What to do when it rains: The Umbrella Method for Compiling Sudoku

Control # 2850

February 19, 2008

Abstract

In this paper we detail the Umbrella Method for Compiling Sudoku.

We begin with an exposition of various algorithms for solving Sudoku, before discussing how this and other factors can affect the difficulty of a puzzle. This leads us to conclude that a metric for defining difficulty levels should be based on the methods required for solution.

Following this, we briefly consider the symmetries of the Sudoku Square and Random Number Generation, in order to better understand the mechanics of compiling Sudoku. After an exploration of current common compilation techniques, we formulate the Umbrella Method.

This is a easily implemented method using a computer, which uses random number generation until a unique grid is found, then increases and reduces the grid until the required difficulty is reached.

This results in a wide variety of uniquely solvable Sudoku of every required level.

Contents

List of Figures

List of Tables

1 Introduction

"It sounds crazy, but I've been lying here wishing I had a crossword puzzle to work on."[1]

Bill, in D. E. Knuth's mathematical novel from 1974, when bored, wishes for a crossword puzzle to complete. Now, in 2008, he would be just as likely to wish for a Sudoku! Sudoku were originally invented by Howard Garns in 1979, initially becoming popular in Japan in 1986, and reaching Europe and the America's in 2005[2]. Since then it has been a mainstay of the puzzle pages of most newspapers, with several variations, from 'killer' Sudoku to 'Hypersudoku', or even a Sudocube; a Rubik's Cube with the numbers 1 to 9 on each side[2]. As such, a lot of effort has gone into solving Sudoku, both by hand and by computer, with Thomas Snyder recently winning $10,000 in the Philadelphia Inquirer Sudoku National Championship[3]. However, much less work has gone into compiling them, with many methods relying on guesswork, especially for classifying the difficulty, with many different levels of classification being used. Thus an algorithm to compile puzzles of set difficulties, with a unique solution without a lot of time is sought.

This paper seeks to create a reliable algorithm to compile Sudoku puzzles of varying difficulty. It will do this by:

1. Analyzing the various methods used to solve Sudoku.
2. Analyzing how difficult Sudoku are, and creating a rating system.
3. Discussing and comparing the main methods for compiling grids.
4. Detailing and analyzing our own algorithm.

2 Terminology

Unfortunately, there is no standard terminology yet for describing parts of a Sudoku, and so it is necessary to introduce the terminology which we will use.

- A **cell** is a single square into which a number is entered.
- A **row** is a horizontal collection of 9 cells.
- A **column** is a vertical collection of 9 cells.
- A **box** is a 3×3 collection of cells.
- A **bridge** is a horizontal collection of 3 boxes.
- A **tower** is a vertical collection of 3 boxes.
- The **neighborhood** of a cell is the row, column and box which contain the cell, minus the cell itself.
- A **grid** is the entire Sudoku puzzle.

For a pictorial representation, please see figure 1.

- A **working** object is the object that any algorithm is currently working on, be it a cell, row, column or box.
- A **possibility** of a cell is a value which has not yet been determined to be inadmissible for that cell.
- A **solved cell** is a cell for which the value has already been set.
- A **Unique Missing Candidate** is a single empty cell in a single row, column or box.

- A **Naked Single** is a cell with only 1 possibility.
- A **solved Sudoku grid** is a 9 × 9 grid with the numbers 1 to 9 appearing in every row, column and box, with one value in every cell.

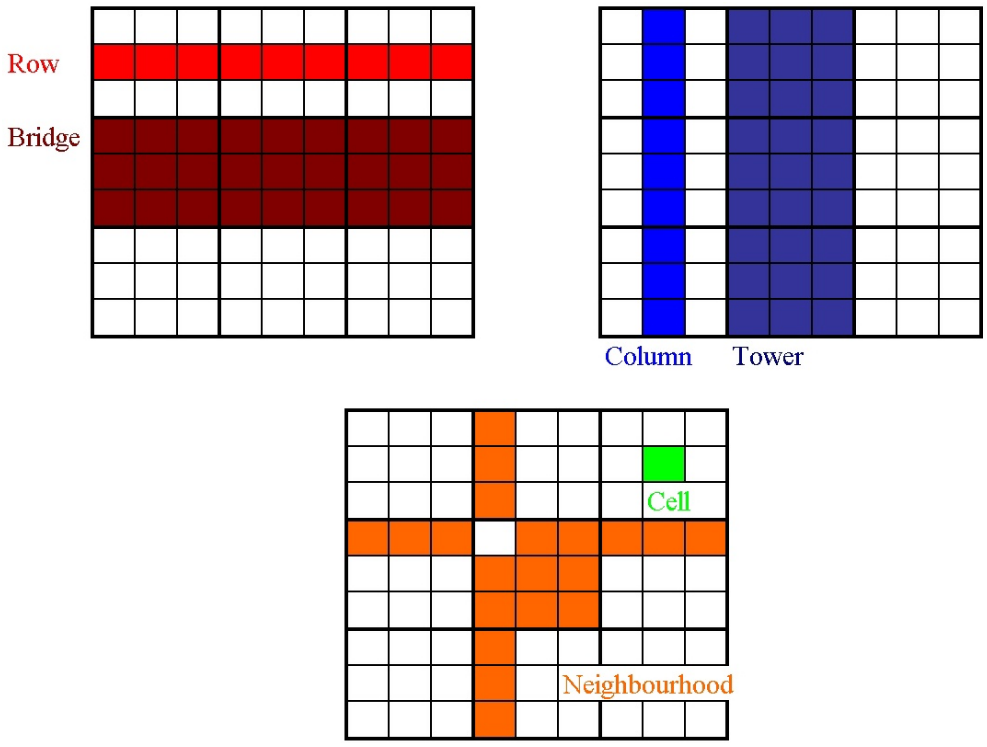


Figure 1: Pictorial representation of the terminology used

3 Algorithms for solving Sudoku

To learn how best to generate a Sudoku of a particular difficulty, it is best to start by looking at various methods of solving them, and how hard they are to solve by hand[1].

3.1 Neighborhood Clearance Algorithm

This is the simplest method for clearing possibilities. Simply put, it involves checking every cell in the grid for naked singles and, having found one, removing the possibility of the naked single from the possibilities of its neighborhood. This solves any Unique Missing Candidates.

3.2 Hidden Singles Algorithm

This method is usually the second or third for any beginner to start to use. It involves working with a row, column, or box. For each value from 1 to 9 that does not already appear in the working object, you check to see how many cells in the object have that value as an allowable value. If there is only one cell that can have that value, then this value is the only possibility. For an example, please see figure 2(a). In this example, we are performing the algorithm on the working column in green. Checking for the value 2, we can see that the top 2 empty cells (arrows in red) can not permit 2 as a value. This means that the only

[1]Methods 1 - 5 have been developed from our experience. Method 6 comes from `www.geometer.org/mathcircles/sudoku.pdf`, as do the names for our methods (Except for Neighborhood Clearance).

cell that can have 2 as a possibility is the bottom cell in the column, and so 2 is the only possibility for that cell. This effectively takes hidden singles, and turns them into naked singles.

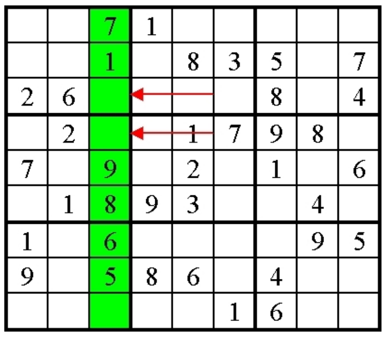
A flowchart of this method, suitable for programming, is in Appendix A.

3.3 Naked or Hidden Groups Algorithm

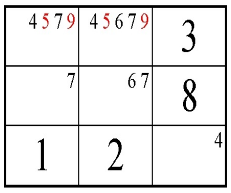
This is the first method requiring more than very basic logic. It involves finding a set of n numbers in a row, column or box, such that all appearances of these n numbers as possibilities for a cell occur in only n cells. This then means that these n numbers must occupy these n cells, and so all other possibilities can be removed.

