFillIn(int num, int line, int column){
        //ï£¡ï£¡ï£¡ï£¡ï£¡ï£¡ï£¡ÍÍï£¡ï£¡ï£¡ï£¡ï£¡ÙśÈ¡Ït'ï£¡ï£¡ï£¡ï£¡ï£¡ï£¡Åžï£¡
//      for(int i=0;i<Number;i++){
//          if(Sudoku[i][column]==num || Sudoku[line][i]==num){
//              return false;
//          }
//      }
        for(int i = 0; i < line; i++){
            if(Sudoku[i][column] == num){
                return false;
            }
        }
        for(int i = 0; i < column; i++){
```

```
            if(Sudoku[line][i] == num){
                return false;
            }
        }
        int linetmp = line - line % GridNumber;
        int columntmp = column - column % GridNumber;

        for(int i = 0; i <= line % GridNumber; i++){
            for(int j = 0; j < GridNumber; j++){
                if(Sudoku[linetmp + i][columntmp + j] == num){
                    return false;
                }
            }
        }
        return true;
    }

    public boolean CanFillInArray(int a[],int top,int line){
        for(int i = 0; i < top; i++){
            if(CanFillIn(a[i], line, i) == false)
                return false;
        }
        return true;
    }

}

class CandidateNumberOriginal{
    int Candidate[] = null;
    int num = 0;
    public CandidateNumberOriginal(int Number){
        Candidate = new int[Number];
        for(int i = 0; i < Number; i++){
            Candidate[i] = i+1;
        }
        num = Number;
    }

    public boolean ChangeOrder(){
        int numtmp = 0;
        boolean Used[] = new boolean[num];
        int canditmp[] = Candidate.clone();
        Random random = new Random();
        for(int i = 0; i < num; i++){
            Used[i] = false;
        }
        for(int i = 0; i < num; i++){
            do{
                numtmp = random.nextInt(num);
            }while(Used[numtmp] == true);
            Candidate[i] = canditmp[numtmp];
            Used[numtmp] = true;
        }
        return true;
    }
}
```

```
package Sudoku;

import java.util.Random;
```

```
class GenerateFullSudoku {
    public int GridNumber = 3;
    public int Number = 9;
    public int CycleTimes = 20;
    public int MaxChangesTimes = 220;
    public int Sudoku[][] = new int[Number][Number];
    public CandidateNumber Candinum = new CandidateNumber(Number,CycleTimes);

    public void Initialize(){
        for(int i=0;i<Number;i++){
            for(int j=0;j<Number;j++){
                Sudoku[i][j] = 0;
            }
        }
    }

    public int[][] Generate(){
        int ChangeTimes = 0;

        //Initialize
        Initialize();

        //Generate
        for(int i=0;i<Number;i++){
            if(i==0){
                Candinum.ChangeOrder();
                Sudoku[i] = Candinum.Candidate.clone();
            } else {
                Candinum.ChangeOrder();
                ChangeTimes++;
                if(ChangeTimes > MaxChangesTimes){
                    i = -1; //Restart the generate process
                    ChangeTimes = 0;
                    Initialize();
                } else {
                    if(CanFillInArray(Candinum.Candidate, Candinum.Used, i) == false){
                        for(int k=0;k<Number;k++){
                            Sudoku[i][k] = 0;
                        }
                        i = i - 1; // Calculate again
                    }
                }
            }
        }

        //Print
        /*for(int i=0;i<Number;i++){
            for(int j=0;j<Number;j++){
                System.out.print(Sudoku[i][j]+" ");
            }
            System.out.println();
        }
        System.out.println("-----------------");*/
        int[][] result = new int[10][10];
        for(int i = 1; i <= Number; i++){
            for(int j = 1; j <= Number; j++){
                result[i][j] = Sudoku[i - 1][j - 1];
            }
        }
        return result;
```

