There is no 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem via Hitting Set Enumeration

Gary McGuire*, Bastian Tugemann†, Gilles Civario‡

August 31, 2013

Abstract

The sudoku minimum number of clues problem is the following question: what is the smallest number of clues that a sudoku puzzle can have? For several years it had been conjectured that the answer is 17. We have performed an exhaustive computer search for 16-clue sudoku puzzles, and did not find any, thus proving that the answer is indeed 17. In this article we describe our method and the actual search. As a part of this project we developed a novel way for enumerating hitting sets. The hitting set problem is computationally hard; it is one of Karp's 21 classic NP-complete problems. A standard backtracking algorithm for finding hitting sets would not be fast enough to search for a 16-clue sudoku puzzle exhaustively, even at today's supercomputer speeds. To make an exhaustive search possible, we designed an algorithm that allowed us to efficiently enumerate hitting sets of a suitable size.

*School of Mathematical Sciences, University College Dublin, Ireland. E-mail: gary.mcguire@ucd.ie
†Munich, Germany.
‡Irish Centre for High-End Computing, Dublin, Ireland.

Contents

1 Introduction **3**
- 1.1 Summary description of method . . . 4
- 1.2 Other applications of hitting set algorithms . . . 4
- 1.3 Miscellaneous comments . . . 5

2 History **5**
- 2.1 Previous work by others . . . 5
- 2.2 Heuristic arguments that 16-clue sudoku puzzles do not exist . . . 7
  - 2.2.1 A statistical observation . . . 7
  - 2.2.2 A way of computing the minimum number of clues needed for general sudoku? 8

3 The catalogue of all sudoku grids **9**
- 3.1 Equivalence transformations . . . 9
- 3.2 Using the sudoku group and Burnside's lemma to count equivalence classes . . . 10
- 3.3 Enumerating representatives . . . 13

4 Overall strategy of checker **13**
- 4.1 Unavoidable sets . . . 14
- 4.2 Hitting sets . . . 16
- 4.3 Running all 16-clue candidate puzzles through a sudoku solver . . . 17

5 Some theory of unavoidable sets **18**
- 5.1 Definition and elementary properties . . . 18
- 5.2 Finding minimal unavoidable sets efficiently using blueprints . . . 20
- 5.3 Higher-degree unavoidable sets . . . 24

6 Hitting set algorithm from the original checker **27**
- 6.1 Description of the basic algorithm . . . 27
- 6.2 Using the dead clue vector to prevent multiple enumeration of hitting sets . . . 28

7 Hitting set algorithm of the new checker **30**
- 7.1 Improving backtracking using higher-degree unavoidable sets . . . 31
- 7.2 Consolidating binary vectors for the innermost loops . . . 34
- 7.3 Taking the effective size of the minimal unavoidable sets into account . . . 36

8 Running through all grids: the final computation **36**
- 8.1 Some remarks on the implementation of the new checker . . . 36
- 8.2 Parallelisation strategy for the grid search . . . 38
- 8.3 Ensuring the correctness of our programme checker . . . 39
- 8.4 The actual computation . . . 40
- 8.5 GPU computing opportunities . . . 40

Bibliography **41**

1 Introduction

Sudoku is a logic puzzle — one is presented with a 9×9 grid, some of whose cells already contain a digit between 1 and 9; the task is then to complete the grid by filling in the remaining cells such that each row, each column, and each 3×3 box contains the digits from 1 to 9 exactly once. Moreover, it is always understood that any proper (valid) sudoku puzzle must have only one completion. In other words, there is only a single solution, only one correct answer. In this article, we consider the issue of *how many* clues (digits) need to be provided to the puzzle solver in the beginning. There are 81 cells in a grid, and in most newspapers and magazines, usually around 25 clues are given. If too few clues are given at first, then there is more than one solution, i.e., the puzzle becomes invalid. It is natural to ask how many clues are always needed. This is the sudoku minimum number of clues problem:

What is the smallest number of clues that can be given such that a sudoku puzzle has a unique completion?

More informally — what is the smallest number of clues that you could possibly have?

There are puzzles known with only 17 clues, here is an example:

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

Figure 1: A 17-clue sudoku puzzle

However, nobody ever found any proper 16-clue puzzles. Consequently, people started to conjecture that the answer to the minimum number of clues problem is 17. We have proved this conjecture, and in this article we present our method.

The strategy we chose to solve this problem is a very obvious one: use a computer to exhaustively search through every possible sudoku solution grid, one by one, for a 16-clue puzzle. So we took the viewpoint of considering each completed sudoku grid, one at a time, and looking for 16-clue puzzles whose solution was that grid — the reader can think of these puzzles as being contained in the given grid. Our year-long search, carried out at the Irish Centre for High-End Computing (ICHEC), turned up no proper 16-clue puzzles, but had one existed, we would have found it.