At no point does this method actually enter the value of a cell, and so it takes a while to see why this actually helps. All it actually does is reduce the list of possibilities for some cells and for the row, column or box in which it is done. For an example, please see figure 2(b). The large numbers are fixed values, and the small numbers are the possibilities for each cell in the box. From this, it is obvious that 5 and 9 (in red) can only appear in the top two boxes. Therefore, 4, 6 and 7 can be removed as possibilities from these boxes.

A mathematical formulation of this method is in Appendix A.



(a) Example of the Hidden Single Method



(b) Example of the Hidden Groups Method

Figure 2: Examples of Methods 1 and 2

3.4 Locked Candidates Algorithm

This is another method that relies on possibilities, and deals with a bridge or tower. We will deal with a tower, but it can be rotated to work for bridges.

Assume that in one of the boxes of the tower, a particular value, a, only appears as a possibility in up to 3 cells, with all of these cells being vertically aligned with each other. This means that, in this box, a must appear in one of these 3 cells, and thus, in the column to which the cells belong, a must belong to that box. Therefore, a can be removed as a possibility elsewhere in the column. For an example, please see figure 3. In this tower, 6 is only a possibility in the top box in the right hand column (In green). Therefore, in this box, 6 must be in the right hand column, meaning that 6 can be removed as a possibility from the right hand column of the bottom box (In red).

A mathematical formulation of this algorithm is in Appendix A.

3.5 Forced Chains Algorithm

This method is the first of the 'trial and error' methods we will mention. To use this method, you pick a cell with as few possibilities available as possible. Then, you simply try each possibility in turn, keeping track of which possibilities are removed from each cell in each run. Then, compare the possibilities that are removed. If any cell always has the

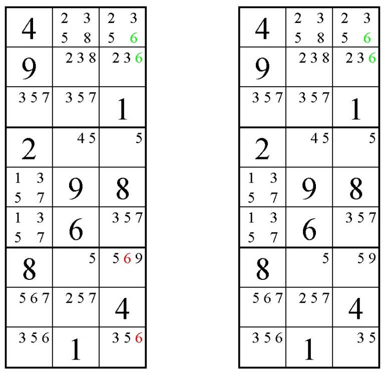
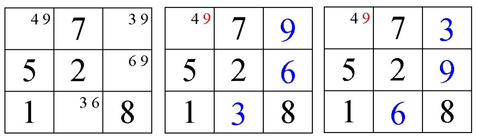


Figure 3: Example of the Locked Candidates Algorithm

same possibilities removed for any possibility of the first cell, then these possibilities must be incorrect. For an example, please see figure 4. Here, we are only dealing with a single box, and are taking our cell to be the top right hand, which has the possibilities 3 and 9 (Figure 4(a)). Trying 9 first, we see that this precludes 9 from the top left hand cell (Figure 4(b)). Trying 3 next, this forces the cell beneath to be 9, also precluding 9 from the top left hand cell (Figure 4(c)). In either case, 9 is not a possibility for the top left cell, and so it can removed, leaving 4 as the only possibility.

The main disadvantage of this method is that it relies upon having a unique solution. However, as a Sudoku is not a Sudoku if it has a non-unique solution, this is not a problem!



(a) The initial box (b) The first option (c) The second option

Figure 4: An example of the Forced Chains Algorithm

3.6 X-wing Algorithm

The X-wing Algorithm[4] is quite complicated. There is a formulation here, and a more mathematical formulation in Appendix A, but for more infomation, see the bibliography reference already citied. We will explain it for rows, but it can trivially be adjusted for columns, and can be generalized to n rows or columns.

Firstly, label the cells of a row x_1 to x_9. For a value of a from 1 to 9, we require 2 rows (r_k and r_l) where the cells with a as a possibility are only x_i and x_j for some i, j. If cell x_i in row r_k is value a, then cell x_i in row r_l is not value a. Therefore, cell x_j in row r_l is value a. In this case, all the cells x_i and x_j except in rows r_k and r_l can not have a as a possibility.

However, if cell x_i in row r_k is not value a, then cell x_j in row r_k must be value a. Therefore, cell x_j in row r_l is not value a, forcing cell x_i in row r_l to be value a. As before, all the cells x_i and x_j except in rows r_k and r_l can not have a as a possibility.

Therefore, all the cells x_i and x_j except in rows r_k and r_l can not have a as a possibility.

There is a mathematical formulation of this at Appendix A.

3.7 Dancing Link Algorithm

Donald E. Knuth's Dancing Links Algorithm[5][6] is currently the most powerful method available to solve Sudoku. It will find every single possible solution for a starting grid, and using this can thus ensure uniqueness of solution. However, understanding how the algorithm works, and in particular how it applies to solving Sudoku, is very tricky!

First, I will describe how the algorithm applies to Sudoku. The Dancing Links Algorithm is an algorithm to find exact covers. The problem we now have is to define a set and subsets such that they define our problem! The elements of our initial set are split into 4 distinct parts. The first part is whether or not a particular cell has a number in it. This corresponds to having a single number in each cell, and thus has 81 elements. The next 3 parts are very similar, and correspond to whether or not each row, column or box has the numbers 1 through 9 in it. This thus has $9 \times 9 \times 3 = 243$ different elements, giving us a total of 324 elements of our set S. The next thing to define is the general subsets with which to cover our solution. These correspond to putting a single digit in each cell. For example, putting a 1 in the top left hand cell corresponds to the subset of S with a number in cell 1, a 1 in row 1, a 1 in column 1, and a 1 in box 1. This thus has $81 \times 9 = 729$ different subsets. The final thing to define is the problem subset. This correspond to checking each cell to see if it has a value already, and then checking each row, column and box to see which values it already has.

As you can see, our problem is now to find an exact cover of a set with 324 elements, out of 730 different subsets! This would take quite a long time (and would actually give us every possible Sudoku grid at this point!). The final thing we need to do is to cut out some of our subsets (and some of our elements) by using the additional constraint that the solution cover must contain the problem subset. This means checking the problem subset for an element s of S and, having found one, removing the element s from our set, and also removing any subset (except the problem subset!) which includes s from our allowable subsets. Eventually the problem subset will contain no elements of S, and at this point it, too, can be discarded. Finally, we apply the Dancing Links Algorithm to obtain our solution.

The Dancing Links Algorithm is a brute force method, and works by taking a subset, removing all the now non-allowable subsets, and picking another subset. This obviously means that this method should only be attempted by hand when the grid is already nearly completed, or with a computer. The reason that the Dancing Links Algorithm is so powerful is that it is based on D. E. Knuth's Algorithm X[7], which makes backtracking very fast when computerized.

4 Difficulty Levels

There are many approaches to the problem of defining a difficulty level for a particular Sudoku puzzle. One way to define relative difficulty is by looking at the length of time it would take someone to solve the Sudoku. This definition causes problems as different Sudoku may take different people different lengths of time depending on the methods required to solve it. In order to resolve this, a method could be devised to take the average length of time that a sample population takes to solve various problems and to use this to find an algorithm to find the relative difficulty. However, this requires a lot of research and finding good algorithms could prove too difficult to find in limited time.

Another way of defining the relative difficulty is to look at the number of times that an algorithm would have to run in order to solve a particular problem. For example, if one puzzle has only 25 squares filled in but another has 30, then if the same methods are used on each, it is likely (though not guaranteed) that the second one will require more loops to complete. This will also be closely linked in with the above method of finding how long it would take a human to solve a problem. Also, a computer could employ different algorithms or apply them in different orders and so two ways of solving a problem may lead to different ideas as to which problem is to be deemed harder.

A further way of trying to find relative difficulty is by looking at the methods required to solve it. If a particular problem requires 'more complex' methods to solve the problem, then it can be deemed a harder problem. However, the order of the various methods must also be looked at. Methods that are 'easier to get your head around' can be one way of classifying the methods, or you can look at the number of steps required to use the method algorithmically (i.e. how abstract or how many logical steps).

