Algorithmics of Sudoku Project

Student Name: L. Entwistle

Supervisor Name: A.Krokhin

Submitted as part of the degree of BSc Computer Science to the

Board of Examiners in the School of Engineering and Computing Sciences, Durham University

29/04/2011

***Abstract* — Background**: Since the surge in popularity of Sudoku in recent years, there has been broad academic interest and computational research into the fundamental problem of solving Sudoku puzzles. Many complex logical techniques have been discovered which can be used as tools in the solving process.

Aims: To investigate the algorithmic properties of these solving techniques. Also explored is the challenge of creating puzzles with attractive properties, such as having a single solution and having no redundant clues. Finally, to investigate how to assess the difficulty of a puzzle for a human to solve based on some metric.

Method: I produced a software framework for manipulating Sudoku puzzles and built programs to use it to solve puzzles, assess puzzle difficulty and to generate puzzles from scratch. Thoughout this process, many areas of Computer Science were touched upon, such as constraint programming, the Exact Cover problem, computational complexity and randomised algorithms.

Results: Robust, functionally complete programs were produced to achieve the various objectives, and all have been measured, analysed and judged to be successful overall.

Conclusions: Sudoku puzzles are flexible enough to be good test subjects for a wide array of computational and mathematical ideas, and excel at highlighting the relative strengths and weaknesses of different approaches to the same problem.

***Keywords* —** Sudoku, puzzle, constraint programming, solving, difficulty rating, generating, exact cover, randomised algorithms

I INTRODUCTION

Sudoku puzzles have had increased public exposure recently due to their popularity in newspapers, which has bought with it an increased level of academic interest across the field of computer science because they act as a very suitable example for many algorithmic techniques and paradigms. The formulation and rules of Sudoku puzzles are simple: a puzzle consists of a 9×9 square grid, partially filled with the numbers one to nine, and it is the goal of the player to fill in the remaining grid completely subject to the following rules: A full grid must have exactly one of each number in each row, column, and 3×3 block of squares, and these rules allow a puzzler to find clues as to the location of each number from the initial puzzle grid.

The core question this project strives to answer is "How far can computational techniques be applied to Sudoku-related tasks?" which I have answered by developing a puzzle solver, a puzzle difficulty assessor and a puzzle generator. During my investigations I tried to always maintain a good level of usability in the programs, and ensured that the software relates well to the requirements of human users. This goal was particularly important in the domain of Sudoku puzzles, since people who have been struggling on a puzzle for a long time are usually less interested in *what* the final solution is and more on *how* the solution is reached, so would usually

prefer a sophisticated solver which explains the steps involved. Moreover, it is no good having a difficulty rating system if the ratings have no bearing on a person's likelihood of completion, and a puzzle generator can be considered all the more successful if it produces puzzles of a certain style or difficulty.

The scope of this project is limited to just looking at the 9×9 grids, or standard Sudoku puzzles, since this will be most relevant to users; working with any smaller grids is verging on the trivial, and any larger risks providing instances which would run for an unfeasibly amount of time when we consider the computational complexity of some of the algorithms involved. Furthermore, the software was written in Java because of the advantages gained from working with such a familiar, Object-Oriented language.

A Terminology

I will now list the terms I will use in this report, as the language to describe Sudoku puzzles varies in the literature. Firstly, a completed Sudoku grid that conforms to the rules is a *solution*, while any partially-filled grid is a *puzzle*. A *well posed* puzzle has exactly one solution; this is important, since any arbitrary puzzle might have large number of solutions or even none at all. The squares and numbers already present in a Sudoku puzzle before solving starts are the *givens*. A puzzle can be described as being *minimal* if there are no redundant givens, that is, you cannot remove one of the givens and still have a well posed puzzle. There are eighty-one *Squares* in a Sudoku grid, each labelled with its co-ordinates in the grid, so the top line contains Square(1,1) on the left, followed by Square(1,2) moving right until Square(1,9). Each square has a set of *candidate* numbers $\subseteq \{1, 2, \ldots 9\}$ which describe the possible numbers that can be assigned to it. Each horizontal line of squares is a *row*, labelled from top to bottom and each vertical line is a *column*, E.G: row 1 is at the top edge, column 9 is the right edge. The groups of 3×3 squares are called *blocks*, and are labelled according to their position in the grid. Row columns and blocks are all considered *scopes* of squares, and are illustrated in Fig. 1.

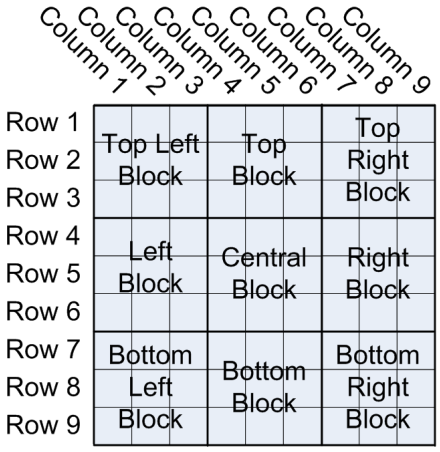

Figure 1: The scopes in a Sudoku grid

B Deliverables

Each deliverable for this project takes the form of a piece of software, themed around Sudoku puzzles and their solutions, and each draws from a variety of algorithms and techniques. Moreover, they will often rely on the functionality of each other, and so they should be prepared in such a way that they can be silently used by other programs (that is, with no user facing front end) The main deliverables in this project are:

- **Sudoku Solver**. Much existing software like this deliverable already exists since the problem of finding a solution from a puzzle grid is a far from trivial one, so there is much interest from researchers of a computational or games background. This component builds upon previous academic work, and components from it have become critical for the subsequent program deliverables. My solver has a suitable UI to help in explaining to the user why a particular technique is relevant and should be applied to the current puzzle state, which allows it to act as a helper application for users to use alongside their recreational Sudoku puzzle solving.
  By phrasing the problem in a variety of ways, the solver can discover many advanced logical techniques remarkably efficiently. The main examples of this are the concept of constraint programming, for which Sudoku puzzles are ideal candidates, and finding contradictions by reducing the problem of solving puzzles to Exact Cover and the optimisations we can achieve though that. Beyond finding logical solving methods, if a particular Sudoku is extremely hard and would take too long to search through from the user's point of view, or if the particular puzzle uses a more advanced technique than those implemented then there is the option to brute force the solution using trial and error; this entails making guesses about a square's value, and following the consequences until a contradiction is found, after which it backtracks and tries again until a non-contradictory, complete solution is found. This was included in the solver mainly as a last resort against tough puzzles, but this functionality became most useful for the Sudoku generator (as seen below).