```
    }

    public boolean CanFillIn(int num, int line, int column){
        for(int i=0;i<line;i++){
            if(Sudoku[i][column] == num){
                return false;
            }
        }
        for(int i=0;i<column;i++){
            if(Sudoku[line][i] == num){
                return false;
            }
        }
        int linetmp = line - line % GridNumber;
        int columntmp = column - column % GridNumber;

        for(int i=0;i<=line%GridNumber;i++){
            for(int j=0;j<GridNumber;j++){
                if(Sudoku[linetmp+i][columntmp+j] == num){
                    return false;
                }
            }
        }
        return true;
    }

    public boolean CanFillInArray(int array[],boolean Used[],int line){
        boolean Flag = false;
        for(int i=0;i<Number;i++){
            Flag = false;
            for(int j=0;j<array.length;j++){
                if(Used[j] == false && CanFillIn(array[j], line, i)){
                    Sudoku[line][i] = array[j];
                    Used[j] = true;
                    Flag = true;
                    break;
                }
            }
            if(Flag == false){
                return false;
            }
        }
        return true;
    }
}

class CandidateNumber{
    public int Candidate[] = null;
    public int CycleTimes = 15;
    public boolean Used[] = null;
    public CandidateNumber(int Number,int CycleTimes){
        Candidate = new int[Number];
        Used = new boolean[Number];
        for(int i=0;i<Number;i++){
            Candidate[i] = i+1;
            Used[i] = false;
        }
        this.CycleTimes = CycleTimes;
    }

    public void ChangeOrder(){
```

```
        int tmp = 0;
        int numtmp = 0;
        Random random = new Random();
        for(int i=0;i<CycleTimes;i++){
            tmp = random.nextInt(9);
            numtmp = Candidate[tmp];
            Candidate[tmp] = Candidate[0];
            Candidate[0] = numtmp;
        }
        for(int i=0;i<Used.length;i++){
            Used[i] = false;
        }
    }
}
```

```
package Sudoku;

import java.util.Random;
import java.util.Stack;

public class GetPuzzle {
    public static final int NUMBER = 40;
    public static final int LIMIT = 100;
    public static final int TIMES = 1000;
    public int[][] input;
    public Sudoku sudoku;
    public Random random;
    public int count;
    public Stack<Array> stack;
    public Resolve resolve;
    public int times;

    public GetPuzzle(int[][] input){
        //sudoku = new Sudoku(input);
        this.input = input;
        random = new Random();
        count = 0;
        stack = new Stack<Array>();
        /*for(int i = 1; i <= 9; i++){
            for(int j = 1; j <= 9; j++){
                System.out.print(input[i][j] + " ");
            }
            System.out.println();
        }*/
    }

    public void get(){
        boolean[][] sign = new boolean[10][10];
        for(int i = 1; i <= 9; i++){
            for(int j = 1; j <= 9; j++){
                sign[i][j] = true;
            }
        }
        //stack.push(new Array(0, 0, 0));
        while(true){
            if(tryToRemoveACell(sign)){
                break;
            }
        }
        while(count < NUMBER){
```

```
            /*int x = 0;
            int y = 0;
            do{
                x = random.nextInt(9) + 1;
                y = random.nextInt(9) + 1;
            }while(!sign[x][y]);
            stack.push(new Array(x, y, input[x][y]));
            input[x][y] = 0;
            sudoku = new Sudoku(input);
            resolve = new Resolve(sudoku);
            resolve.resolve(0);
            count = count + 1;
            //ArrayList<LinkedList<SudokuNode>> list = resolve.allAnswers;
            if(resolve.allAnswers.size() > 1){
                Array array = stack.pop();
                input[array.x][array.y] = array.value;
                count = count - 1;
            }*/
            int i = 0 ;
            for(; i < LIMIT; i++){
                //System.out.println("!!!!!!!!!!!");
                if(tryToRemoveACell(sign)){
                    break;
                }
            }
            if(i == LIMIT){
                Array array = stack.pop();
                //sign[array.x][array.y] = true;
                input[array.x][array.y] = array.value;
                count = count - 1;
            }
        }