Due to the sheer number of sudoku solution grids a brute force search would have been infeasible, but we found a better approach to make this project possible. Our software for exhaustively searching through a completed sudoku grid, named *checker*, was originally released in 2006. However, this first version was rather slow. Indeed, the paper [1] estimates that our original *checker* of late 2006 would take over 300,000 processor-years in order to search every sudoku grid. With our new algorithm, this

computation actually took only about 800 processor-years. In the present article we describe both the original and the new version of *checker*. In particular, we describe our hitting set algorithm, which is at the heart of *checker*.

The proof in this article is a computer-assisted proof of a mathematical theorem. There are many precedents for this type of proof. The most famous is without doubt the proof of the four-colour theorem by Appel and Haken [2], see also [3]. Another more recent but also already legendary example of a computer-assisted proof is the proof of the Kepler conjecture by Thomas Hales [4]. A very recent example is the proof of the weak Goldbach conjecture [5]. There are a great many theorems published in the mathematical literature where the authors have used computers in their proof. Comments about correctness always apply; we discuss this more with regard to our proof in Section 8.3.

As a side comment, there have been attempts to solve the sudoku minimum number of clues problem using mathematics only, i.e., without the help of computers. However, nobody has made any real progress — though it is easy to see why a 7-clue puzzle cannot have a unique solution,[1] a theoretical proof of the nonexistence of an 8-clue puzzle is still lacking. This is far from the answer of 17, so that a purely mathematical solution to the minimum number of clues problem is a long way off.

1.1 Summary description of method

As just explained, the objective of this project was to prove that there are no 16-clue sudoku puzzles having exactly one completion, or to find such a puzzle, had one existed. Our choice of method meant that there were three steps:

1. make a catalogue of all completed sudoku grids — obviously only finitely many grids exist;
2. write *checker*, i.e., implement efficient algorithms for exhaustively searching a given completed sudoku grid for puzzles having 16 clues and whose (unique) solution is the grid at hand;
3. run through the catalogue of all completed grids and search every grid in turn using *checker*.

We will describe each of these subprojects in detail in Sections 3, 4–7 and 8, respectively.

1.2 Other applications of hitting set algorithms

Solving the sudoku minimum number of clues problem serves as a way to introduce our algorithm for enumerating hitting sets. However, this algorithm has uses beyond combinatorics. To begin with, it is in principle applicable to any instance of the hitting set problem, be it the *decision version* (determining if hitting sets of a given size exist) or the *optimization version* (finding a smallest hitting set). Such situations occur in bioinformatics (gene expression analysis [6]), software testing as well as computer networks [7], and when finding optimal drug combinations [8, 9]; the last of these papers lists protein network discovery, metabolic network analysis and gene ontology as further areas in which hitting set problems naturally arise. Besides, our algorithm can be applied to the hypergraph transversal problem, which appears in artificial intelligence and database theory [10]. Lastly, we note that the vertex cover problem from graph theory is a special case of the hitting set problem, and that the set cover problem is in fact equivalent to the hitting set problem.[2] The former occurs in computational biology, see [11], while the latter finds application in reducing interference in cellular networks [12].

[1] In a 7-clue sudoku puzzle, the two missing digits can be interchanged in any solution to yield another solution.

[2] We did not find any papers about the set cover problem that were useful to us in improving our hitting set algorithm.

1.3 Miscellaneous comments

This article is an update of a preprint, which we posted online [13] on 1^{st} January 2012 together with the full source code of the new *checker*. We shall not discuss any of the following topics here: how to solve or create sudoku puzzles, how to rate the difficulty of sudoku puzzles, or how to write a sudoku solver programme. We emphasize that we are not saying that every completed sudoku grid contains a 17-clue puzzle (actually, very few do). We *are* saying that no grid contains any 16-clue puzzles.

2 History

The work on this project began in August 2005, when we started developing *checker*, which to our knowledge was the first computer programme that made it possible to search a sudoku solution grid exhaustively for all n-clue puzzles, where both the grid and the number n were supplied by the user. The final release of this original version of *checker* is of November 2006 [14]. Throughout 2007, this project lay practically dormant, but in the second half of the following year we realized that there was considerable potential to make *checker* faster. Therefore, in early 2009 we got started implementing a new version of *checker*, redesigned from the ground up and tailored to the case $n = 16$, which we did not post online, however, until every sudoku grid had actually been exhaustively searched for 16-clue puzzles. This wholly new *checker* takes only a few seconds to search through a typical grid, whereas our original release from late 2006 needs about half an hour per grid on average.

To be able to do some early testing on their clusters, we were granted a Class C project account at the Irish Centre for High-End Computing (ICHEC) in July 2009. In September of the same year we also successfully applied for a PRACE prototype award, which provided us with nearly four million core hours on JUGENE in Jülich, Germany — then Europe's fastest supercomputer — as well as time on a Bull cluster at CEA, France, a Cray XT5 cluster at CSC, Finland, and an IBM Power 6 cluster at SARA in the Netherlands. The goal of this prototype project was to evaluate different loadbalancing strategies of the hitting set problem; the actual code we ran was an early version of the new *checker*.