- **Sudoku Assessor**. This is my name for the program designed to rate the difficulty of Sudoku puzzles relative to each other. Given a puzzle, this module will analyse the path to the solution and gather certain properties such as the techniques used. Using this data as a guide, the program can look up against pre-computed factors how each property will affect the average time for a human user to find a solution, and it will terminate by producing an estimate as to the number of minutes finding a solution will take. I aimed to have this metric as accurate as possible to reflect the difficulty a reasonably competent human player would have in solving the puzzle by hand, and uses the 'minimal path' to the solution, IE: the sequence of actions from initial state to the solution state, using the easiest techniques available. This is why this deliverable borrows heavily from the Sudoku Solver, which effectively can be used to simulate the actions a human would do. Gathering the data and producing this metric has been very interesting, and the results of such experimentation will be analysed in the Results section.

- **Sudoku Generator**. This program is effectively the inverse of the Solver; given a Sudoku solution grid as input, it can produce a well posed, minimal puzzle. The solution itself is randomly generated, and uses a step by step process to produce the final puzzle by working backwards, removing squares until a suitable output is produced. Various optimisations

can be found here, to improve the speed and quality of the output puzzle. It also builds upon the work of the Brute Forcer and Assessor programs, particularly because there is an option to produce a puzzle of a set difficulty.

II RELATED WORK

The problem of solving a Sudoku puzzle is NP-Complete (Takayuki 2003), due to the fact that it is at least as hard as solving Latin Squares. Latin Squares have the same grid and configuration as Sudoku puzzles, but it is sufficient just to place the numbers uniquely in each row or column rather than including blocks, so every Sudoku puzzle is an instance of a Latin Square. (Colbourn et al. 1984) showed that Latin Squares are equivalent to the '1-in-4 SAT' (same as standard 4SAT, but a valid truth assignment must have exactly one true literal per clause), which itself is NP-Complete because 3SAT can be reduced to '1-in-3SAT' which itself is reducible to '1-in-4 SAT'.

The factors in producing a valid Sudoku grid are strictly finite: the size of the grid and the numbers that can be assigned to it, so it follows that the total number of possible valid Sudoku solutions is finite too. Calculating this number is far from trivial, but has been found to be $\approx 6.671 \times 10^{21}$ (Felgenhauer & Jarvis 2005). However, many of these Sudoku grids are equivalent to each other, for example, simply by swapping all the number 1s and 2s. A more pressing question is to find the number of *unique* solution grids, that is, the size of a set of grids in which no grid can be 'transformed' into any other. These transformations have been identified by multiple sources (and were used in my Generator) and can be applied to create up to $\approx 1.2 \times 10^{12}$ different but equivalent grids. The 4 types of transformations generally recognised are (Russell & Jarvis 2006): permutating the numbers 1 to 9, permutating the stacks/bands, permutating the rows/columns within a band/stack and lastly transposing the grid (which is equivalent to rotating, flipping, reflecting the grid in combination with the other transformations).

Considering minimal puzzles where there are no redundant given clues, it is conjectured that the minimum number of givens is 17, but the evidence is heavily supportive that this is a mathematical property of 9×9 sized Sudokus. (Royle 2010) has compiled a growing collection of 49151 distinct and nonequivalent minimal Sudoku puzzles of size 17, while not once has a puzzle with 16 or fewer given squares been found to be well posed. Unsurprisingly, this lower bound falls away sharply with Sudoku variants; adding additional constraints results in more possible implicit clues. For example, just adding the rule that the two long diagonals must be assigned the number 1 to 9 uniquely reduces the minimal puzzle size to 12. Similarly, the conjunctured upper bound for a minimal Sudoku puzzle is 39, since no minimal puzzle has been found to contain 40 or more given clues.

A powerful solver idea employed by my Solver is that of modelling Sudokus as instances of Constraint Programming (Simonis 2005). This paradigm is used in general to assign a set of values to a set of variables, the relationship between assignments being constrained by certain rules. A commonly used example for this is the Nine Queens problem (Dechter 2003): you must place nine queens on a chessboard such that no queen is in the same row, column or diagonal. In this example, the values are the 64 squares on the chessboard, the variables are the 9 queen pieces (eg: in a set Q), and an assignment is putting a piece on the board. From an assignment a player can deduce the row, column and diagonal of a piece (eg: using functions row, col and $diag$ respectfully), and so the constraints in this example ensure that $\forall_{q_1,q_2 \in Q}[(row(q_1) \neq row(q_2)) \wedge (col(q_1) \neq col(q_2)) \wedge diag(q_1) \neq diag(q_2))]$. (Simonis 2005) applied this to Sudokus

successfully, intentionally focusing on trying to find solutions to well posed problems logically with minimal guesswork. As well as modelling the explicitly stated rules of Sudoku, he also added redundant constraints in order to improve performance. Such ideas were built into my Solver and shall be explored later in this paper.

(Reeson et al. 2007) have implemented an impressively transparent Constraints-based Sudoku solver in the form of an online Java applet. They implement two representational models based on constraints that ensure unique assignments: one includes 810 'binary' disequality constraints working on a pair of squares (each square will be a member of exactly 20 constraint pairs with each of the other squares in its row, column and block), and the other with 27 'all-different' constraints working over 9 different squares each (one constraint for each row, column or block). The applet is easy to use and understand, particularly with reference to applying particular CP actions to the puzzle grid. This work demonstrates the feasibility of such methods by the use of sufficient application of constraints to solve many hard puzzle quickly.

Beyond constraint programming for a solver, related academic work has had one of two problems: either they are too similar to the brute force approach in that they find a solutions quickly, but without details about how a human can copy the solving process, or the solver has made use of complex and exotic solving techniques (Sudopedia 2011) found by the Sudoku research community and forums which would be inappropriate for me to attempt to recreate. (Green II 2009) provides a good recent summary of AI heuristics on this subject, but "most fall short as they do not solve the problem efficiently", while Constraint Programming and similar ideas proved effective in general. Covered techniques include genetic algorithms, particle swarms and bee colony optimisation, neural networks and simulated annealing, and a common problem with such approaches seems to be that the execution time has an unacceptably large standard deviation, and often some puzzle are found not to be solvable at all.

It has been noted that the difficulty of rating a Sudoku puzzle is harder conceptually than either the solving or generating processes (Fowler 2008). (Simonis 2005) mentions how Sudoku puzzles of a described difficulty often, but not always, match up with a certain level of propagation scheme when tackled with constraint programming. (Leone & Vaswani 2010) use human solving time data in order to produce as accurate a metric as possible using individual solving techniques as variables in a formula created through 'bootstrapping' techniques, and such an approach was recreated by my own Assessor module.