        return;
    }

    public boolean get(int number){
        boolean[][] sign = new boolean[10][10];
        GenerateFullSudoku generateFullSudoku = new GenerateFullSudoku();
        input = generateFullSudoku.Generate();
        for(int i = 1; i <= 9; i++){
            for(int j = 1; j <= 9; j++){
                sign[i][j] = true;
            }
        }
        //stack.push(new Array(0, 0, 0));
        times = 0;
        count = 0;
        while(true){
            if(tryToRemoveACell(sign)){
                times = times + 1;
                break;
            }
        }
        while(times < TIMES){
            if(count == number){
                return true;
            }
            int i = 0 ;
            for(; i < LIMIT; i++){
                //System.out.println("!!!!!!!!!!!");
```

```
                if(tryToRemoveACell(sign)){
                    break;
                }
            }
            if(i == LIMIT){
                Array array = stack.pop();
                sign[array.x][array.y] = true;
                input[array.x][array.y] = array.value;
                count = count - 1;
            }
            times = times + 1;
            //System.out.println(times);
        }

        return false;
    }

    public boolean tryToGet(){
        boolean[][] sign = new boolean[10][10];
        for(int i = 1; i <= 9; i++){
            for(int j = 1; j <= 9; j++){
                sign[i][j] = true;
            }
        }
        //stack.push(new Array(0, 0, 0));
        times = 0;
        count = 0;
        while(true){
            if(tryToRemoveACell(sign)){
                times = times + 1;
                break;
            }
        }
        while(times < TIMES){
            int i = 0 ;
            for(; i < LIMIT; i++){
                //System.out.println("!!!!!!!!!!!!");
                if(tryToRemoveACell(sign)){
                    break;
                }
            }
            if(i == LIMIT){
                Array array = stack.pop();
                sign[array.x][array.y] = true;
                input[array.x][array.y] = array.value;
                count = count - 1;
            }
            times = times + 1;
            if(count == NUMBER){
                return true;
            }
        }

        return false;
    }

    public void getHard(){
        while(!tryToGet()){}
    }

    public void getHard(int number){
```

```
        while(!get(number)){}
    }

    public boolean tryToRemoveACell(boolean[][] sign){
        int x = 0;
        int y = 0;
        do{
            x = random.nextInt(9) + 1;
            y = random.nextInt(9) + 1;
        }while(!sign[x][y]);
        sign[x][y] = false;
        /*System.out.print(x + " " + y + " ");
        System.out.println(input[x][y]);*/
        stack.push(new Array(x, y, input[x][y]));
        input[x][y] = 0;
        sudoku = new Sudoku(input);
        resolve = new Resolve(sudoku);
        resolve.resolve(0);
        count = count + 1;
        //ArrayList<LinkedList<SudokuNode>> list = resolve.allAnswers;
        if(resolve.allAnswers.size() > 1){
            Array array = stack.pop();
            input[array.x][array.y] = array.value;
            sign[array.x][array.y] = true;
            count = count - 1;
            return false;
        }
        return true;
    }
}
```

```
package Sudoku;

import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.LinkedList;

public class Main {
    public static void main(String[] args) throws IOException{
        int[][] s = new int[10][10];
        int x = 1;
        for(int i = 1; i <= 3; i++){
            for(int j = 1; j <= 3; j++){
                for(int k = 1; k <= 9; k++){
                    s[3 * (i - 1) + j][k] = x % 9 + 1;
                    x = x + 1;
                }
                x = x + 3;
            }
            x = x + 1;
        }

        s[1][1] = 0;
        s[1][2] = 0;
        s[9][9] = 0;
        s[6][7] = 0;
        s[2][1] = 0;
        s[2][2] = 0;
        s[1][3] = 0;
```

```
for(int i = 1; i <= 9; i++){
    s[1][i] = 0;
    s[2][i] = 0;
    s[3][i] = 0;
}