In order to try and classify the methods mentioned in the previous section, first note that the definition of a solved grid is that every row, column and box contains one of each of the digits 1,..,9, with every cell having only 1 value. So for the neighborhood clearance algorithm, this is clearly directly checking that these conditions are satisfied and is also very easy to implement, so can be justified to be the easiest algorithm. The 'hidden groups' algorithm is directly a generalization of the hidden singles algorithm and it also contains the neighborhood clearance algorithm[2], and so can be seen to be harder than both of these methods. Similarly, the locking candidates' algorithm also generalizes the neighborhood clearance algorithm and is both conceptually harder and requires more logical steps than the hidden singles algorithm, so can be classified as being more difficult. The forced chains algorithm is very easy to conceptualize, however it dramatically increases the number of steps needed, as some steps have to be taken multiple times, especially if looking at a working cell with more than 2 possibilities. It also requires being able to store the state of a grid, as two have to be compared. Using this method, allowing for as many possibilities in a working cell as needed will always find a solution if there is one, as, given enough time, this method will go through all the possible values of the current grid. Hence this method ends up being quite complex. Which cells are better choices for using this algorithm are also unknown, and hence this method could require a lot of time. Finally, the X-wing algorithm is conceptually the hardest as it requires having the broadest idea as to what is going on. Most methods only require you to look at the corresponding neighborhood or object, whereas the X-wing method requires you consider the entire grid. It also requires a large number of steps in the algorithm, especially for cases larger than 2, since there are a large number of different possibilities to look at to find a candidate for the X-wing algorithm. Whilst using a program[3] to solve existing Sudoku, we have not found a grid which requires both the X-wing and forced chain algorithms[4]. For these reasons, we suggest that the X-Wing method and the Forced Chains Algorithm should be in the same difficulty level.

Further to these general methods of classifying the different algorithms, there is also the problem of how much harder does using certain special cases of an algorithm make it. For example, for the X-wing algorithm, $n = 2$ or $n = 3$ could be looked at, with the latter requiring many more computational steps. Is the hidden groups algorithm only looking at the case when $n = 2$ easier than the locked candidates, whereas using $n = 3$ is harder? Also, rather than basing the difficulty of a problem on how hard an algorithm is being used, there is also the issue of whether using one algorithm multiple times makes a problem harder than using a harder algorithm only once.

One requirement for the classifications of difficulty levels is that there is a significant enough jump between them all. That is, if you took the set of all Sudoku problems that can be considered as one level of difficulty and you compare them to the set of all Sudoku that can be solved by a higher level, then the difference needs to be 'significant'. In fact, in order to define good levels of difficulty, the difference between the different sets should be similar, and they should also cover the range from a genuine (but non-trivial) easy through to a hard puzzle, that can still be solved realistically with the various methods, with consideration for the target audience.

To test if our current difficulty levels conform to this restriction we could take a filled grid and enumerate the cells. Then take the first enumerated cell and remove its value. Then use a computer to use the methods used at each level to try and solve the grid, taking note of which ones can and which ones can't. If one of them can, then remove the value from

[2]If a working cell is filled in, then using the hidden groups algorithm on the other 8 cells in an object (of which the working cell is a part) and the 8 values not equal to the value that the working cell has, leads to removing the working cell's value as a possibility from the other cells

[3]Use of a computer program to solve Sudoku problems occurs frequently throughout this project. The authors would like to note that these programs were modified from code developed by themselves over the last year, in particular over Easter 2007

[4]These methods were tried on 10 of the hardest Sudoku published for solving by the general public

another cell and continue. If no methods work, this cell value can't be removed, and we must backtrack before moving onto the next cell. Continue like this until all cell removals have been tried with each of the difficulty methods. Then, compare the number of different grids that each method can solve in order to decide whether or not the levels chosen were 'good' ones. Obviously, this requires a base difficulty level and a maximum difficulty level already set in place for which to compare intermediate levels. Since the numbers of solvable grids will be large and other solved grids are similar (all share a certain number of properties), it could be conjectured that knowing the values for this one grid will be enough to determine whether the difficulty levels set are good ones, rather than trying out every grid that exists.

Obviously, this is computationally unfeasible, since the number of possibilities for taking out any combination of numbers in any one grid is 2^{81}, although once you got down to the point where no methods could solve a grid, you wouldn't need to remove any other numbers to test. For each grid, you would have to attempt to solve it up to 4 times, in order to test each difficulty level (more if you were trying many different algorithms to try and come up with a good difficulty level). This gives about $2^{81} \times \frac{4}{2^{17}} = 2^{66}$ attempted solutions as an upper bound, where 17 is the conjectured minimum amount of filled cells needed to define a unique grid[8]. However, again, in the same way that you could approximate the total number of grids by only starting off with a base case of one grid, you could approximate this one grid by starting with a grid in which some of the numbers have already been removed. As long as your first level (easiest) is still able to solve this grid, then this will give a reasonably good idea as to the jumps between the difficulty levels. This also gives a computationally feasible way of trying to find whether the methods that you've chosen for being in different difficulty levels are good methods to choose. We call this process 'shredding'.

We decided at first to try splitting the levels like this, pending further analysis:

- Level 1: Using only the neighborhood clearance algorithm
- Level 2: Using the above and the hidden singles' algorithm
- Level 3: Using the hidden groups and locked candidates algorithm
- Level 4: Using level 3 algorithms and X-wing and forced chains algorithm (for $n = 2$)

We tried the shredding method[5] on various Sudoku, which had been previously classified as fiendish[9] (some at level 2, most at level 3), and didn't have too many routes to work through. The results are shown in table 1. These point towards the jump from level 3 to level 4 as not being that large, as there are not many level 4's created from level 3 grids. This shows that our current definitions for setting the difficulty levels may not be very evenly spread. However the results from removing one number only (In Appendix B) suggest that from a level 1 to a level 2 and from a level 2 to a level 3 or level 2 to 4 is not too difficult to do, and hence is a good definition for difficulty levels. However, from level 3 to level 4 is similarly not so good a ratio and so implies that this might not be such a good distinction. However, looking at computational difficulty of the level 4 methods as well as the leap in the difficulty of the concept leads to this being a valid choice. In order to properly decide whether or not these are good difficulty levels it would be necessary to use the process of shredding on a starting grid with more solved cells, which would give a better idea as to exactly how many grids of each different difficulty level there are.

Grids Tried	Level 1	Level 2	Level 3	Level 4
825	0	3	19	2

Table 1: Results from shredding fiendish puzzles

One issue is that many of our starting grids were pre-defined when we analyzed them. This means that in fact, whilst the different levels look reasonably spread with the data we

[5]Our programs did not include the X-Wing method, but as mentioned before this is roughly equivalent to the forced chains algorithm, which was included

have, the data is skewed. There are fewer readings of level 2 difficulty levels compared to level 3 than there are in reality. This means that there should probably be an extra difficulty level inserted between levels 2 and 3.

In order to add in a new difficulty level between 2 and 3, we could see how many instances of the hidden singles algorithm is required, based on the simplest first method, by which you use the neighborhood clearance algorithm as many times as possible, then use the hidden singles algorithm, before repeating. In this way, you can find out how many times the hidden singles algorithm is needed (although this result may change depending on which working objects were chosen). Otherwise, you could look at the number of algorithms that are required to solve a problem. However, more analysis would need to be done to accurately determine where the difficulty level should be set. Also, depending on the target audience, another difficulty level may not need to be added, if the people who were solving the Sudoku were more used to solving harder problems.

Overall, we have decided that the difficulty levels that have been described above are reasonable definitions, and to get a better one would require more analysis than time currently allows.

5 Sudoku Squares: A Special case of a Latin Square

If we are to be able to construct Sudoku grids, we need to have some idea of what makes one completed grid different to any other completed grid, and how difficult it is to find them.

A Latin Square is a $n \times n$ grid, with the numbers 1 to n in every row and column. From this, it is obvious that a Sudoku square is a special case of a Latin Square, with the additional constraint that the boxes must also contain the numbers 1 to n[10]. So, how likely is it that a 9×9 Latin Square is a Sudoku square?

The answer is not very likely at all. There are 5.5×10^{27} distinct Latin Squares[10], but only 6.7×10^{21} distinct Sudoku squares[11]. This is a difference of a factor of 8.3×10^{5}. This tells us that the likelihood of finding a Sudoku square, even with a Latin Square, is incredibly small! And, if we increase the starting grids to any grid with 9 appearances of the digits 1 to 9, the likelihood of it being a Sudoku grid plummets to 1 in 6.6×10^{87}!