Considering the generation of Sudoku puzzles, many different techniques have been identified. (Leone & Vaswani 2010) suggest three kinds of generators: firstly, one based on a large library of Sudoku puzzles uses a hash table to retrieve puzzles of a given difficulty, before transforming it into a new puzzle. This technique, while running in O(1) time, hardly fits with my deliverable definition of a Generator and is more a method of ensuring differentiation based on prior work, but such a technique was still found to be useful in my Generator. Their second technique is to construct a puzzle from a pre-generated complete solution grid ('decremental' generation). Producing a seed solution was done quickly using Latin Squares, and then applying an 'intelligent' brute force method to randomly remove givens, testing for uniqueness and difficulty as it goes until an appropriate puzzle is found, else the algorithm backtracks, and again this is an idea I have emulated in my own Generator. The author's final contribution is that of a template-based generator, similar to the previous in that it works down from a complete solution, but only in a pre-described template shape (ie: to ensure an attractive puzzle). However, compared to the other algorithms, this has a significantly higher complexity and running time,

and experimentally has a heavy bias towards producing easy puzzles. Other authors have employed an 'incremental' based approach consisting of assigning random numbers to squares on an initially empty grid, applying a puzzle solver after each step to test it can be solved search-free and for uniqueness (Weller 2008). This method requires backtracking if an inconsistent grid is produced, and may have to start again entirely.

III SOLUTION

The produced software has a very "Model View Controller"-style overall structure, which means that there are very clear logical divisions between classes that represent what the user sees (view), the algorithmic core (controller) and the shared data structures (model). This made designing the system clear and easier to produce code. The main computational classes extend a Module class as a template, each of which has a GUI focused class associated with it. The core to each project deliverable is represented in these modules, namely the AutoSolver, Assessor and Generator modules, but other modules have been added for convenience, including the creator for graphically creating puzzles instead of loading them from a file or pasting an encoded string, and a UserSolver modules for users to make changes towards a solution themselves without automated help. Individual Modules can work on data structures directly through method calls, the changes of which are propagated to other related controllers and views by employing the Observable/Observer pattern; observer objects can register their interest in Observable objects by adding a listener to it; for example, the user might load a puzzle, solve it partially in the UserSolver module, and then open the AutoSolver module; this can find and apply moves, and have those changes automatically updated in the original UserSolver module.

Usually, key algorithms have been extracted and encapsulated into their own objects, particularly if they do work to some internal data structure or can be used by multiple modules. The main examples of this are the objects which represent path consistency which is part of constraint programming, and also the "Exact Cover" data structure, which is actually used by all three main deliverables.

A Constraint Programming Tool

Constraint programming is a flexible algorithmic tool which can be applied to a wide range of problems, but Sudokus especially fit into such a concept very naturally. Each square in the Sudoku grid can be considered to be a variable, and the numbers 1 to 9 are the values that can be assigned to them. The main constraint employed in my solver is one that ensures that a given pair of squares cannot be assigned the same number, namely those pairs of squares which share a common scope (that is, a common row, column or block). This is analogous to colouring a graph with 9 colours, with each square being a node, and adding an edge between squares with a common scope; these edges represent the constraints here. Note that this is an example of a binary constraint, it is also possible in general to have unary constraints acting on just one variable or others that apply the same rule to multiple variables at once. For this project, the set of variables is thus $\{Sq(1,1), Sq(1,2)...Sq(1,9), Sq(2,1), Sq(2,2), ..., Sq(9,9)\}$, and the value set is $\{1, 2, ..., 9\}$.

Formally, all the possible values of a variable x_i can take can be represented by a domain D_i, and all constraints can be a represented as a relation between the two domains: the relation R_{ij} is a subset of all the valid pairs of values that can be assigned between the variables x_i and x_j, where

$(a_i, a_j) \in R_{ij}$ if the assignments a_i and a_j are allowed to be assigned simultaneously. Using this notation, we can apply known constraint propagation techniques to the model to reduce the domain of possible values for each variable. This is equivalent to finding candidate numbers to remove from the squares in our Sudoku puzzle. The simplest way to propagate constraints in our case is using Arc-Consistency, named because it makes sure that only valid arcs or edges remain between assignments with respect to the constraint relations. This means that considering the values x_i and x_j if there is a value $a_i \in D_i$ such that $\forall_{a_j \in D_j}[(a_i, a_j) \notin R_{ij}]$ then there is no assignment in x_j that is possible by assigning $x_i \leftarrow a_i$, so we can remove a_i from D_i since it would make the model inconsistent. In terms of a Sudoku puzzle, this is like assigning a square the number 4, and then removing the number 4 from all the other squares in that squares scopes.

Path-Consistency is taking the concept of Arc-Con to the next level; instead of looking at one assignment, it looks at a pair of assignments, and checks to see if a third square within the scope of the pair can be assigned a value that is consistent with both assignments; say we are looking at the assignment pair $(a_i, a_j) \in R_{ij}$, then we need to find a triangle of relations such that $(a_i, a_k) \in R_{ik}$ and $(a_j, a_k) \in R_{jk}$ where a_i, a_j and a_k are values in the domain of the variables x_i, x_j and x_k respectively. If no such a_k exists, then the pair (a_i, a_j) should be removed from R_{ij}. This continues to remove pairs until we reach an arc-inconsistent state in which a value a_i is not included in any of the pairs in the relation R_{ij}, (as before, $\forall_{a_j \in D_j}[(a_i, a_j) \notin R_{ij}]$), so we can remove that value assignment completely. Applying this idea with Sudoku puzzles, the target candidate in a square that we want to remove forms half of each pair we test, with the other assignment coming from another square within the target's square's scope. If each pair of assignments leads to a third square having all its candidates removed, (IE: because the third square only has candidates from the currently assumed pair), then we can mark that pair as invalid, and if we can mark all the pairs between the target assignment and all the assignments in the second square, then we know that setting the target assignment will always lead to a contradiction no matter what we assign the second square.

Using Constraint Programming, we can add to the traditional techniques used in the Solver. Implementing Arc-Con in my project was fairly easy; it didn't need any new data structures, and was applied whenever a value was set to a square. Arc-Con just scans through each Square in each of the SquareScope objects, and if that square has as a candidate the value we just set then the technique produces an action which says this candidate can be removed.Because we are already tracking the scopes, and because we only run it when a value is set, the running time for this technique never becomes an issue.