2.1 Previous work by others

- In Japan, sudoku was introduced by the publisher Nikoli in the 1980s. Japanese puzzle creators have made puzzles with 17 clues, and with no doubt wondered whether 16 clues were possible. Nikoli have a rule that none of their puzzles will have more than 32 clues.

- Over the years, people have collected close to 50,000 different 17-clue sudoku puzzles, which may be downloaded from Gordon Royle's homepage [15]. Most of these were found by Royle, who is a mathematician at the University of Western Australia and who compiled this list while searching for a 16-clue puzzle. On the sudoku forums [16] we became aware of a rather special grid (shown on p. 30), which has 29 different 17-clue puzzles, also discovered by Royle. Many considered this grid a likely candidate to contain a 16-clue puzzle. Actually, we initially started to work on *checker* to search this particular grid.

- By the end of 2007, the sudoku minimum number of clues problem had been mentioned in several journal publications [17, 18, 19, 20, 21]. The first one is an article by the French computer scientist Jean-Paul Delahaye entitled *The Science behind Sudoku*, which was first published in

the June 2006 issue of the *Scientific American*. This article in fact quotes one of the authors of this study (Gary McGuire) in conjunction with the minimum number of clues problem.

- Between autumn 2007 and spring 2008, a team at the University of Graz in Austria verified by distributed exhaustive computer search that there are no proper sudoku puzzles having eleven clues. They continued for about another year, and by May 2009, they had apparently completed most of the computations necessary to show that twelve clues are never sufficient for a unique solution either. Their stated aim was to build up to proving that no 16-clue sudoku puzzle exists, yet as of early 2010, that project seemed not to be active anymore [22].

- Since 2007, Max Neunhöffer of the University of St. Andrews in Scotland has lectured several times at different venues on the mathematics of sudoku, discussing in particular the minimum number of clues problem [23]. According to the slides of his talks, Neunhöffer appears to even have written a computer programme for searching a sudoku solution grid for 16-clue puzzles.

- In 2008, a 17-year-old girl submitted a proof of the nonexistence of a 16-clue sudoku puzzle as an entry to *Jugend forscht* (the German national science competition for high-school students). She later published her work in the journal *Junge Wissenschaft* (No. 84, pp. 24–31). However, when Sascha Kurz, a mathematician at the University of Bayreuth, Germany, studied the proof closely, he found a gap that is probably very difficult, if not impossible, to fix.

- In mid-2010, the Bulgarian engineer Mladen Dobrichev released the initial version of his open-source tool *GridChecker* [24]; he has since provided several updates. Dobrichev's programme, also written in C++, basically does the same thing as our original *checker*, although it is at least one order of magnitude faster.

- In October 2010, a group of computer scientists at the National Chiao Tung University, Taiwan, led by I-Chen Wu started a distributed search for 16-clue puzzles on the Internet using BOINC. Their strategy is exactly the same as ours (i.e., consider each sudoku solution grid individually and exhaustively search for a 16-clue subset whose only completion is the given grid). This had become feasible since Wu's team had succeeded in speeding up our original *checker* by a factor of 129, so that it would take them an estimated 2,400 processor-years to examine every grid for 16-clue puzzles. Wu also spoke about this in November 2010 at the International Conference on Technologies and Applications of Artificial Intelligence, and he and Hung-Hsuan Lin together published the article *Solving the Minimum Sudoku Problem* in the conference proceedings, describing some of the techniques they had used to improve *checker*. The complete details of their work are described in the article [1]. As far as the progress of the BOINC search is concerned, according to that project's website, as of 31^{st} December 2011 — the day before we announced our result ($\mathrm{arXiv:1201.0749v1}$) — they had checked 1,453,000,000 sudokus (26.5 per cent of all cases that need to be considered, see Section 3).

- Toward the end of December 2011, the digital edition of the book *Taking Sudoku Seriously: The Math Behind the World's Most Popular Pencil Puzzle* by Jason Rosenhouse and Laura Taalman of James Madison University in Harrisonburg, Virginia, came out; the printed version followed several weeks later [25]. This book devotes the whole of Section 9.4 *The Rock Star Problem* to the minimum number of clues problem, saying:

> "If you can figure it out, you will be a rock star in the universe of people who care about such things. Granted, that is a far smaller universe than the one full of people who care about actual rock stars, but still, it would be great."