So, if finding grids is so unlikely, is there any easy way to algorithmically do it? Unfortunately, without using Knuth's Dancing Links Algorithm and back tracking, the answer is no. It is just too likely that something will create a grid that is impossible to solve. However, if you have any grid, it is possible to find lots more due to the group symmetries of a Sudoku!

5.1 The symmetries of a Sudoku square

So, what group operations can we perform on a Sudoku grid[12]? There are 5 main types, in two distinct groups - Permutations, and rotations.

5.1.1 Permutations

The first permutation that has to mentioned is a permutation by S_9 of the digits of the grid. Obviously, if we do this, we will still have a Sudoku.

All the other permutations are of S_3, as they all involve permuting 3 objects. The first 2 we will mention involve permuting within the towers and bridges. We will only explicitly explain why permuting within the towers is an allowable operation. The first thing to note, is that all the columns still have all the numbers from 1 to 9 in them, as nothing has disturbed that. Also, no numbers have moved up or down within the columns, therefore they are still within the same box. Finally, as the numbers have only moved horizontally, all of the rows still have all the numbers from 1 to 9 in them, so the grid is still allowable. The final two permutations involve permuting the towers and bridges themselves. A very similar treatment will show that this is also an allowable operation.

It is possible to flip the grid in the vertical axis by permuting the outside towers, and then permuting the outside columns in each tower. For example, starting from figure 5(a), if we permute the outside towers, we get figure 5(b). If we then permute the outside columns of each tower we get figure 5(c), which is just a flip in the vertical axis.

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

(a) Our starting grid

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

(b) Permute the first and last towers

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

(c) Permute the first and last columns

Figure 5: The equivalence of permutations and reflection

5.1.2 Rotations

There are two rotations possible. The first is a rotation by $90°$, and the second is a reflection. However, a reflection is also the result of certain permutations, as shown above. Therefore we can ignore a reflection, as it is part of the permutations of the square.

So how many Sudoku squares does this mean are non-equivalent? One might be tempted to take the total number of Sudoku squares (6.7×10^{21}), and divide by the total number of permutations ($9! \times 3!^8 \times 2$). However, this gives a non-integer answer, roughly equal to 5.472×10^9, due to some operations being equivalent to the identity operation only for particular grids. The exact answer, by an application of Burnside's Lemma, is actually 5.473×10^9. However, this does have a very strong correlation with our rough calculation, down to the 4th significant figure.

6 Random Number Generation

Some of the most well known ways of generating random numbers[13] are by the use of dice, coins, cards, roulette wheels, or other physical objects. These objects are widely available for the general public to use in everyday situations, but their use is, in general, quite limited and time-consuming. One of the first tables of random numbers produced, for experimental uses and other applications requiring large lists of numbers, was by L.H.C Tippett in 1927 from 'digits taken at random from census data'. One of the significant problems with random number tables however, is the amount of memory needed if they are to be stored on a computer. If they are not stored, then they will have to be input whenever they are required, which would be extremely time-consuming and inefficient.

For this reason, computer scientists and numerical analysts have tried to produce algorithms that output random (or pseudo-random) numbers. One of the most well known and most widely used in computers was developed by D.H Lehmer in 1951[14];

Choose 4 Integers;
m - modulus
a - multiplier
c - increment
x_0 - starting value
Where $m > 0, 0 < a < m, 0 \leq c < m, 0 \leq x_0 < m$

Then

$$x_{n+1} = (a \times x_n + c) mod(m)$$

This is a *linear congruential sequence.* The sequence is only pseudo-random as each entry is determined by the entry before. As the algorithm is completely dependent on the values of m, a, c and x_0, it will produce different sequences for different values of these. For certain values, it can produce sequences with short periods (in which the sequence enters a cycle, during which numbers recur sooner than they should). For example, with $m = 10, a = c = 6, x_0 = 1$, the produced sequence is $2, 8, 6, 2, 8, 6, \ldots$. It has been shown by number analysis that this can be avoided if m is large and b is of order $\frac{m}{10}$, and is an arbitrary constant ending with the digits $..x21$, where x is even[15]. An effective way of generating a starting seed is to use the current time in millisecond, *mod* a reasonable value.

By using the formula $I_n = \lceil \frac{x_n}{m} \rceil \times r$ we will be given random numbers in the interval $[1, r]$. There are several statistical tests that have been carried out to verify that these choices produce suitably random sequences, i.e. Kolmogolov- Smirnov test, Poker Test, Permutation test, and χ^2 test. An even 'more random' selection of numbers can be gained if a random selection of entries is chosen from the created random sequence. As most computers, and computer software will use this, or similar, algorithms to create random numbers, anytime an object is required to be selected at random for the process of creating a Sudoku, a computer command will be used to generate the numbers.

7 Compiling Sudoku puzzles

There are many different factors that can be taken into account when compiling Sudoku. Obviously, the most important are that they obey the rules, are unique and are solvable by logic. However, less important considerations are its aesthetic appeal i.e. rotational symmetry, evenly spread numbers, and how 'fun' it is to solve. These considerations will be discussed with reference to each of our methods.

As previously mentioned, there are two major types of method commonly used to compile Sudoku puzzles. The first method involves starting with a finishing pattern of givens in place, and filling the pattern in, whereas the second starts with a full grid, and slowly removing numbers until you reach the pattern you want. These methods are sufficiently different to require separate handling. We will also mention a completely random way of choosing grids, before comparing the methods. However, we will first discuss how the pattern itself can affect the possible grids.

7.1 Patterns

The pattern of cells which are completed, and which are left, is important in making sure that a Sudoku has a unique solution. For instance, clues given only in boxes 2, 4, 5, 6 and 8 cannot give enough information about the rest of the boxes, to ensure uniqueness (See figure 6).

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

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

Figure 6: An example of non-uniqueness

As the towers and bridges can be permuted within their rows/columns, without losing validity as a solution, this applies also to several other combinations in which four boxes are left entirely blank. It has been conjectured that the minimum number of entries that could ensure a unique solution is 17; however, grids with as few as four blank cells can also have no unique solution[8]. This indicates that the number of cells left in a grid bears little relation to whether or not it is solvable. In fact, within most grids there are certain cells, or groups of cells for which it is essential that at least 1 appears visible in order to ensure uniqueness. Due to these problems with uniqueness, setting which cells are to be revealed before completing the grid can lead to unsolvable, or invalid, problems, even when a valid grid is found. Figures 7(a) and 7(b) show two puzzles starting with the same pattern of revealed cells. While the first is uniquely solvable at difficulty level 3, the second can lead to two different solutions (See figure 7(c)). Removing any more than 3 cells from a completed grid can also lead to non-uniqueness, whether the pattern is set initially, or afterwards. For this reason, it is important to check that the puzzle is a true Sudoku after it is constructed. This can be done by computer Sudoku solvers, such as the Dancing link Algorithm.

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

(a) First Grid: Solvable at level 3

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

(b) Second Grid: Non-unique

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

(c) Both of the possibilities of the second grid

Figure 7: An example of the importance of the starting pattern

In addition to the considerations for uniqueness, another issue for consideration is whether symmetry (rotational or reflectional), is desired in the puzzle. This is, in general, an aesthetic concern. Also, pictures or patterns such as lattices may be desired[16]. If none of these aesthetics are required, a pattern can be produced by randomly choosing squares. If symmetry is required, then a random number generator can again be used, with the chosen cells corresponding opposite also being chosen/removed.

7.2 Additive Compilation

When generating a Sudoku from a blank grid with a preset pattern, there are several methods that can be used. One method is to start by filling in the entire set of one's, then the two's, and so on, checking with each entry that the rules for a Sudoku grid still apply. This can either be done completely randomly, using a number generator to assign cells, or by initially assigning numbers randomly, and then working out their logical consequences. With this method, if a point is reached such that a number cannot be placed without breaching the rules, it is necessary to back track, and delete previously placed entries. This will then allow other combinations of placements to be attempted. Using random permutations of the nine numbers to fill in the grid row by row, (or columns/boxes) is another similar technique.