Path-Consistency is notably harder however, and requires additional data structures to work efficiently. I needed to produce an Assignment object, each of which represented a square:number mapping, and an AssignmentPair object to define the relations. I implemented the optimum algorithm $PC - 4$ as described by (Mohr & Henderson 1986, Han & Lee 1988), which runs in $O(n^3k^3)$ where n is the number of variables and k is the size of the domain; it is clearly impossible to achieve a better upper bound, because there are nk assignments, and we must consider three assignments simultaneously to test for consistency, hence $O(nk \times nk \times nk) = o(n^3k^3)$. This algorithm first constructs the relations and pairs, with the idea that each assignment pair is supported by a number of assignments per square, meaning that an assignment supports a pair if all three assignments can be set simultaneously without breaking any constraints; if the third supporting assignment is removed, then the support for that pair from that square is reduced, and if it ever reaches zero then we mark that pairing as invalid. Also, each Assignment object tracks

its incident pairs (that is, that assignment is part of that pair), and if all the incident pairs with another square are removed, then this assignment is removed too. This makes the running and updating of the data structure reasonably efficient once initialised, but it is that first initialisation which takes a lot of time to complete despite being the most efficient algorithm; constructing the data structures for a standard application usually takes 1 to 2 second to initialise and ~40MB of space on the Java VM heap, while for the worst case scenario (a completely empty grid), initialisation takes at least 6 seconds and ~120MB space. Both Arc and Path-Consistency are part of the same abstract consistency concept; Arc-Consistency is equivalent to 2-Consistency (named because it compares two variables), while Path-Consistency is equivalent to 3-Consistency, and this can be extended arbitrarily to i-Consistency by considering sets of $(i-1)$-variable assignments simultaneously and finding a contradicting i^{th} variable. Enforcing each level of consistency takes at a minimum $\Omega(n^i k^i)$ time and space (Dechter 2003, p69), and for this reason pursuing i-Consistency deeper wasn't considered useful to the solver program; it would take exponentially more time and space to initialise and run, and each levels cannot be reused to produce the next level in any meaningfully efficient way, so it would have to effectively rebuild from scratch to prove consistency at the next level. Some of the super advanced puzzles found in the online Sudoku community require techniques that think about at least 10 squares simultaneously, and ensuring the consistency at this depth and beyond would have vast time and memory requirements. This is not a problem however; we can apply other tools such as the Exact Cover data structure below to do the job of solving techniques lost to the infeasibility of i-consistency.

B Exact Cover with 'Dancing Links' Tool

The problem of filling in a Sudoku puzzle can be posed as an instance of the Exact Cover problem, which allows us to take advantage of the data structures commonly used in solving such a problem. Exact Cover can be described as so: You are given a set X and a collection of subsets of X called $S = \{x_1, x_2 ... x_n\}$, and we want to find an 'exact cover', which is a sub-collection of S such that every element of X is contained in exactly one chosen subset. This is commonly represented as a binary matrix, where each column is an element of X, each row is a subset from S, and if a particular element from X is in a particular subset, then a 1 is placed in the intersection of the corresponding row and column.

It is clear to see how the Exact Cover problem is a form of constraint satisfaction; each element of X can represent an individual constraint, and the subsets in S are the various options that can be chosen to satisfy them. To 'solve' the matrix, we must select some of the options to satisfy all constraints (and in fact Exact Cover goes further by requiring that each constraint is satisfied exactly once). In the case of Sudoku puzzles, the options are which number is assigned to which square, and the constraints are the rules of the puzzle. Thus, we can formulate a matrix with each row representing a square assignment (81 squares $\times$ 9 candidates=729 rows in the matrix), represented as a string, for example assigning the number 4 to the square at co-ordinates (8,2) would be the string "Sq(8,2):4". Each column would be represented by one of the following constraints:

- Each scope must contain the numbers 1 to 9, where a scope is considered to be a row, column or block in the Sudoku puzzle. (27 scopes in standard puzzle $\times$ 9 numbers each = 243 columns added to matrix). For example, the column in the matrix that represents the constraint "There must be a number 1 in row 5" will have a 1 in each row representing

"Sq(5,1):1", "Square(5,2):1", "Square(5,3):1" ... "Sq(5,9):1" since each of these squares is in the row 5 scope.

- Each square must be assigned a number (adds 81 columns to the matrix, one for each square). Since this constraint applies to any candidate number, the matrix will contain 1's on any assignment on that square. For example, "Square(1,1) must be assigned" would be "Sq(1,1):1", "Sq(1,1):2" ... "Square(1,1):9".

This setup results in each square assignment being represented as a subset of the constraints that become satisfied by applying the assignment to the puzzle. For example, "Sq(3,4):5" would satisfy the constraints that "Square(3,4) must be assigned", "Row 3 must have a 5", "Column 4 must have a 5" and "The Top Block must have a 5".

Finding an Exact Cover can be done using Algorithm X, described predominantly by D. Knuth, and which is a natural trial and error process. It works like so: pick a column (constraint) by some metric, and then randomly pick a row with a 1 (an assignment that would satisfy that constraint). Assume that row is part of the solution, and when you do, delete all of the columns in it with 1s from A (these are satisfied constraints which don't need to be considered anymore), and also delete all the rows in those columns (rows that have been made impossible to pick now the one of its constraints is satisfied). Repeating this process will eventually lead to one of two conclusions: Either A is empty, in which case we have found the solution because all constraints have been satisfied, or a column will become filled with all zeroes; this means that it is impossible for this constraint to be satisfied from the given assumptions, so we backtrack by undoing the assumption that the randomly chosen row is in the solution and choosing another valid row. If we have tried all possible assumptions, then there is no solution for this instance.

Knuth uses this algorithm in a paper on his 'Dancing Links' technique, which focused on the fact that a removed element from a doubly linked list can be reinserted in constant time by using the left over pointers in the removed object. For example, object o has pointers to its left ($o.left$) and right ($o.right$) elements in the list. To remove o, simply redirect the pointers of it's neighbours:
$o.left.right \leftarrow o.right$
$o.right.left \leftarrow o.left$
and to reinsert o:
$o.left.right \leftarrow o$
$o.right.left \leftarrow o$
This simple observation means that we can add or remove elements from a doubly linked list as quickly asymptotically as we can flip a bit in the matrix in Exact Cover. This means that we can take advantage of the linked list by replacing the matrix with a grid of doubly linked lists made of 'Node' objects, with each node representing a 1 in the original matrix, with pointers to the next object above, below, left and right of it. This means that the process of finding the next 1 in the matrix in any direction is reduced from running in $O(n)$ time to $O(1)$, and since this is heavily used in Algorithm X, the speed up possible was worth implementing in the project. Further speed up is possible by choosing the correct columns to explore first; if we set the heuristic to choose columns with the lowest number of possible assignments that can cover it, if we arrive at a contradiction we can cover more of the complete search tree faster. Moreover, a naive implementation would take $\Theta(n \times m)$ time (where n is the number of columns and m is the number of rows) to count up all the nodes in each column, while my data structure has a header

node on each column which tracks the number of nodes below it, and also explicitly shows if a column is in or out of the data structure. This counter gets updated every time a node in the column is added or removed from the structure. This means that finding the minimal column is just a matter of visiting each header node, reducing the complexity to $\Theta(n)$.