/*for(int i = 1; i <= 9; i++){
    for(int j = 1; j <= 9; j++){
        System.out.print(s[i][j] + " ");
    }
    System.out.println();
}*/

//GenerateByLasVegas generateByLasVegas = new GenerateByLasVegas();
/*while(true){
    s = generateByLasVegas.Generate();
    if(s != null){
        break;
    }
}*/

GenerateFullSudoku generateFullSudoku = new GenerateFullSudoku();
//s = generateFullSudoku.Generate();

for(int i = 0; i <=9; i++){
    s[1][i] = 0;
}

Sudoku sudoku = new Sudoku(s);
ColumnHead columnHead = sudoku.columnLists[1];
SudokuNode pHead = columnHead.downNode;
SudokuNode pNode = pHead.rightNode;
/*do{
    System.out.println(pNode.columnHead.name);
    pNode = pNode.rightNode;
}while(pNode != pHead.rightNode);*/

/*for(int i = 1; i <= Sudoku.MAX_COLUMN; i++){
    System.out.print(sudoku.columnLists[i].sum);
}*/

//System.out.println();

Resolve resolve = new Resolve(sudoku);
resolve.resolve(0);
ArrayList<LinkedList<SudokuNode>> answers = resolve.allAnswers;
/*for (LinkedList<SudokuNode> linkedList : answers) {
    for (SudokuNode sudokuNode : linkedList) {
        SudokuNode sNode = sudokuNode;
        do{
            System.out.print(sNode.columnHead.name + " ");
            sNode = sNode.rightNode;
        }while(sNode != sudokuNode);
        System.out.println();
    }
    System.out.println();
}*/

TranslateSolution translateSolution = new TranslateSolution(answers);
//translateSolution.translate();
```

```
//GetPuzzle getPuzzle = new GetPuzzle(generateFullSudoku.Generate());
GetPuzzle getPuzzle;
//getPuzzle.get();
/*for(int i = 1; i <= 9; i++){
    for(int j = 1; j <= 9; j++){
        System.out.print(getPuzzle.input[i][j] + " ");
    }
    System.out.println();
}*/
//System.out.println("-----------------");
//resolve = new Resolve(new Sudoku(getPuzzle.input));
//resolve.resolve(0);
//answers = resolve.allAnswers;
//translateSolution = new TranslateSolution(answers);
//translateSolution.translate();

/*for(int i = 0; i < 100; i++){
    getPuzzle = new GetPuzzle(generateFullSudoku.Generate());
    getPuzzle.get();
    for(int j = 1; j <= 9; j++){
        for(int k = 1; k <= 9; k++){
            System.out.print(getPuzzle.input[j][k] + " ");
        }
        System.out.println();
    }
    resolve = new Resolve(new Sudoku(getPuzzle.input));
    resolve.resolve(0);
    resolve.showRunTimes();
}*/

/*getPuzzle = new GetPuzzle(generateFullSudoku.Generate());
getPuzzle.getHard();
for(int j = 1; j <= 9; j++){
    for(int k = 1; k <= 9; k++){
        System.out.print(getPuzzle.input[j][k] + " ");
    }
    System.out.println();
}
System.out.println();
resolve = new Resolve(new Sudoku(getPuzzle.input));
resolve.resolve(0);
resolve.showRunTimes();*/

//System.out.println("\n\n\n\n\n\n\n\n");

ProblemReader problemReader = new ProblemReader("SudokuProblem.csv");
/*for(int k = 0; k < 10; k++){
    s = problemReader.readAndGet();
    for(int i = 1; i <=9; i++){
        for(int j = 1; j <= 9; j++){
            System.out.print(s[i][j] + " ");
        }
        System.out.println("");
    }
    System.out.println("@@@@@@@@@@@@@@@@@");
    resolve = new Resolve(new Sudoku(s));
    resolve.resolve(0);
    translateSolution = new TranslateSolution(resolve.allAnswers);
    translateSolution.translate();
    resolve.showRunTimes();
    //System.out.println("$$$$$$$$$$$$$$$$$$$$$");
```