Regarding the hitting set problem, we were surprised by the paucity of relevant literature, given the wide range of applications of an efficient algorithm for finding hitting sets, see Section 1.2. Most authors appear to have concentrated on the special case of the d-hitting set problem for small d, such as $d = 3$, where d is the (maximum) number of elements in the sets to be hit. However, to tackle the sudoku minimum number of clues problem efficiently, we would have needed a method for $d = 12$ at least, so that these algorithms were not actually useful to us. Some researchers have generalized their ideas to the case of arbitrary d, but the only such recent paper we could find describes an algorithm of complexity $\mathrm{O}(\alpha^k + n)$, where $\alpha = d - 1 + \mathrm{O}(\frac{1}{d})$, k is the cardinality of the desired hitting set, and n is the length of the encoding of the input [11]. In fact, this hitting set algorithm is somewhat similar to the one in our original version of *checker*. In a forthcoming article we shall therefore carry out the formal complexity analysis of our new algorithm.[3] We estimate its average-case runtime complexity to be $\mathrm{O}(d^{k-2})$ with any instance of the hitting set problem for which the sets that need to be hit are of comparable density as with the sudoku minimum number of clues problem.

2.2 Heuristic arguments that 16-clue sudoku puzzles do not exist

There are two heuristic arguments as to why 16-clue puzzles should not exist, which we present here. The first argument is statistical, while the second one is a pattern that appears to correctly predict the minimum number of clues needed with general $(n \times n)$ sudoku.

2.2.1 A statistical observation

For several years now, whenever people send Gordon Royle (see the second item in Section 2.1) a list of 17-clue puzzles, there are usually not too many new puzzles. One correspondent sent 700 puzzles, and it turned out that only 33 were new. Assuming that both Royle and this correspondent had drawn their puzzles at random from the universe of all 17-clue puzzles, Ed Russell computed the maximum likelihood estimate for the size of the universe to be approximately 34,550 at a time at which Royle's list contained around 33,000 puzzles [26]. In retrospect this is an underestimate; nevertheless we may infer that the 49,151 puzzles on the most recent version of Royle's list must be almost all the 17-clue puzzles in existence.

On the other hand, among all sudoku solution grids known to contain at least one 17-clue puzzle, the highest number of such puzzles in any one grid is 29.[4] So presently there is no sudoku grid known with 30 or more 17-clue puzzles. Given a (hypothetical) 16-clue sudoku puzzle, adding one clue to it in all possible ways results in 65 different 17-clue puzzles all having the same solution grid. Because

[3] "Often, performance measures the line between the feasible and the infeasible [...] If you're talking about doing stuff that nobody's done before, one of the reasons often that they haven't done it is, because it's too time-consuming, things don't scale, and so forth. So that's one reason, is the feasible vs. infeasible." —Charles E. Leiserson, Professor of Computer Science, MIT, explaining why it is important to study algorithms and their performance (Source: MIT 6.046J / 18.410J Introduction to Algorithms, Fall 2005, Lecture 1, http://www.youtube.com/watch?v=JPyuH4qXLZ0 from 23:40 to 24:40)

[4] Exactly one grid having this many 17-clue puzzles is known; further there are four grids known containing 20, 14, 12 respectively 11 puzzles with 17 clues. All other grids arising from a puzzle on Royle's list have less than ten such puzzles.

of the large gap between 29 and 65, assuming that Royle's list is practically complete, when we began running compute jobs for this project, it was already clear that a 16-clue puzzle was unlikely to exist.

One additional experimental finding that supports the assumption that Royle's list is nearly complete is the following. When considering the actual solutions of all the puzzles on this list, then there are exactly 46,294 distinct sudoku grids (up to equivalence, see Section 3). We took a random sample of 1 in 5,000 sudoku solution grids, exhaustively searched all grids in this sample for 17-clue puzzles, and got a hit in exactly nine cases, which is remarkable since $46{,}294/5{,}000 \approx 9.26$.

2.2.2 A way of computing the minimum number of clues needed for general sudoku?

In order to describe the second heuristic we first need to introduce some notation. With $n \times n$ sudoku, we denote the minimum number of clues needed for a puzzle to have a unique solution by $\mathrm{scs}(n)$, for *smallest critical set*. In general terms, critical sets are (minimal) subsets of a structure that determine the entire structure. There have been papers on smallest critical sets in other combinatorial structures; for example, see [27] for 8×8 Latin squares. We will now demonstrate how to compute $\mathrm{scs}(n)$ from only the total number of $n \times n$ solution grids — the results we get are correct for all n for which the value of $\mathrm{scs}(n)$ is currently known. Here is a summary of the literature to date:

grid dimension $(n \times n)$	**boxes**	**total number of solution grids**	$\mathbf{scs}(n)$
4×4	2×2	288	4
6×6	2×3	28,200,960	8
8×8	2×4	29,136,487,207,403,520	14
9×9	3×3	6,670,903,752,021,072,936,960	17