The obvious application for this data structure and algorithm is in finding a solution to a given Sudoku puzzle, and so this provides the brute-force functionality to the software framework. This tool can check to see how many solutions a Sudoku puzzle has by running algorithm X on the Dancing Links enhanced data structure; if it visits all the assignments in the first chosen column without finding a solution, then the puzzle is unsolvable. Otherwise, it will try and find two solutions, if it only finds one then the puzzle is well posed, otherwise it has multiple solutions. Using the brute force feature in the Solver, the user can indicate that they want to continue finding valid solutions, and the brute forcer will continue with algorithm X and display all the puzzles it finds and cancel if it takes a long time (which it probably will do; often the number of valid solutions runs into the thousands on a badly posed puzzle).

Another incredibly useful application for this tool is in quickly finding testing assumptions and finding contradictions, which is used for when all standard solving techniques have failed to find a clue and before we have to resort to brute force. This tool works by running a modified algorithm X in which each possible assignment is chosen in turn and assumed to be correct, before running the algorithm as standard but only covering columns with only one node; these represent constraints where there is only one possible assignment that will satisfy it, and so is directly implied from our initial assumption. This continues until either all columns have at least two nodes (in which case we can't prove anything, so undo all changes and continue with the next assumption) or we find a column with a node count of zero. No nodes in a column indicates a contradiction, since we cannot satisfy that column, thus we can deduce that the initial assumption must be incorrect and the assignment must be wrong, allowing us to remove that assignments number from that assignments square. The reverse also works; we can assume that a square IS NOT assigned a particular number by removing the row representing that assignment from all of its columns, effectively stopping it from being in the final solution, and this might force another square to be assigned a value and then the implications continue as normal (for example, if the number 4 can only go in two squares in a block, then assuming 4 can't be in one square forces it to be assigned to the other). I considered it to be more powerful than the other kind of assumption, because the final effect is setting a value rather than removing a candidate, and it might take many applications of finding contradictions that leads to candidate removal to be equivalent to one application that sets that value. The Contradiction Finder generates an Action object when it finds a good contradiction, which can be used by the solver to show the chain of implications used to reach the contradiction. The final contradictions can either be in the form of "all candidates have been eliminated from a square X" or "There is no place for the number N in the scope S" because they are found directly from the columns which represent these rules. With respect to the standard solving techniques, arc consistency is automatically applied by covering columns in the data structure, and all the implications are just applications of hidden single and naked singles; these will be explained below.

C Solver Module

The aim of the solver module is to show clearly the steps involved in solving a Sudoku puzzle from start to finish. Any changes that the Solver suggests are in the form of Action objects, which

are interpreted and displayed graphically, and the kind of Action you get depends on from which SolverTechnique object it originated. SolverTechnique is an abstract class which is inherited by more specific classes which fill in the technique details, but they all share the following features: each contains its own queue of actions, and whenever the user requests an action, it is popped from this queue. Each technique directly manages a portion of the GUI in the form of a button to display the next action and a check box which is used to activate automatic mode for technique, in which the queue is automatically emptied and all the actions found are applied to the puzzle as soon as they are found.

I have implemented five kinds of Sudoku solving technique styles, but really these are abstractions of techniques found in the wild and cover many real-world moves (mainly due to the fact that all scopes are treated identically: rows columns and blocks are all just HashSets of Squares). In fact, looking at one method list from the online Sudoku community (Sudopedia 2011), these five implementations in the program actually cover the fourteen most commonly used methods by human solvers, and any harder moves are covered by the Contradiction Finder and Path-Consistency. In order of easiest to apply, these techniques are:

- Hidden Single: There is only one square left which contains the candidate number n in a particular scope. The result is we can assign n to that square. A surprisingly high number of published Sudoku puzzles can be solved using this technique alone, yet it is the simplest to find most of the time.

- Naked Single: There is only one candidate number remaining in a particular square. Result: Assign the last candidate to the square. While this technique may seem easier than finding hidden singles for a computer, in practice many puzzlers consider it harder because they are not automatically tracking the remaining candidates in every square, and time may be needed before an opportunity for naked singles is recognised.

- Intersections: This involves a number n and two overlapping scopes, A and B. Consider the subset of squares with n as a candidate in A and B (EG: $A_n = \{sq : \ sq \text{ has the candidate } n\} \subset A$. Similar definition for B_n), then if $A_n \subseteq A \cap B$, then we can remove the candidate n from all squares in $B_n \setminus A$. In words, this means that if all the squares with candidate n in A are also a member of another scope B, then squares with candidate n other than these squares should have the n removed as a candidate, because we know that the n in B MUST be in one of the intersection squares in order to satisfy scope A. This might sound long winded, but applications of intersection are relatively easy to spot by looking at what assignments are 'covering' the squares in a certain scope and looking for lined up candidate squares. These intersection come in a few flavours, depending on if the scopes involved are rows, columns or blocks.

- Hidden subset: This technique comes in three varieties: pairs, triplets and quadruplets, and works exclusively inside one scope. This technique extends the idea of hidden singles into sets of more squares. Consider the set of squares with a 1 as a candidate. If there is only one such square in the set, then we know that square must be assigned 1. Similarly, if we consider the numbers 1 and 2 and the union of the set of squares with those candidates, then if there are only two squares in the union then we know that each of these squares must be assigned either a 1 or a 2. In this case, we can remove all the other numbers 3 to 9 from those two squares. The same logic applies to sets of numbers and sets of squares of size

3 or 4. Not all of the squares need to have all the numbers as candidates, but the program tries to look only at cases where each square has at least two of the chosen numbers as candidates, otherwise a smaller hidden subset can be found by removing the square(s) that break the rule.