```
        }*/

        FileWriter fileWriter = new FileWriter("data.txt");

        NormalReader normalReader = new NormalReader("10_5sudoku_plain.txt");
        /*for(int k = 0; k < 100000; k++){
            s = normalReader.getProblem();
            //System.out.println(normalReader.nonzero);
            resolve = new Resolve(new Sudoku(s));
            resolve.resolve(0);
            //resolve.showRunTimes();
            fileWriter.write(normalReader.nonzero + " " + resolve.runTimes + " " + resolve.loop
            fileWriter.flush();
        }*/

        fileWriter.close();

        for(int n = 56; n <= 60; n++){
            System.out.println(n);
            fileWriter = new FileWriter("data" + n + ".txt");
            for(int i = 0; i < 1; i++){
                getPuzzle = new GetPuzzle(s);
                getPuzzle.getHard(n);
                resolve = new Resolve(new Sudoku(getPuzzle.input));
                for(int j = 1; j <= 9; j++){
                    for(int k = 1; k <= 9; k++){
                        System.out.print(getPuzzle.input[j][k] + " ");
                    }
                    System.out.println();
                }
                resolve.resolve(0);
                if(resolve.allAnswers.size() > 1){
                    System.out.println("!!!!!!!!!!!!!!!!!!!!!!!!!");
                    System.exit(0);
                }
                if(resolve.allAnswers.size() == 1){
                    System.out.println("@@@@@@@@@@@@@@@@@@@@@@@@@");
                }
                translateSolution = new TranslateSolution(resolve.allAnswers);
                translateSolution.translate();
                fileWriter.write(n + " " + resolve.runTimes + " " + resolve.loopTimes + "\n");
                fileWriter.flush();
            }
            fileWriter.close();
        }
    }
}
```

```
package Sudoku;

import java.util.ArrayList;
import java.util.LinkedList;

public class Resolve {
    public Sudoku sudoku;
    public LinkedList<SudokuNode> aList;
    public ArrayList<LinkedList<SudokuNode>> allAnswers;
    public ColumnHead head;
    public ColumnHead[] columnHeads;
```

```
public long runTimes;
public long loopTimes;
//public long branchLoopTimes;

public Resolve(Sudoku sudoku){
    this.sudoku = sudoku;
    aList = new LinkedList<SudokuNode>();
    allAnswers = new ArrayList<LinkedList<SudokuNode>>();
    this.head = sudoku.head;
    this.columnHeads = sudoku.columnLists;
    runTimes = 0;
    loopTimes = 0;
}

public void resolve(int k){
    if(head.rightNode == head){
        allAnswers.add(new LinkedList<SudokuNode>(aList));
        return;
    }
    runTimes = runTimes + 1;
    int s = 1000;
    int n = 0;
    SudokuNode h = head.rightNode;
    for(;h != head; h = h.rightNode){
        if(((ColumnHead)h).sum < s){
            s = ((ColumnHead)h).sum;
            n = ((ColumnHead)h).name;
        }
    }
    //System.out.println(n);
    h = columnHeads[n];
    coverColumn(h);
    SudokuNode rNode = h.downNode;
    for(; rNode != h; rNode = rNode.downNode){
        aList.add(k, rNode);
        for(SudokuNode jNode = rNode.rightNode; jNode != rNode; jNode = jNode.rightNode){
            coverColumn(jNode.columnHead);
        }
        resolve(k + 1);
        rNode = aList.get(k);
        aList.remove(k);
        h = rNode.columnHead;
        for(SudokuNode jNode = rNode.leftNode; jNode != rNode; jNode = jNode.leftNode){
            uncoverColumn(jNode.columnHead);
        }
    }
    uncoverColumn(h);
    //System.out.println("branchLoopTime " + branchLoopTimes);
    return;
}

public void coverColumn(SudokuNode cNode){
    cNode.rightNode.leftNode = cNode.leftNode;
    cNode.leftNode.rightNode = cNode.rightNode;
    for(SudokuNode iNode = cNode.downNode; iNode != cNode; iNode = iNode.downNode){
        SudokuNode jNode = iNode.rightNode;
        for(; jNode != iNode; jNode = jNode.rightNode){
            jNode.downNode.upNode = jNode.upNode;
            jNode.upNode.downNode = jNode.downNode;
            jNode.columnHead.sum = jNode.columnHead.sum - 1;
            loopTimes = loopTimes + 1;
```