A more logical way to complete a grid is to place one number on a cell that is to remain blank, and then place numbers on the surrounding 'visible' cells that will allow this cell to be solved with this number. These newly entered numbers will then lead to the possible positions of other numbers. Again, this method relies on some elements of random choosing, and may require some backtracking. With this method, it is also possible to miss logical steps that determine the numbers to be set in certain places. This could lead to problems with solvability and non-uniqueness[17].

7.3 Reductive Compilation

For this method, the compiler starts with a completed Sudoku solution grid. The aim is to produce a puzzle by removing elements from the grid. This reduction can be done either randomly, by removing cells/pairs of cells chosen at random, or deliberately in order to reach a desired pattern. At any stage in this method, the removal of a certain cell may cause the puzzle to become non-unique, and hence, no longer a Sudoku. Consequently, with each removal it is necessary to use an algorithm to verify the continuing uniqueness of the puzzle. If the puzzle loses uniqueness at any point, then the last element to be removed must be replaced. At this point the compiler may decide to leave the puzzle as it is, or may attempt to take away a different element.

Trying to produce a Sudoku that fits a specific pattern with this method is very difficult. If any one element needs to be removed from a cell that is required to be filled to ensure uniqueness for that specific grid, then the puzzle will no longer work. In this instance, the pattern must be modified, in order to enable a valid puzzle to be created. Choosing the pattern at random is in many ways the best method for creating puzzles when starting with a completed solution grid[18][19].

7.4 Random Compilation

This method relies very heavily on a computerization of the Dancing Links Algorithm. However it does guarantee a Sudoku and uniqueness. It involves randomly entering digits into cells, and then checking, via the Dancing Links Algorithm to see if it has at least 1 solution. If it has no solutions, it back tracks 1 step, and then continues. If it has more than 1 solution it also continues entering digits. If, however, it only has 1 unique solution, it stops.

7.5 Comparison

All of these methods have their different strengths and weaknesses that complicate the production of Sudoku. With the reduction method, a valid Sudoku can always be found by replacing the last cell that was removed (provided it is being checked for uniqueness with every removal). With the random method, a valid Sudoku will also be produced with every attempt. The additive compilation method however can lead to a non-unique problem, even after completing a full solution grid. This would mean that either the grid would have to be effectively scrapped, and a new one started, or a significant number of steps would need to be back-tracked. This could obviously take a long time, especially as there can be no guarantee that it will not occur more than once. For a computer this is clearly not a problem, given their memory capacities and processing power. If it is to be attempted by hand, there will be the obvious problems of remembering, and keeping track of, the order in which cells are filled, how far back the backtracking went, and what solutions have been previously attempted.

The significant reduction in the number of times that the compiler needs to backtrack in the reduction method before a valid Sudoku is found could be seen as an advantage over the additive method. The drawback however is the need to check for uniqueness at every step. With access to a computer this can be done quite easily with Dancing Links, but if working by hand, this may become a challenge once the solution grid has been significantly reduced. The number of times that the compiler would be required to backtrack would also be increased in this method if, after one removal proves invalid, the compiler then attempts to remove a different cell. This may be done in order change the difficulty level. At this stage, the compiler can encounter some of the same problems as when using the additive method if they are attempting to create the Sudoku by hand.

The difficulty when working without a computer applies also to the random method, due to the use of the dancing links algorithm to check for uniqueness. Backtracking in this method occurs when the random number generator produces a combination of number and cell which do not fit the Sudoku rules. As in the reduction method, this involves only going back by one entry each time.

One problem with the reductive method is in how to produce the initial grid. Those methods used to create a solution grid in the additive method can be used here, but the same problems will occur as in the additive method. Grids can also be created by permutations of any one grid, but this will lead to only a limited number of grids, and very little variety in the Sudoku and their solutions. The random generation used in the random and additive methods will produce a lot of variety between puzzles. With a computer, it is possible to use the Dancing Links Algorithm to produce an assortment of grids. This would increase the amount of variety within the different Sudoku, but these would probably be very similar, due to the way they are produced in a logical order.

With the additive method where the entries after the initial placement are added following logic rather than at random, if someone tries to solve the finished puzzle, they will generally have to do it in the opposite direction to which it was first created. Normally, this would not pose too many problems, unless the creator wished to solve it themselves, although in some instances it could lead to the solution requiring some extremely complex methods initially and only easier methods as it nears completion. This does not occur in any of the other methods. The level of enjoyment gained by completing the Sudoku could suffer because of this.

The ease with which a chosen difficulty level can be reached varies greatly among the methods. While the difficulty levels are defined by the algorithms needed to solve the Sudoku, reaching a preset level can present quite a problem. With the additive method, there is no way to actually choose what level you wish to have. The level of a produced Sudoku can be discovered, but the production of the Sudoku could create a puzzle of any difficulty. For both of the other methods, it is easier to choose what difficulty level you wish for a created Sudoku to be. The random method starts with more difficult problems, and generally works up towards the easier levels. A level three problem can be converted into a level two puzzle by the addition of more values into the randomly chosen cells. The values that are taken will be specified once the Sudoku has achieved uniqueness. In the same way, a level two problem can become a level one. In theory, this method seems to be quite adaptable to finding each of the different levels; however in practice, it will often miss out the higher levels, and first produce a Sudoku of level two or lower. This is mainly due to a combination known as a Unique Rectangle[4], in which you have 4 cells arranged in a rectangle, each of which can take one of two values. In order for this to be solvable, at least one of these numbers is required.

The reductive method works in the opposite direction. From the original completed grid, cells are emptied to produce a level one puzzle. As cells are removed, either the puzzle will stay at the same difficulty level, the difficulty will increase to a higher level, or the puzzle will become unsolvable. The method of shredding used in trying to define difficulty levels is actually an extremely powerful reduction technique, which can remove several cells at once, and analyze the levels of the produced Sudoku. From this analysis, it can be seen that it is possible to reduce lower level grids to create higher difficulty puzzles. This does however require a computer, as the number of times which unsolvable puzzles are created by a removal and the move needs to be reversed can become quite large. Occasionally, however, a point may be reaching in which any removal from the grid will result in non-uniqueness. If this occurs before a desired difficulty is reached, then the compiler may need to backtrack several stages.

In conclusion:

- The additive method does not produce a Sudoku with every trial.
- There is more variety of puzzles provided by the additive, and random methods, than if the reduction method is used from a set subset of Sudoku grids.
- The additive method requires significantly more backtracking and repeating of steps to create a Sudoku. Random and reductive methods require only backtracking one step at a time to reach the same point.
- All methods ideally require the use of a computer.

- The additive method can only be classified for difficulty level; it cannot be produced to a certain level. The random method will sometimes miss out harder difficulty levels, while the reductive method will occassionally reach a point where no more removals may occur before the required level is reached.

8 The Umbrella Method

So far we have discussed how we are going to rate Sudoku, and various methods already used to compile Sudoku. The main problems encountered are getting non-unique grids, and predicting the level of the Sudoku. The method we suggest using is as follows:

- Stage 1: Start with a blank Sudoku grid. Randomly place the numbers 1 to 8. At this point it is impossible to have an unsolvable grid.
- Stage 2: Generate a random digit, x, between 1 and 81. Find out if cell x is empty. If the cell is not empty, begin Stage 2 again. If the cell is empty, place a random digit between 1 and 9 in the cell. Finally, use the Dancing Links Algorithm to check for at least 2 solutions. If there are 2 or more solutions, begin Stage 2 again. If there are less than 2 solutions, go to Stage 3.
- Stage 3: See how many solutions there are. If there are no solutions, remove the last value, and go back to Stage 2. If there is only 1 solution, note down the solution, and continue to Stage 4.
- Stage 4: Check the grid to see what level it is currently at. If it is currently unsolvable using any level method, generate a random digit x between 1 and 81, and fill cell x in with its unique value, and go back to Stage 4. If it is at level n, you now have a level n grid, and continue to Stage 5.
- Stage 5: Attempt to solve your level n grid using level $n-1$ methods only. If a partial solution is obtained, it should then be inspected, and just enough digits entered to enable a level $n-1$ method to proceed. If a full solution is obtained, you now have a level $n-1$ grid. Let $n = n-1$, and return to Stage 5.