For the total number of grids, [28] has references. A proof that $\mathrm{scs}(4) = 4$ may be found, e.g., in [21]. The result that $\mathrm{scs}(6) = 8$ is due to Ed Russell, who, back in 2006, had checked that no proper 7-clue puzzle exists for 6×6 sudoku [29]. In contrast, the 8×8 case remained open until November 2011, when Christoph Lass of the University of Greifswald in Germany announced that he had proved that $\mathrm{scs}(8) = 14$ [30]. Although Lass, too, performed a computer search, his approach was not to inspect every completed 8×8 sudoku grid, but rather to generate all 13-clue puzzles (up to equivalence) and then see if any of them had a unique completion. Observe now that, for these four cases,

$$\begin{aligned}
4 \times 6 \times 8 &< 288 &< 4 \times 6 \times 8 \times 10,\\
6 \times 8 \times \cdots \times 18 &< 2.82 \times 10^7 &< 6 \times 8 \times \cdots \times 18 \times 20,\\
8 \times 10 \times \cdots \times 32 &< 2.91 \times 10^{16} &< 8 \times 10 \times \cdots \times 32 \times 34,\\
9 \times 11 \times \cdots \times 39 &< 6.67 \times 10^{21} &< 9 \times 11 \times \cdots \times 39 \times 41.
\end{aligned}$$

So in each case, if T is the total number of completed $n \times n$ sudokus and $C := \mathrm{scs}(n)$, then

$$\underbrace{n \times (n+2) \times \cdots \times (n+2(C-2))}_{(C-1) \text{ factors}} < T < \underbrace{n \times (n+2) \times \cdots \times (n+2(C-1))}_{C \text{ factors}}.$$

Moreover, for 16×16 sudoku, according to [31], the conjectured size of a smallest critical set is 56, while from [28] the estimate of the total number of completed 16×16 grids is about 5.96×10^{98}. If we assume that these two values are correct, then both inequalities hold. Note also that the first of the above inequalities is logarithmically quite sharp in all five cases just considered.

3 The catalogue of all sudoku grids

This section describes Step 1 of Section 1.1. As already stated in the last section, in total there are

$$6{,}670{,}903{,}752{,}021{,}072{,}936{,}960 \;\approx\; 6.7 \times 10^{21}$$

completed sudoku grids [33]. However, with this project, it was not necessary to analyze all of them. For instance, even though relabelling the digits of a given grid yields a different grid, clearly this will not actually change the substance of the grid because the individual digits carry no significance — any nine different symbols could be used. There are several other such *equivalence transformations*, e.g., rotating the grid or taking its transpose, or interchanging the first with the second row, none of which alter the property of containing a 16-clue puzzle. Technically, we introduce an equivalence relation on the set of all completed sudoku grids, where two grids are equivalent if one may be obtained from the other by applying one or more of the equivalence transformations. Whether a 16-clue puzzle exists in a particular grid is then an invariant of its equivalence class. Hence we only needed to search one representative from each grid equivalence class for 16-clue puzzles in order to solve the sudoku minimum number of clues problem.

3.1 Equivalence transformations

In order to give a concise description of the equivalence transformations, we introduce the following terminology. A *band* in a sudoku grid is either the set of rows 1–3, rows 4–6, or rows 7–9; similarly a *stack* denotes either the set of columns 1–3, columns 4–6, or columns 7–9. (Just like with matrices in linear algebra, the rows and columns of a sudoku grid are numbered from top to bottom respectively left to right.) This illustration shows the three bands and stacks in an empty grid:

Figure 2: The bands and stacks in an empty sudoku grid

Here are the actual equivalence transformations, which other authors sometimes refer to as *symmetry operations* — note that reflections and rotations are already included in these:

1. relabelling the digits 1–9, i.e., replacing each instance of the digit d by $\pi(d)$ for some $\pi \in S_9$, for all $d = 1, \ldots, 9$;
2. permuting the rows (columns), by
   (a) swapping bands (stacks), or
   (b) swapping the three rows (columns) within a given band (stack);
3. transposing the grid.

We call two sudoku solution grids *equivalent* if one may be obtained from the other by any sequence of the equivalence transformations listed above. It is routine to check that this definition gives rise to an equivalence relation in the mathematical sense on the set of all sudoku solution grids. A sequence of these transformations mapping a grid to itself is called an *automorphism* of the grid. The possible automorphism groups have been determined [34], although the vast majority of grids do not actually possess any nontrivial automorphisms. The largest automorphism group occurring has order 648, and is realized, up to equivalence, by exactly one grid (see Figure 10).

Now note that the operation of relabelling the digits commutes with all other equivalence transformations. Also, every row operation commutes with each of the column operations. Finally, swapping the i^{th} with the j^{th} row (column), $1 \leq i < j \leq 9$, and then transposing the resulting grid is clearly no different from first taking the transpose and then interchanging the i^{th} with the j^{th} column (row). The consequence of all this is that if G and H are equivalent grids, then there exists a digit permutation π, as well as a row permutation ρ and a column permutation σ such that either

$$H \ = \ (\pi \circ \sigma \circ \rho)(G) \qquad \text{or} \qquad H \ = \ (\pi \circ \sigma \circ \rho)(G^T). \tag{1}$$

The following easy result shows that the property of containing a 16-clue puzzle is an invariant of the equivalence class of a grid.