- Naked subset: Also comes in pairs, triplets and quadruplets. A naked subset is a set of squares from a scope such that the union of all of the potential candidate sets of these squares is the same size as the square subset itself; from this, we know that the numbers set of size 2, 3 or 4 MUST be assigned to these squares, otherwise we get a contradiction. This lets us remove all of the numbers in the candidate set from the other squares in the scope. Again, each square need not have exactly the same candidate set, so long as the union of the sets equals the number of the squares used to generate it.
  This is pretty much the opposite of hidden subsets, and in fact this is the reason why we don't need to search for subsets of size greater than 4: because in each scope with a naked subset of size ≥ 5, there must also be present a hidden subset of size ≤ 4 (because there are only a maximum of 9 unassigned squares in a scope), and vice versa, and we can use the smaller subset to remove the same candidates.

Implementing these techniques doesn't require any additional data structures, and applications of the techniques are found as the Sudoku puzzle is updated appropriately. The number of squares which have a particular candidate is actually tracked in the SquareScope objects, so hidden singles runs in almost constant time. Naked singles are similarly trivial to find since the candidate count is tracked by the individual Square objects. Finding instances of intersection is done by looking at the remaining squares with a candidate when it is removed from a square in a particular scope and seeing if these squares share a different common scope. Both naked and hidden subsets can be found in a very symmetrical fashion: the hidden subset technique looks at each subset of possible candidate numbers in a scope and looks for a union of common squares in the scope of equal size, while naked subset looks at each subset of the unassigned squares from a scope and looks for a union of their candidates of equal size. Generating the subsets in both cases uses the class SubsetGenerator which takes a seed list of any objects and returns the subsets in a provided range of sizes in lexicographical order by implementing the Iterator interface.

When a technique finds a logical way to improve the Sudoku puzzle towards the solution, it creates an Action object. These can handle both setting values and removing candidates, and can act on one or many targets depending on the kind of technique. Each Action has a set of source squares, and an explanation String to explain the process. There is also a special class of actions that work like logical chains such as the Contradiction Finder or Path-Consistency, which store the sources as one or many lists of assignments, in which one assignment directly follows the previous one towards a contradiction of some sort. In the GUI portion of the Solver, clicking a technique's display button gets the Action, interprets and displays the result graphically with coloured squares to highlight the critical squares to the user. The explanation text is also shown with coloured words to link the two displays together easily to aid user understanding. From here, the user can click 'Apply' to update the SudokuPuzzle object with the consequences of this Action, or they can 'Apply all' to have all the actions queued for this technique to be applied instantly.

D Assessor Module

The assessor module is designed to provide an accurate measurement of how difficult a human user will find a particular Sudoku puzzle by producing an estimated average completion time and a descriptive difficulty, either Gentle, Moderate, Tough or Diabolical. A very high level overview of the algorithm involved is that the assessor module controls the actions of the Solver, deciding which move to apply, and it gathers data about the particular puzzle as it goes. Once solved, this data is fed into a precomputed statistical function to return an estimate of the amount of time the puzzle will take for a human solver, which is then compared to set boundaries to determine the categorised descriptive difficulty.

Another feature of the Assessor is to find and display the hardest moves used to reach the solution, which is provided to the user as an explanation of sorts as to why the puzzle is a particular difficulty. It does this simply by rerunning the list of moves created in the assessing algorithm, weighing each move based on how much time it contributes to the final solution and saving the snapshot of the puzzle just before it is applied, and displaying the top five largest contributions at the end.

Creating the actual function which returns a time took a process inspired by the work of (Leone & Vaswani 2010). I created a program to scrape the archives at Sudoku.org.UK for data to work with, which resulted in 4 years worth of daily puzzles and solutions (1461 in fact) complete with statistics like average solve time, number of people who completed it and the described difficulty. Using these puzzles, the Assessor was used to find the following property information from each, with the general idea that higher numbers equate to a harder puzzle:

- A count of the number of times a particular technique is used. If the technique can have many 'levels', these are counted separately, eg: hidden subsets coming in sets of two, three or four, and contradictions might include any number of implications.
- The number of source squares used by an action. The more complex the action, the more sources it typically requires.
- The geometric distance between the action sources to the target(s). Relationships between the source squares and the targets are harder to spot if they are further apart.
- The geometric distance from the previously applied action. Usually a human player will look in the immediate vicinity following a successful action to see how this has changed the puzzle.
- How long ago we last used this technique. Longer gaps lead actions being relatively harder to spot. If a human solver has seen many applications of hidden pairs, they are more likely to find them in future.
- The search space, or, how many unsolved squares remain in the puzzle. A measure of the size of the area the player needs to look at - smaller areas increases the likelihood of finding an action.

What resulted is that we had a large source of data relating the average solve time of a puzzle to its properties, which can be used to create a multiple linear regression model which takes the

input properties and produces an average solve time. The form of such a model is simple:

$$m_0 + \sum_{i=1}^{n} m_i p_i$$

Where there are n properties from the Sudoku being analysed ($p_1 ... p_n$) and $n + 1$ coefficients from the model ($m_0 ... m_n$), which is one intercept value m_0 and then one coefficient per property. Here I used the statistical technique known as cross-validation, in which I partitioned the dataset randomly into two with a ratio of 2:1, and used the larger partition to 'train' the model by applying linear regression to find the model, and the smaller partition to validate the produced model by calculating the expected calculated time and comparing it to the observations. To find the final model, this process was repeated several times using a new random partition and the average of each coefficient was taken. The boundaries that define the difficulty descriptions is taken directly from the available data; each puzzle was categorised by difficulty and the displayed as a boxplot, with the boundary determined by the relative averages and distributions of each category. The calculations of the linear regression model were handled by the apache commons maths library, and any graphics and direct experimentation with the data using the R statistical package.

E Generator Module

The Generator module's primary aim is to produce puzzles of a requested difficulty, quickly and accurately, and as a secondary goal it is to minimise the puzzles as much as possible. The program can either do all the steps automatically, or a user can control the steps with some input into the algorithm itself. The program works in a decremental manner in a three step process: it first generates a random solution, then a puzzle is formed by removing some of the givens, before randomising the output puzzle to mask some of the sideaffects of my algorithm. Producing a random complete grid is made trivial using the BruteForcer; starting with a blank grid the program fills in twenty-four random squares with a random number, applying arc-consistency as it goes to ensure that no directly invalid assignments are made. Twenty-four was chosen because it was at least seventeen which is the minimum size of a Sudoku puzzle, plus a few more to increase the diversity of the solutions generated. It then runs the BruteForce algorithm, and returns the first solution it finds. If in a minority of cases no solution is found, the algorithm simply restarts itself.