```
                //branchLoopTimes = branchLoopTimes + 1;
            }
        }
    }

    public void uncoverColumn(SudokuNode cNode){
        for(SudokuNode iNode = cNode.upNode; iNode != cNode; iNode = iNode.upNode){
            for(SudokuNode jNode = iNode.leftNode; jNode != iNode; jNode = jNode.leftNode){
                jNode.columnHead.sum = jNode.columnHead.sum + 1;
                jNode.downNode.upNode = jNode;
                jNode.upNode.downNode = jNode;
                loopTimes = loopTimes + 1;
                //branchLoopTimes = branchLoopTimes + 1;
            }
        }
        cNode.rightNode.leftNode = cNode;
        cNode.leftNode.rightNode = cNode;
    }

    public void showRunTimes(){
        System.out.println("searched " + runTimes + " branch(es) to get the answer");
        System.out.println("looped "+ loopTimes + " times to get the answer");
        System.out.println("*****************\n");
    }
}
```

```
package Sudoku;

import java.util.ArrayList;

public class Sudoku {
    public static final int MAX_COLUMN = 324;
    public static final int MAX_ROW = 729;
    public static final int SIZE = 81;
    public ColumnHead head;
    public ColumnHead[] columnLists = new ColumnHead[MAX_COLUMN + 1];
    public int[][] input;

    public Sudoku(int[][] input){
        this.input = input;
        buildHead();
        resolveInput();
    }

    public void buildHead(){
        head = new ColumnHead(null, null, null, null, null, 0);
        columnLists[0] = head;
        for(int i = 1; i <= MAX_COLUMN; i++){
            columnLists[i] = new ColumnHead(null, null, null, null, null, i);
        }
        for(int i = 1; i < MAX_COLUMN; i++){
            columnLists[i].rightNode = columnLists[i + 1];
            columnLists[i].leftNode = columnLists[i - 1];
        }
        head.rightNode = columnLists[1];
        head.leftNode = columnLists[MAX_COLUMN];
        columnLists[MAX_COLUMN].rightNode = head;
        columnLists[MAX_COLUMN].leftNode = columnLists[MAX_COLUMN - 1];
        head.sum = 1000;
    }
```