There are several advantages of this method.

1. You are guaranteed to get a Sudoku grid with a unique solution as, if at any point you have a non-unique solution, you back track a single step to get multiple solutions.
2. It is possible by this method to generate any of the 5.5×10^9 complete Sudoku grids, thus giving you a wide variety of problems, hopefully making our grids 'fun' for a human solver.

There are unfortunately also several disadvantages of this method.

1. This method is not really feasible for a human compiler, especially in the early stages, due to the use of the Dancing Links Algorithm. To give an idea of the scale of this problem, with 8 digits entered into the grid, you are trying to find an exact cover of a set with 284 elements from between 497 and 553 subsets of your set.
2. It is also possible for the first unique Sudoku grid to not be of the highest difficulty. This is mainly due to Unique Rectangles.

We ran this method 5 times in order to initially evaluate its effectiveness. Unfortunately, after Stage 3, we checked the grids for their difficulty, and they were all at Level 2. This suggests that some modification is necessary.

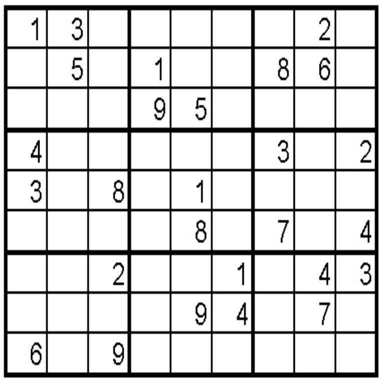
However, from the results in Appendix B, you can see that Level 2 Sudoku were reasonably effective at 'containing' both Level 3 and Level 4 Sudoku. From the 22 Level 2 Sudoku that we tested, by removing only one value at a time we found 6 Level 3 Sudoku, and 6 Level

4 Sudoku. This is a reduction of approximately a quarter. Our first method is very fast at producing Sudoku, and so finding enough Sudoku to test for contained Sudoku should not be a problem.

This suggests that we should modify our method as follows:

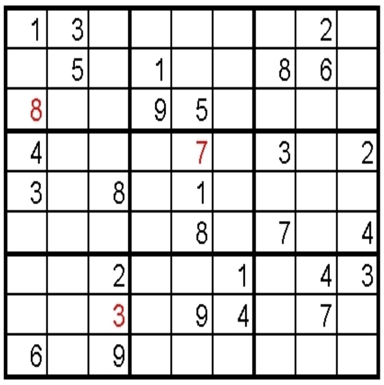
1. Stage 1(b): Follow our previous method until the end of Stage 3.
2. Stage 2(b)Check the grid for its level. If it is the right level, end. If it is below the level required, use the 'shredding' method to increase the level. If this does not work, add a single number into the grid, and repeat from Stage 2(b). If it is above the level required, add a single number into the grid, and repeat from Stage 2(b).

Examples of the Sudoku created by this method are below in figures 8 and 9. All changes are marked in red. Figure 8(a) is the grid initially produced by our random production method, at level 2. Figure 8(b) is our initial grid with 3 numbers added, making it a level 1. Figures 9(a) and 9(b) are Levels 3 and 4 respectively, each involving the removal of 1 digit from our starting grid.



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

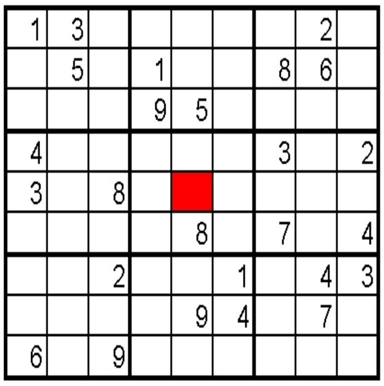
(a) Our initial Level 2 Grid



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

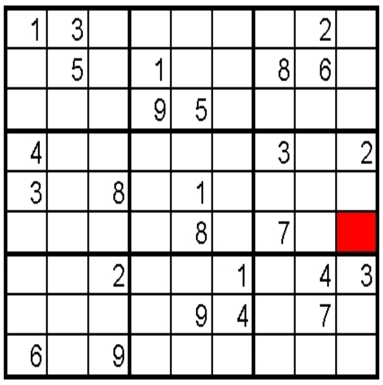
(b) Our Level 1 Grid

Figure 8: Our Level 1 and 2 grids



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

(a) Our Level 3 Grid



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

(b) Our Level 4 Grid

Figure 9: Our Level 3 and 4 grids

The umbrella method which we have produced is not feasibly implemented by hand, however using a computer, and the correct software, it becomes relatively quick and easy to execute. In essence, the random compilation method is used to build a Sudoku, while the reductive compilation method is used in the refining of the difficulty level of the product.

As such, many of the strengths of the methods are exploited, while their weaknesses are minimised. For instance, the random beginnings of the method will ensure there is a lot of variety between the different puzzles which are created, and will also guarantee that a Sudoku will always be found. The provision that, if a grid cannot be reduced any further or the level is too low then a new cell will be added at random reduces the problem of being unable to reach certain difficulty levels. Problems caused by unique rectangles in the random method, will generally not be as severe in the umbrella method, due to the addition of the reduction method techniques.

There are still problems though. Like in the reduction method, it may still be possible to find a grid with a relatively high number of cells filled in that cannot be reduced using the shredder. This example (Figure 10) is a level 1 problem, but using the shredder reveals that there only 4 level 2 possibilities and no level 3 Sudoku problems. Using our shredder on this example required testing 3880 different Sudoku grids. As, in general, adding an extra number doubles the number of Sudoku problems that the shredder would have to solve (although this can be improved upon, since the number that has just been entered is fixed), by only adding a few numbers, the amount of processing required soon becomes too large to be feasible.

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

Figure 10: A problem grid for our Shredder method

Another problem with this method is what happens when you get unique rectangles. Since many grids contain them, building up a random grid by using the dancing links algorithm leads to a likelihood of finding a grid with a unique rectangle in. If this happens, then the dancing links algorithm will keep giving a result of it being non-unique, until one of those 4 squares have been managed to fill in.

In order to get around this problem, one method is by taking a record of which numbers you filled in, in which order. If you find that you have to use a large amount of numbers to define a unique grid (therefore making the shredder method too complicated), take the final number that you had to use. This is likely to be a number for one of the unique rectangles. Remove all the numbers, and go through the process again, but this time filling in this cell first, before following in the same order as before. Again repeat the process. Eventually, you should get a unique solution for which the number of cells is not so large that the shredding method doesn't work. This number depends on the efficiency of the shredding algorithm, as well as time constraints and processing constraints, however, an absolute maximum of 36 cells filled in would be a good start, since any more than this probable makes the problem too easy[6].

If you ignore the above method for getting around the problem of unique rectangles, the method itself is not particularly complex, although it does require the use of a computer to do the dancing links algorithm and shredding. Any other method would also need to use many of the algorithms in the shredding algorithm, since you need to be able to find out

[6]while there is not a direct correlation between the number of cells filled in and the difficulty level, nevertheless, there is a correlation, and any grids with more than this number can be deemed too simple (looking at pre-existing Sudoku problems, and counting their totals)

what difficulty level a certain problem is.

One advantage of this method is that it is easily extensible to include finding grids with certain properties, for example symmetry. For this method, when picking a random place to put the next number, require that the number placed after this is placed in the correct symmetric place, filling it in with a random number and checking that this still defines a possible grid. This method would be more complex and is harder to find a grid of the correct difficulty level, but could be done.

A flow chart of this method is shown in Appendix C, and this shows that the method is relatively simple to implement. The method has been termed 'the umbrella method' because of the property that as you start looking for a grid, first of all, you fill in numbers one at a time. Once a number is in the grid, having been checked that it does not define an impossible-to-solve grid, it is not removed during this stage. Hence the grid can be thought of as being a line, slowly becoming more difficult. Once the point is reached where the shredding method is needed, it checks all strands of the grid, removing each number in turn. With each number removed, the grid can be thought of as getting harder. Hence if the difficulty level is plotted against the one dimensional representation of the grid that is being filled in, then it looks like figure 10, which we liken to an umbrella.