Through experimentation and running my Assessor on minimal puzzles found online, I found that they were rarely Gentle difficulty, and so using the current assessor, the generator would have to make exceptions for that and relax the restriction that a generated puzzle should be purely minimal. The process of creating a puzzle decrementally first creates a backup of the puzzle and removes a square from the original. If the new puzzle is not well posed (that is, it does not have exactly one solution) then the backup is used to restore the old puzzle, and we repeat from the start. How we choose which square to remove can have an effect on the difficulty of puzzles generated (Yuan-hai et al. 2009). I compared two methods: random square removal versus sequential removal (row by row, scanning left to right) and discovered that both methods created the same sort of minimal puzzles in the end: moderate puzzle half the time, with the others being split between tough and diabolical, and as expected they never created a minimal gentle puzzle. Random has the advantage of creating a better distribution of givens which looks nicer and can lead to slightly easier puzzles compared to sequential, which will always remove the first row straight away, which can be trickier to solve. However, by assessing the difficulty of

puzzles after each square is removed reveals that randomly produced puzzles increase in difficulty at a steady rate, while in general sequential puzzles will be relatively easier to start with before having steep jumps in difficulty towards the end. The final algorithm I employed tries to balance between both these techniques to use the advantages of both. To start with, the algorithm removes random squares up to a limit (default is 50) and if the puzzle at this time is too hard we go back to the start. Otherwise it starts the sequential method (starting from a random square everytime for more variation in puzzles) until the puzzle is fully minimised. We again check the difficulty of the puzzle, and if it is just right we return the generated puzzle. If it is too easy we start again from the start (since in the majority of cases we cannot make it harder by adding givens), and if it is too hard we add squares back in again in the reverse order until the right difficulty is achieved. This is where the advantage of sequential removal pays off; because the puzzles get harder quickly towards the end, it also means we can quickly reduce a hard puzzle to an easier one in a very short amount of time without having to start again.

The third and final part of the algorithm is a simple randomiser to change the layout of the puzzle, which is needed because if at the end of puzzle generation we have to add in many squares to correct the difficulty, then they will all be in a very noticeable horizontal line. To improve the spread, the algorithm applies 50 random transformations to the puzzle, which can either be: swapping two row or columns, swapping two stacks or bands, or transposing the puzzle (so rows are now columns and vice versa). This is a purely cosmetic step to adjust the layout, and doesn't need to use all of the possible transformations.

IV RESULTS

The primary method of deciding the effectiveness of the Solver is to look at what proportion of puzzles can be solved by it logically and without resorting to forcing a solution with trial and error, and in this respect the results are positive. In casual testing, the vast majority of the puzzles found in newspapers and books can be solved by logical pure applications in a very short amount of time, usually in the region of 100ms per puzzle. I attempted to quantify these findings by running the program on as many puzzles as I could obtain from the Internet of all kinds of alleged difficulties. We know from the work of the Assessor that the solver can handle all of the 1461 puzzles sampled from Sudoku.org.uk archives, but beyond that the collections of puzzles that have been attempted are recorded in Table 1.

Collection name	Puzzles	Description	Percentage Solved
Daily Puzzles	1461	sudoku.org.uk archives, difficulties range from 'Gentle' to 'Diabolical'. Used to build Assessor.	100%
Superior Puzzles	100	Published in The Times, some of the hardest from newspapers. Require advanced techniques from the contradiction finder.	100%
"Swordfish"	404	enjoysudoku.com forums, all using the 'swordfish' technique, again covered by the contradiction finder.	100%
Puzzle Zoo	375	enjoysudoku.com forum. Large variety of puzzles grouped into 100+ technique catagories. Each puzzle can be solved using only that technique and minimal other trivial techniques.	100%
"Hardest Ever"	20	Widely covered in media to the hardest puzzles conceived, including the infamous "AI Escargot". Published by Dr Arto Inkala in 2010.	0%
"Even Harder"	203	Sourced from enjoysudoku.com forum in response to previous collection, all measured to be of an even greater difficulty.	0%

Table 1: Table showing the results of running the solver on multiple different puzzle collections

To get the results for the Assessor, I ran the 20 rounds of training the model, before verifying it with the test partition of the dataset provided, and averaging the relating models to produce an aggregate model which will be a better model for the whole dataset in general. The R^2 value provides a measure of the correlation between my assessors prediction and the observed data, comes out at a value of 0.797 for the aggregate model. Over the rounds, this correlation varied remarkably rarely, with a range $= [0.78, 0.81]$. This strong correlation can be observed in the plot in Figure 2, with the predictions increasing accordingly from a base minimum prediction level at around the 15 minute mark. Also, the histogram of the relative errors is shown in Figure 3, where $percentageError = 100(calculatedTime - observedTime)/observedTime$. This shows that even though the errors tend to increase as the Assessor tried to predict harder puzzles, the error distribution is fairly low, with 80 percent of the predictions lying less than ± 20 percent of the true value.

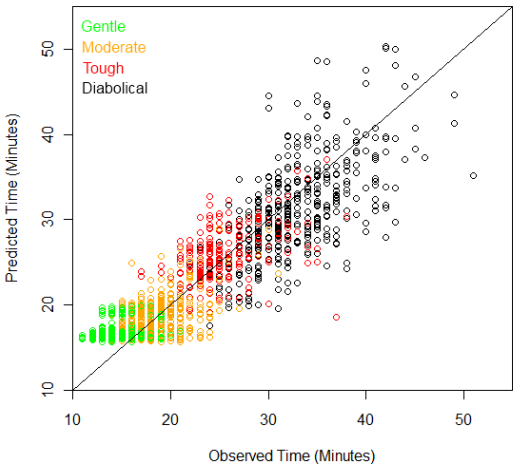

Figure 2: Graph plotting predicted solution time against the observed average solution time

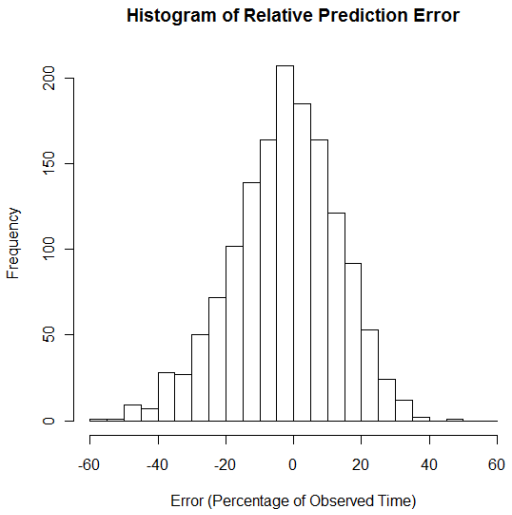

Figure 3: Histogram of the percentage error in the Assessor predictions

The Generator is functionally complete, in that it will always produce a puzzle of the desired difficulty, and it is just the running time which may vary; such algorithms are known as "Las Vegas" algorithms. The first step of creating a random solution runs in an average time of 50ms, and has to repeat itself only once or twice in the vast majority of cases. Furthermore, the transformation step is also trivially quick as it is just a set amount of applications of quick array manipulations. The most expensive step is the puzzle generation, which usually takes one to three seconds, but can be up to ten if it must start from the beginning too many times. However, this is standard performance across the difficulties, and this is preferable than it taking longer to produce Diabolical puzzles than Gentle ones.