Lemma 1. *Suppose that a sudoku solution grid G has a 16-clue puzzle. Then all grids equivalent to G contain a 16-clue puzzle.*

Proof. Let $P \subset G$ be a 16-clue puzzle. Then P^T is a 16-clue puzzle contained in G^T, because if H is any sudoku solution grid containing P^T, then $P = (P^T)^T \subset H^T$, but as G is the unique completion of P it follows that $H^T = G$, i.e., $H = G^T$, so that G^T is in fact the only completion of P^T.

Next let ρ be any permutation of the rows of a grid. We claim that $\rho(P)$ is a proper 16-clue puzzle of $\rho(G)$. For if H is any completion of $\rho(P)$, then $P = \rho^{-1}(\rho(P)) \subset \rho^{-1}(H)$, and by uniqueness of G it follows that $\rho^{-1}(H) = G$, i.e., $H = \rho(G)$. Hence $\rho(G)$ is the only completion of $\rho(P)$.

A similar argument shows that if σ is any rearrangement of the columns and π is any permutation of the digits 1–9, then $\sigma(P)$ and $\pi(P)$ are 16-clue puzzles of the grids $\sigma(G)$ and $\pi(G)$, respectively. On combining everything we just noted, the statement of the lemma now follows directly from (1). □

An obvious question to ask is: how many sudoku solution grids are there, up to equivalence? This is a natural mathematical question in its own right, but it was especially relevant for this project since we only needed to search one representative from each grid equivalence class for 16-clue puzzles, by the preceding lemma.

3.2 Using the sudoku group and Burnside's lemma to count equivalence classes

The set of all equivalence transformations defined above forms a group with respect to composition of functions, called the *sudoku group*, which acts on the set of all sudoku solution grids. The orbits under this group action are then precisely the grid equivalence classes. Of course the reason for employing group theory is that this makes it easier to determine the total number of equivalence classes. So if

$$T \ := \ \{\sigma \in S_9 \mid \forall\, 1 \leq i < j \leq 9 \,:\, \lceil i/3 \rceil = \lceil j/3 \rceil \rightarrow \lceil \sigma(i)/3 \rceil = \lceil \sigma(j)/3 \rceil\},$$

then (as a *set* only — see (2) below) the sudoku group is $S_9 \times T \times T \times C_2$. In this cartesian product, S_9 corresponds to the digit renumberings, the two copies of T provide the row and column permutations, while C_2 (the cyclic group having two elements) is appended to specify if the transpose is being taken. Note that, in the definition of T, the condition $\lceil i/3 \rceil = \lceil j/3 \rceil$ only if $\lceil \sigma(i)/3 \rceil = \lceil \sigma(j)/3 \rceil$, for all $1 \leq i < j \leq 9$, ensures that every $\sigma \in T$ preserves bands (stacks) as the index of the band (stack) containing the i^{th} row (column) is given by $\lceil i/3 \rceil$. Moreover, the number of elements in the group T equals $9 \times 2 \times 6 \times 2 \times 3 \times 2 = 1{,}296$ — if $\sigma \in T$, then there are nine possibilities for $\sigma(1)$, but once the value of $\sigma(1)$ has been chosen, only two possibilities remain for $\sigma(2)$ because of the requirement that $\lceil \sigma(1)/3 \rceil = \lceil \sigma(2)/3 \rceil$, and since we further need that $\lceil \sigma(2)/3 \rceil = \lceil \sigma(3)/3 \rceil$, the choice of $\sigma(2)$ already implies the value of $\sigma(3)$; similarly there are six possibilities for $\sigma(4)$, and with $\sigma(4)$ chosen, just two possibilities for $\sigma(5)$ remain etc. So $\#T = 1{,}296 = 6^4$, which is in line with what one would expect as there are $3! = 6$ ways to arrange the bands, as well as 3! permutations of the rows in each of the three bands. Actually, an isomorphism between $S_3 \times S_3 \times S_3 \times S_3$ and T is given by the map

$$(\sigma_1, \sigma_2, \sigma_3, \gamma) \;\mapsto\; \big(\{1, \ldots, 9\} \ni i \mapsto \sigma_b(i - 3(\lceil i/3 \rceil - 1)) + 3(b-1)\big), \quad \sigma_1, \sigma_2, \sigma_3, \gamma \in S_3,$$

where $b := \gamma(\lceil i/3 \rceil)$ and the group multiplication in $S_3 \times S_3 \times S_3 \times S_3$ is defined by