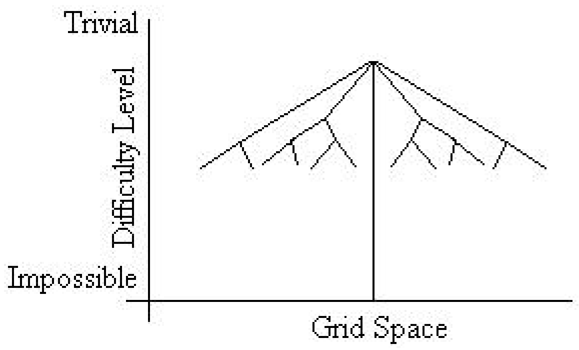


Figure 11: A Pictorial representation of the Umbrella Method

9 Conclusion

Throughout this project, we have considered many different techniques, methods and factors that can affect the production or classification of Sudoku. Some of these have been discarded, while others have become integral parts of our final method; the Umbrella Method.

When considering difficulty levels, we decided that the most logical way to classify the different levels was by the logical methods needed to complete the puzzle, with each increasing difficulty level requiring more complex methods. While we believe our grading system to be reasonable, with more analysis the system could be extended. A direct extension would be to find and classify further methods of solution. Another method could be to take account of the number of repetitions of each method needed to complete a Sudoku, or the total number of algorithms a computer must use.

Our algorithm for creating Sudoku was produced from the amalgamation of two separate methods used to create Sudoku. In brief, a Sudoku is created by the addition of random numbers to a blank grid, until a grid is produced which has a unique solution. At this point, the current difficulty level is established. If a harder Sudoku is required, then the shredding technique is applied, to find a suitably difficult puzzle. If the requirement is for an easier puzzle, the compiler has to fill in more random cells, to try to obtain a puzzle of the required level.

The method in itself is not complex; it is only the necessity to utilize and program a computer in order to complete the otherwise extremely time consuming actions of the dancing links algorithm and shredding that causes this algorithm to be considered anything other than simple.

Without a computer we suggest using a reductive method, but be prepared for a lot of backtracking.

References

[1] D. E. Knuth *Surreal Numbers* Addison-Wesley, Massachusetts 1974

[2] http://en.wikipedia.org/wiki/Sudoku

[3] http://wpc.puzzles.com

[4] www.geometer.org/mathcircles/sudoku.pdf

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

[6] http://en.wikipedia.org/wiki/Dancing_Links

[7] http://en.wikipedia.org/wiki/Algorithm_X

[8] http://en.wikipedia.org/wiki/Mathematics_of_Sudoku

[9] The Times One a Day Sudoku Calendar, 2005 - Used for analysis.

[10] http://en.wikipedia.org/wiki/Latin_squares

[11] www.afjarvis.staff.shef.ac.uk/sudoku/felgenhauer_jarvis_spec1.pdf

[12] www.afjarvis.staff.shef.ac.uk/sudoku/sudgroup.html

[13] D. E. Knuth *The Art of Computer Programming: Volume 2* Addison-Wesley, Massachusetts 1998

[14] http://en.wikipedia.org/wiki/Random_number_generation

[15] R. Sedgewick *Algorithms* 2nd Edition Addison-Wesley, Massachusetts 1988

[16] http://people.csse.uwa.edu.au/gordon/sudokupat.php

[17] http://puzzle.gr.jp/show/English/LetsMakeNPElem/01

[18] www.sudokuessentials.com/create-sudoku.html

[19] http://contentdig.com/recreation-and-leisure/how-to-create-your-own-sudoku-puzzle.html

[20] http://sudoku.sourceforce.net - Used for its user friendly implementation of the Dancing Links Algorithm.

A Algorithms

A.1 Hidden Singles Algorithm

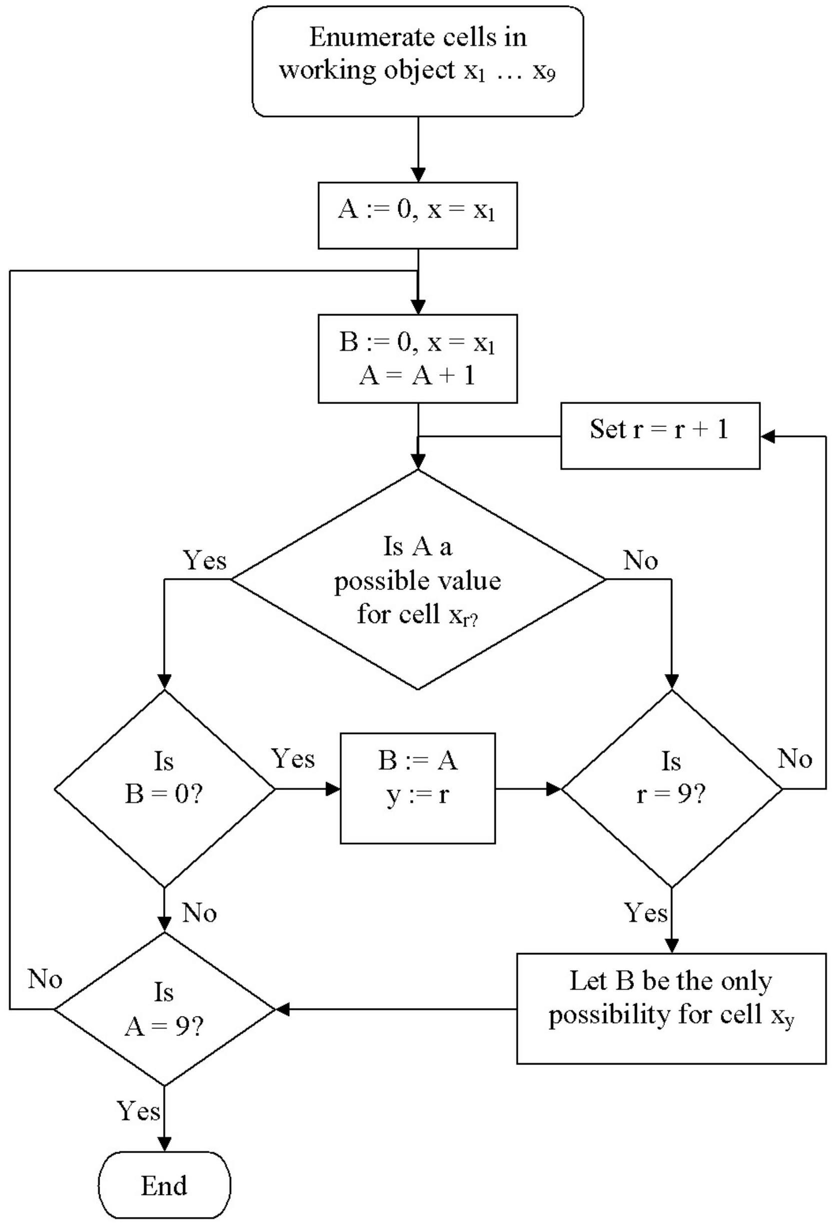

Enumerate cells in working object x1 … x9
A := 0, x = x1
B := 0, x = x1
A = A + 1
Set r = r + 1
Is A a possible value for cell xr?
Yes
No
Is B = 0?
Yes
B := A
y := r
Is r = 9?
No
No
Yes
No
Is A = 9?
Let B be the only possibility for cell xy
Yes
End

Figure 12: A Flow Chart of the Hidden Singles Algorithm

A.2 Hidden Groups Algorithm

Take a working object, with cells x_1 to x_r. Then, for $z = 1$ to 9, set $S = \{x_i : z \text{ is a possibility in } x_i\}$. If $|S| = |\{y \in \{1 \ldots 9\} : y \text{ is a possibility in } x_i \Rightarrow x_i \in S\}|$ then $\forall z \notin |\{y \in \{1 \ldots 9\} : y \text{ is a possibility in } x_i \Rightarrow x_i \in S\}|$ the z is not a possibility for each $x_i \in S$.