V EVALUATION

Table 1 shows a clear cut between what puzzles my Solver can solve logically and those that it fails on. Since there is no collection with a mixture of solvable and unsolvable puzzles. I think this result is directed by the capabilities of the contradiction finder; while it can search for contradictions arbitrarily deep, it can only deduce squares where there is an application of a hidden single or a naked single, which just happen to be represented very well as columns to the Dancing Links model. I believe that the solver fails on the hardest of puzzles because they necessarily require some more advanced deductions about the candidates remaining in the squares, which would be very complicated to implement using DLX. For example, even a contradiction with only one implied square would fail to be recognised if it depended on an application of the intersection technique, even though this is a relatively easy technique to spot otherwise. Regardless of this limitation, I still consider the Solver a huge success since "AI Escargot" is the easiest known puzzle that can not be solved and this is still one of the most famous hard puzzles known. this means that the Moreover, the fact that the Solver can still provide a solution though the failsafe trial and error approach is an attractive feature to human users looking to tackle such puzzles.

For the Assessor, the quality of the correlation is very strong indeed; an R^2 of 0.797 is even better than those from the literature, of which 0.7103 was the best achieved by (Leone & Vaswani 2010) who used the same technique to generate a linear regression model. We both used the data available from sudoku.org.uk, but I consider my approach to works better because my dataset is larger because more time has elapsed and thus more daily puzzles are available, which will lead to a more informed model. Furthermore, their models just considered the techniques themselves rather than any extra features such as the sources of the techniques, memory or chaining of one move into the next like mine does. Indeed, the coefficients in the model indicate that the memory property and the number of remaining squares have a significantly larger affect on the final difficulty than any of the other properties. The model is not perfect however; strictly, each coefficient should be independent, while there is some overlap between the variables since multiple technique types contribute to the same pool of common properties - if you were to remove this limitation, the accuracy could be even greater.

The Generator ensures that the result is as minimal as is practically feasible. When the algorithm adds givens sequentially backwards to lower the difficulty, a better quality of minimality could be produced by looking at subsets of the removes squares rather than working backwards, but in practise this process could take a long time since even if you remove 50 random squares fist, there might still be 14 squares that need to be considered in subsets, which equates to checking $2^{14} = 16384$ different subsets. This is particularly damaging when you consider that assessing the difficulty of the puzzle is one of the more expensive operations in the process. Use of the BruteForcer is relatively less expensive, and so this is why the algorithm was designed to create a extremely minimal well posed solution before finally fitting it to the requested difficulty.
Considering my initial project requirements, I have met all of the high requirements, all of the medium level requirements and all but two of the low requirements; I havn't implemented Sudoku variations such as adding addition constraints or bounding the possible candidates in certain squares, but I think that it a justifiable omission since I have been so productive with the other main elements of the project. The other requirement was to implement a full Sudoku transformer to produce symmetrical, aesthetically pleasing puzzles. While I did use these partially in the Generator, I soon realised that the method I was thinking about using (Genetic Programming) would be far too unlikely to have any significant effect on any puzzle it would work on due to the limit nature of the transformations themselves.

VI CONCLUSIONS

Overall, this project has been a fascinating exploration into the realm of Sudoku puzzles. I have had an enjoyable time managing the project and exploring my own ideas as well as those of others from the related literature. I am happy with the results of the software; I have created a Sudoku solver which can manage all but the toughest puzzles, an Assessor with a good accuracy with real-world applications, and a Generator which can produce satisfying puzzles quickly and with enough variation to keep a human puzzler interested.

To continue the project, I would consider improving the Assessor further so that there are individual properties weighted for each technique type instead of one large pool of properties per puzzle, as I suspect this would provide more information to make an even further improved metric. I would still like to explore Sudoku transformations, particularly with respect to the minimum number of transformations required to change one puzzle into another, if at all possible.

References

Colbourn, C., Colbourn, M. & Stinson, D. (1984), 'The computational complexity of recognizing critical sets', *Graph Theory Singapore 1983* pp. 248–253.

Dechter, R. (2003), *Constraint processing*, Morgan Kaufmann.

Felgenhauer, B. & Jarvis, F. (2005), 'Enumerating possible Sudoku grids', *Preprint* .

Fowler, G. (2008), 'A 9x9 sudoku solver and generator', `http://public.research.att.com/~gsf/sudoku`.

Green II, R. (2009), 'Survey of the Applications of Artificial Intelligence Techniques to the Sudoku Puzzle'.

Han, C. & Lee, C. (1988), 'Comments on Mohr and Henderson's path consistency algorithm', *Artificial Intelligence* **36**(1), 125–130.

Leone, A., M. D. & Vaswani, P. (2010), 'Sudoku: Bagging a difficulty metric & building up puzzles', `http://www.math.washington.edu/~morrow/mcm/team2280.pdf`.

Mohr, R. & Henderson, T. (1986), 'Arc and path consistency revisited', *Artificial intelligence* **28**(2), 225–233.

Reeson, C., Huang, K., Bayer, K. & Choueiry, B. (2007), An interactive constraint-based approach to Sudoku, *in* 'PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE', Vol. 22, Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, p. 1976.

Royle, G. (2010), 'Minimum sudoku', `http://mapleta.maths.uwa.edu.au/~gordon/sudokumin.php`.

Russell, E. & Jarvis, F. (2006), 'Mathematics of sudoku II', *Mathematical Spectrum* **39**(2), 54–58.

Simonis, H. (2005), 'Sudoku as a constraint problem', *Proc. 4th Int. Works. Modelling and Reformulating Constraint Satisfaction Problems* pp. 13–27.

Sudopedia (2011), 'Solving techniques', `http://www.sudopedia.org/wiki/Solving_Technique`.

Takayuki, Y. (2003), Complexity and completeness of finding another solution and its application to puzzles, PhD thesis, Citeseer.

Weller, M. (2008), 'Counting, Generating, and Solving Sudoku'.

Yuan-hai, X., Biao-bin, J., Yong-zhuoAdvisor, L., Gui-feng, Y. & Hua-fei, S. (2009), 'Sudoku Puzzles Generating: From Easy to Evil', *Mathematics in Practice and Theory* .