$$(\sigma_1, \sigma_2, \sigma_3, \gamma) * (\tau_1, \tau_2, \tau_3, \lambda) \;=\; (\sigma_1 \circ \tau_{\gamma^{-1}(1)}, \sigma_2 \circ \tau_{\gamma^{-1}(2)}, \sigma_3 \circ \tau_{\gamma^{-1}(3)}, \gamma \circ \lambda),$$

for $\sigma_i, \tau_i, \gamma, \lambda \in S_3, i = 1, 2, 3$. The reader may have noticed that $S_3 \times S_3 \times S_3 \times S_3$ equipped with this particular composition law is in fact $S_3 \text{ wr } S_3$, the *wreath product* of S_3 by itself such that S_3 acts on $\{1, 2, 3\}$ in the natural fashion. Even so, we shall briefly recall how to construct wreath products.

For two groups N and H and a homomorphism $\psi \colon H \to \mathrm{Aut}(N)$, the (external) *semidirect product* of N by H with respect to ψ, denoted $N \rtimes_\psi H$, is defined like so. As a set, $N \rtimes_\psi H := N \times H$, the ordinary cartesian product of N and H; the group multiplication is given by

$$(n_1, h_1) * (n_2, h_2) \;=\; (n_1 \psi(h_1)(n_2), h_1 h_2),$$

where $n_1, n_2 \in N$ and $h_1, h_2 \in H$ (a straightforward calculation confirms that all group axioms are satisfied). The wreath product is then a special case of the semidirect product. Suppose that G and H are groups such that H acts on the set $\{1, \ldots, n\}$ for some positive integer n, and let $\phi \colon H \to S_n$ be the homomorphism corresponding to this (left) action. Set $N = G^n$, the direct product of n copies of G. For each $h \in H$, the map $\pi_h \colon N \to N, (g_1, \ldots, g_n) \mapsto (g_{\phi(h)(1)}, \ldots, g_{\phi(h)(n)})$, is clearly a group isomorphism, i.e., $\pi_h \in \mathrm{Aut}(N)$, for all $h \in H$. So there is a mapping $\pi \colon H \to \mathrm{Aut}(N), h \mapsto \pi_{h^{-1}}$, which is in fact a homomorphism (routine verfication). The wreath product of G by H with respect to the associated group action, written $G \text{ wr } H$, is finally defined to be $N \rtimes_\pi H$, the semidirect product of $N = G^n$ by H using the homomorphism $H \to \mathrm{Aut}(N)$ induced by the action of H on $\{1, \ldots, n\}$. Explicitly, the group multiplication in $G \text{ wr } H$ is given by ($g_1, \ldots, g_n, g'_1, \ldots, g'_n \in G, h, h' \in H$)

$$(g_1, \ldots, g_n, h) * (g'_1, \ldots, g'_n, h') \;=\; (g_1 g'_{\phi(h^{-1})(1)}, \ldots, g_n g'_{\phi(h^{-1})(n)}, hh').$$

After this digression, we now continue with the actual definition of the sudoku group and its action on the set of all solution grids. In the beginning of this subsection, we already defined the sudoku group to be $S_9 \times T \times T \times C_2$ as a set only, the reason being that the law of composition is again not simply

that of the direct product, which is so because the various types of equivalence transformations do not all commute with each other.

Specifically, as already noted in the discussion right before (1), the relabelling of digits commutes with all other equivalence transformations, and every row permutation commutes with every column permutation. Thus the group generated by just these three types of transformations actually is a direct product, namely $S_9 \times T \times T$. However, first permuting the rows and then taking a grid's transpose is not the same as performing these operations in reverse order; on taking the transpose first, we need to apply the respective permutation to the columns instead of the rows. So this leads very naturally to a semidirect product because the semidirect product, really being a generalization of the direct product, provides exactly the additional flexibility needed to handle this situation. In fact, the homomorphism $\psi\colon C_2 \to \mathrm{Aut}(S_9 \times T \times T)$ that yields the correct semidirect product in our case is very simple — if the group C_2 is generated by g, i.e., if g is the nonidentity element of C_2, then under ψ,

$$g \;\mapsto\; \big(S_9 \times T \times T \ni (\pi, \rho, \sigma) \mapsto (\pi, \sigma, \rho)\big).$$