```
public void resolveInput(){
    SudokuNode[] p = new SudokuNode[MAX_COLUMN + 1];
    for(int i = 1; i <= MAX_COLUMN; i++){
        p[i] = columnLists[i];
    }
    for(int i = 1; i <= 9; i++){
        for(int j = 1; j <= 9; j++){
            if(input[i][j] != 0){
                int l = countLocation(i, j);
                int r = countRow(i, input[i][j]) + SIZE;
                int c = countColumn(j, input[i][j]) + 2 * SIZE;
                int la = countLattice(i, j, input[i][j]) + 3 * SIZE;
                p[l].downNode = new SudokuNode(p[l], null, null, null, columnLists[l]);
                p[r].downNode = new SudokuNode(p[r], null, null, null, columnLists[r]);
                p[c].downNode = new SudokuNode(p[c], null, null, null, columnLists[c]);
                p[la].downNode = new SudokuNode(p[la], null, null, null, columnLists[la]);
                p[l] = p[l].downNode;
                p[r] = p[r].downNode;
                p[c] = p[c].downNode;
                p[la] = p[la].downNode;
                p[l].downNode = columnLists[l];
                p[r].downNode = columnLists[r];
                p[c].downNode = columnLists[c];
                p[la].downNode = columnLists[la];
                p[l].rightNode = p[r]; p[r].leftNode = p[l];
                p[l].leftNode = p[la]; p[la].rightNode = p[l];
                p[r].rightNode = p[c]; p[c].leftNode = p[r];
                p[c].rightNode = p[la]; p[la].leftNode = p[c];
                columnLists[l].upNode = p[l];
                columnLists[r].upNode = p[r];
                columnLists[c].upNode = p[c];
                columnLists[la].upNode = p[la];
            }
            else {
                ArrayList<Integer> arrayList = possibleNumber(i, j);
                //System.out.println("!");
                for (Integer integer : arrayList) {
                    //System.out.println("!!");
                    int v = integer.intValue();
                    //System.out.println(v);
                    int l = countLocation(i, j);
                    int r = countRow(i, v) + SIZE;
                    int c = countColumn(j, v) + 2 * SIZE;
                    int la = countLattice(i, j, v) + 3 * SIZE;
                    p[l].downNode = new SudokuNode(p[l], null, null, null, columnLists[l]);
                    p[r].downNode = new SudokuNode(p[r], null, null, null, columnLists[r]);
                    p[c].downNode = new SudokuNode(p[c], null, null, null, columnLists[c]);
                    p[la].downNode = new SudokuNode(p[la], null, null, null, columnLists[la
                    p[l] = p[l].downNode;
                    p[r] = p[r].downNode;
                    p[c] = p[c].downNode;
                    p[la] = p[la].downNode;
                    p[l].downNode = columnLists[l];
                    p[r].downNode = columnLists[r];
                    p[c].downNode = columnLists[c];
                    p[la].downNode = columnLists[la];
                    p[l].rightNode = p[r]; p[r].leftNode = p[l];
                    p[l].leftNode = p[la]; p[la].rightNode = p[l];
                    p[r].rightNode = p[c]; p[c].leftNode = p[r];
                    p[c].rightNode = p[la]; p[la].leftNode = p[c];
```

```
                    columnLists[l].upNode = p[l];
                    columnLists[r].upNode = p[r];
                    columnLists[c].upNode = p[c];
                    columnLists[la].upNode = p[la];
                }
            }
        }
    }
    for(int i = 1; i <= MAX_COLUMN; i++){
        //System.out.println(i);
        int s = 0;
        SudokuNode node = columnLists[i].downNode;
        while(node != columnLists[i]){
            s = s + 1;
            //System.out.println(node.columnHead.name);
            node = node.downNode;
        }
        columnLists[i].sum = s;
    }
}

/*public void add(SudokuNode p, ColumnHead head){
    p.downNode = new SudokuNode(p, null, null, null, head);
    p = p.downNode;
    p.downNode = head;
}*/

public int countLocation(int row, int column){
    return 9 * (row - 1) + column;
}

public int countRow(int row, int value){
    return 9 * (row - 1) + value;
}

public int countColumn(int column, int value){
    return 9 * (column - 1) + value;
}

public int countGrid(int row, int column){
    int b;
    if(row <= 3 && column <= 3){
        b = 1;
    }else if (row <= 3 && column > 3 && column <= 6) {
        b = 2;
    }else if (row <= 3 && column > 6 && column <= 9) {
        b = 3;
    }else if (row > 3 && row <= 6 && column <= 3) {
        b = 4;
    }else if (row > 3 && row <= 6 && column > 3 && column <= 6) {
        b = 5;
    }else if (row > 3 && row <= 6 && column > 6 && column <= 9) {
        b = 6;
    }else if (row > 6 && column <= 3) {
        b = 7;
    }else if (row > 6 && column > 3 && column <= 6) {
        b = 8;
    }else {
        b = 9;
    }
    return b;
```