A.3 Locked Candidates Algorithm

If $\exists$ 3 cells, $x_1 \dots x_3$, x_i belonging to the same box and the same column or row, such that $\exists y \in \{1 \dots 9\}$ such that y is not a possibility in the box$\setminus\{x_1 \dots x_3\}$, then y is not a possibility in the column or row$\setminus\{x_1 \dots x_3\}$.

A.4 General X-Wing Algorithm

For each value of $a \in \{1 \dots 9\}$, if $\exists r$ cells $x_1 \dots x_r, r < 9$ for which $|\{y$ a row : $\exists i$ with a a possibility in cell x_i in row $y\}| = r$. Let these rows be $y_j, j \in \{1 \dots 9$. Then, for $x \notin \{x_1 \dots x_r\}, a$ is not a possibility for cell x in row $y_j, j \in \{1 \dots 9\}$.

B Statistics

Starting Level	Starting Number of Grids	Grids Created		
		Level 2	Level 3	Level 4
1	17	17	0	0
2	22		6	6
3	8			1

Table 2: Grids created by removing 1 number

C The Umbrella Method

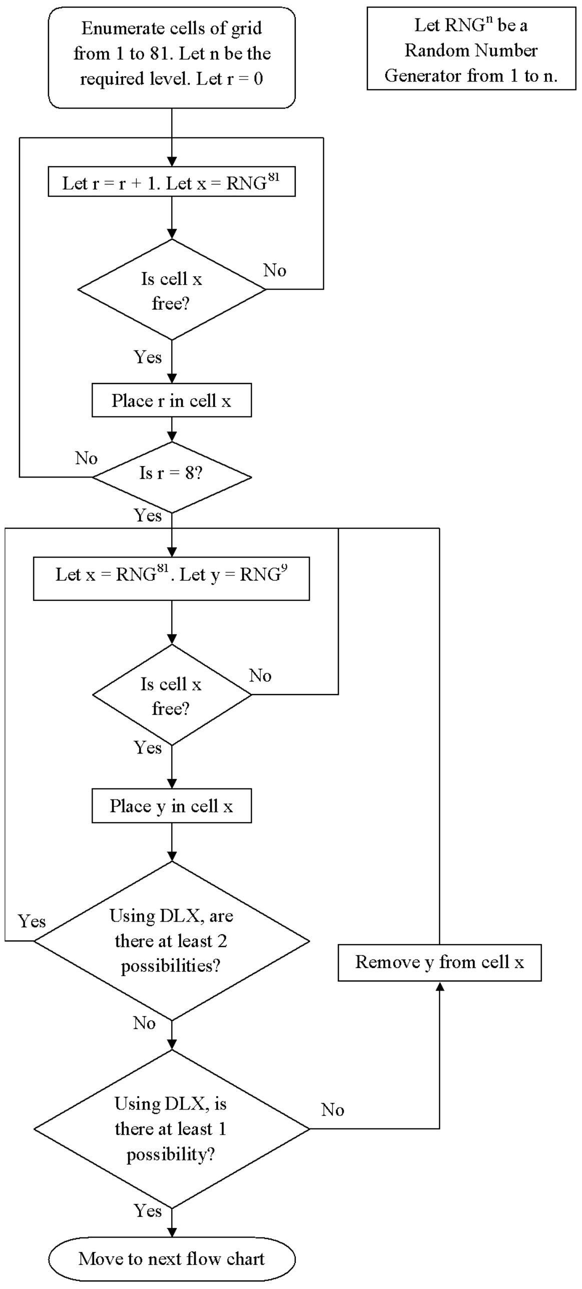

Enumerate cells of grid from 1 to 81. Let n be the required level. Let r = 0
Let RNG^n be a Random Number Generator from 1 to n.
Let r = r + 1. Let x = RNG^81
Is cell x free?
No
Yes
Place r in cell x
Is r = 8?
No
Yes
Let x = RNG^81. Let y = RNG^9
Is cell x free?
No
Yes
Place y in cell x
Using DLX, are there at least 2 possibilities?
Yes
No
Remove y from cell x
Using DLX, is there at least 1 possibility?
No
Yes
Move to next flow chart

Figure 13: The first flow chart describing our method

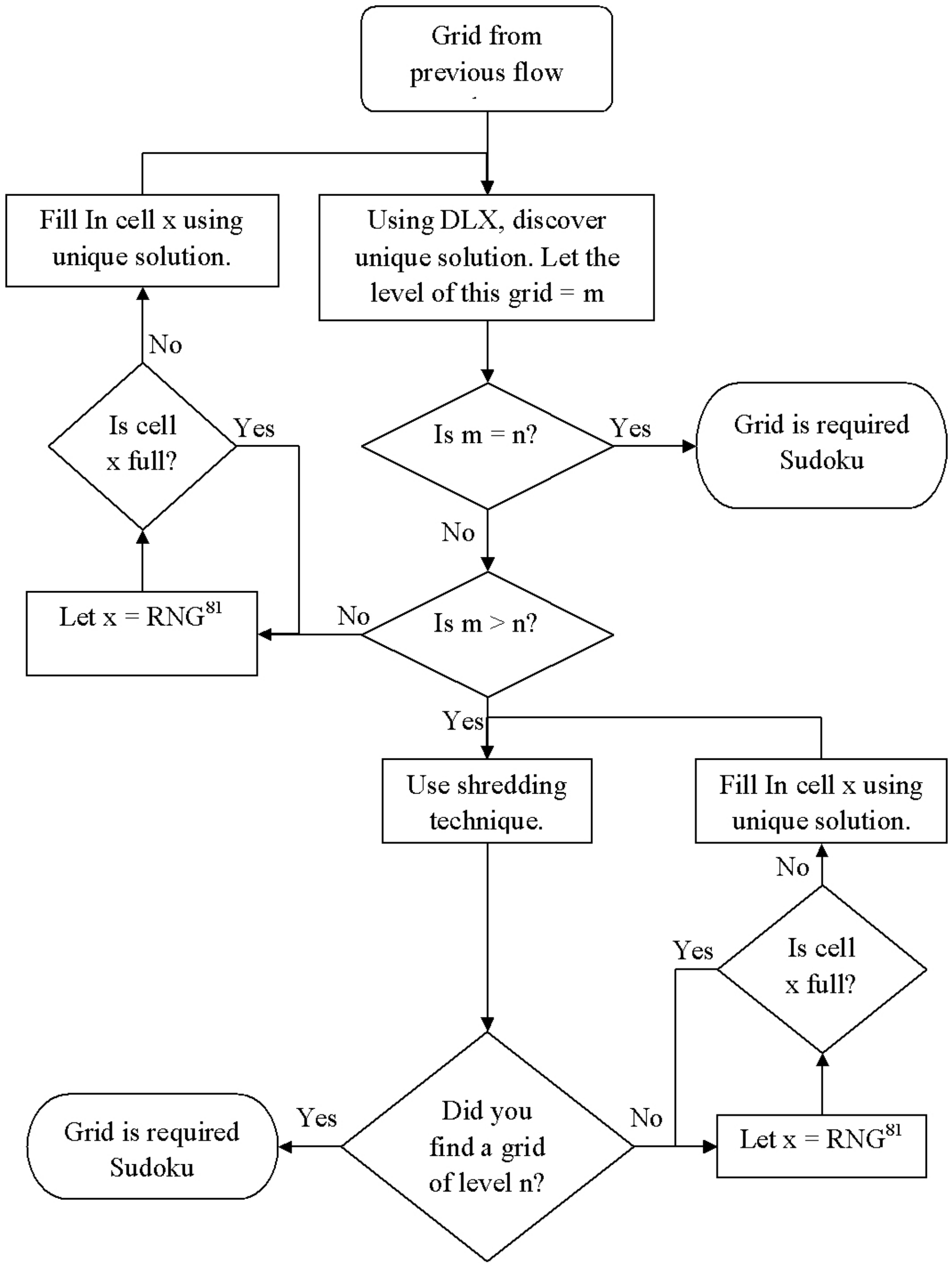


Figure 14: The second flow chart describing our method