```
    }

    public int countLattice(int row, int column, int value){
        return 9 * (countGrid(row, column) - 1) + value;
    }

    public ArrayList<Integer> possibleNumber(int row, int column){
        ArrayList<Integer> number = new ArrayList<Integer>();
        boolean[] possible = new boolean[10];
        for(int i = 1; i <= 9; i++){
            possible[i] = true;
        }
        for(int i = 1; i <= 9; i++){
            possible[input[row][i]] = false;
        }
        for(int i = 1; i <= 9; i++){
            possible[input[i][column]] = false;
        }
        int b = countGrid(row, column);
        int c = b <= 3 ? 1 : b <= 6 ? 2 : 3;
        int d = 0;
        switch (b) {
        case 1 : case 4 : case 7 :
            d = 1;
            break;
        case 2 : case 5 : case 8 :
            d = 2;
            break;
        case 3 : case 6 : case 9:
            d = 3;
        }
        for(int i = 3 * (c - 1) + 1; i <= 3 * c; i++){
            for(int j = 3 * (d - 1) + 1; j <= 3 * d; j++){
                possible[input[i][j]] = false;
            }
        }
        for(int i = 1; i <= 9; i++){
            if(possible[i]){
                number.add(new Integer(i));
            }
        }
        return number;
    }
}
```

```
package Sudoku;

public class SudokuNode {
    public SudokuNode upNode;
    public SudokuNode downNode;
    public SudokuNode leftNode;
    public SudokuNode rightNode;
    public ColumnHead columnHead;

    public SudokuNode(SudokuNode upNode, SudokuNode downNode, SudokuNode leftNode, SudokuNode r
        this.upNode = upNode;
        this.downNode = downNode;
        this.leftNode = leftNode;
        this.rightNode = rightNode;
        this.columnHead = columnHead;
```

```
    }
}
```

```
package Sudoku;

import java.util.ArrayList;
import java.util.LinkedList;

public class TranslateSolution {
    public ArrayList<LinkedList<SudokuNode>> answers;

    public TranslateSolution(ArrayList<LinkedList<SudokuNode>> answers){
        this.answers = answers;
    }

    public void translate(){
        for (LinkedList<SudokuNode> linkedList : answers) {
            int[][] result = new int[10][10];
            for (SudokuNode sudokuNode : linkedList) {
                int[] a = new int[4];
                int i = 0;
                SudokuNode sNode = sudokuNode;
                do {
                    a[i] = sNode.columnHead.name;
                    //System.out.println(a[i]);
                    sNode = sNode.rightNode;
                    i = i + 1;
                } while (sNode != sudokuNode);
                int x = 0;
                int y = 0;
                int value = 0;
                for(int k = 0; k < 4; k++){
                    //System.out.println(a[k]);
                    if(a[k] <= 81){
                        //x = a[k] / 9 + 1;
                        y = a[k] % 9;
                        y = y == 0 ? 9 : y;
                        x = (a[k] - y) / 9 + 1;
                        //System.out.println(x + " " + y);
                    }
                    if(a[k] > 81 && a[k] <= 162){
                        value = (a[k] - Sudoku.SIZE) % 9;
                        value = value == 0 ? 9 : value;
                    }
                }
                result[x][y] = value;
            }
            for(int i = 1; i <= 9; i++){
                for(int j = 1; j <= 9; j++){
                    System.out.print(result[i][j] + " ");
                }
                System.out.println();
            }
            System.out.println();
        }
    }
}
```

Appendix B Second appendix

some more text **Input C++ source:**

```
//============================================================================
// Name        : Sudoku.cpp
// Author      : wzlf11
// Version     : a.0
// Copyright   : Your copyright notice
// Description : Sudoku in C++.
//============================================================================

#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;

int table[9][9];

int main() {

    for(int i = 0; i < 9; i++){
        table[0][i] = i + 1;
    }

    srand((unsigned int)time(NULL));

    shuffle((int *)&table[0], 9);

    while(!put_line(1))
    {
        shuffle((int *)&table[0], 9);
    }

    for(int x = 0; x < 9; x++){
        for(int y = 0; y < 9; y++){
            cout << table[x][y] << " ";
        }

        cout << endl;
    }

    return 0;
}
```