So $\psi(g)$ merely swaps the row and column permutations (the second respectively third component of a triple $(\pi, \rho, \sigma) \in S_9 \times T \times T$). At last we can properly and fully define the sudoku group; it is

$$(S_9 \times T \times T) \rtimes_\psi C_2. \tag{2}$$

This group has order $9! \times 6^4 \times 6^4 \times 2$. The permutation part only (i.e., when the digit relabellings are omitted) is a subgroup of order $6^4 \times 6^4 \times 2 = 3{,}359{,}232$. For defining the group action on the set of all completed sudoku grids, we identify a grid with the corresponding element of $\mathbb{M}_{9\times 9}(\{1, \ldots, 9\})$, i.e., we think of a grid as a 9×9 matrix $[a_{i,j}]$ with integral coefficients between 1 and 9. An arbitrary member $(\pi, \rho, \sigma, \ell)$ of the sudoku group $(S_9 \times T \times T) \rtimes_\psi C_2$ then acts on a grid $[a_{i,j}]$ as follows:

$$(\pi, \rho, \sigma, \ell).[a_{i,j}] \;=\; \begin{cases} [\pi(a_{\rho^{-1}(i),\sigma^{-1}(j)})], & \ell = 1_{C_2}, \\ [\pi(a_{\sigma^{-1}(j),\rho^{-1}(i)})], & \text{otherwise.} \end{cases}$$

The check that this really defines a group action is again a completely routine calculation. (Here, 1_{C_2} denotes the identity element of the group C_2.) Determining the precise number of orbits of this action can be done by appealing to Burnside's lemma, sometimes called Burnside's counting theorem or the Cauchy-Frobenius lemma. This was carried out by Ed Russell and Frazer Jarvis in 2006 [35]. Using a very clever computer calculation — taking only one second — they proved that there are exactly

$$5{,}472{,}730{,}538 \;\approx\; 5.5 \times 10^9$$

orbits. So this is the number of equivalence classes, i.e., there are $5{,}472{,}730{,}538$ *essentially different* sudoku grids in total. Later this result was independently established by Kjell Fredrik Pettersen.

We remark that this provides another way to see that most sudoku grids do not have any nontrivial automorphisms, as noted earlier. As if we divide the total number of grids (see p. 9) by the number of essentially different grids, we get that the average number of grids per equivalence class is

$$\frac{6{,}670{,}903{,}752{,}021{,}072{,}936{,}960}{5{,}472{,}730{,}538} \;\approx\; 9! \times 3{,}359{,}058.6,$$

which is roughly 99.9948% of the order of the sudoku group — in actual fact there are only $560{,}151$ essentially different grids having nontrivial automorphism group [36].

3.3 Enumerating representatives

For this project it was not enough to just know the number of sudoku grid equivalence classes; it was necessary to enumerate a complete set of representatives, and store these in a file. Fortunately for us, Glenn Fowler had already written a computer programme named *sudoku* [37] that is capable, among many other things, of enumerating all inequivalent completed sudoku grids in their respective minlex representation — the *minlex representation*, or *minlex form*, of a grid is the member of its equivalence class that comes first when all grids in that equivalence class are ordered lexicographically.

Uncompressed, the *catalogue* (of grids) would require about 418 GB of storage space, but Fowler also developed a data compression algorithm to store the catalogue in under 6 GB. Note the genuinely amazing compression rate; each grid occupies on average only a little more than one byte. Fowler has kindly shared his executables, and we have used them in order to generate and compress the complete list of essentially different sudoku solution grids in minlex form. We were thus able to store the entire catalogue on a single DVD.

We verified the completeness of the catalogue as follows. We wrote a small tool that computes the minlex form of a given sudoku solution grid, and then used this tool to double-check that each grid in the catalogue actually was in minlex form already. Further, we made sure that the catalogue itself was ordered lexicographically and that there were no repetitions. In this way we knew that no two grids in the catalogue were members of the same equivalence class. Because of the result on the total number of equivalence classes from the last subsection, that was all it took to be certain that the catalogue was really complete.

4 Overall strategy of checker

This section concerns Step 2 of Section 1.1. The obvious algorithm for exhaustively searching a given completed sudoku grid for 16-clue puzzles is to simply consider all subsets of size 16 of that grid and test if any have a unique completion. Even though such a *brute force* algorithm is effective in that the minimum number of clues problem could in principle be solved that way, the actual computing power required would be far too great. In fact, searching just one grid in this way would be infeasible as

$$\binom{81}{16} \approx 3.4 \times 10^{16}.$$

Fortunately, with just a little theory, the number of possibilities to check can be reduced quite dramatically. Very briefly, it is possible to identify regions (subsets) in a given completed sudoku grid, called *unavoidable sets*, that must always have at least one clue from any proper puzzle in that grid; once all the unavoidable sets have been identified for the given grid, the problem of finding 16-clue puzzles in that grid can be solved by finding *hitting sets* — sets of size 16 that intersect (hit) each member of the collection of unavoidable sets. This is a much smaller problem, i.e., the approach just outlined greatly reduces the compute time taken per grid compared to the brute force way.

In practice, we use only a restricted collection of unavoidable sets, namely those having no more than twelve elements. Obviously any proper 16-clue puzzle necessarily hits in particular the unavoidable sets in this subcollection, but not conversely — a hitting set for this smaller collection may miss other unavoidable sets, of size 13 or greater. So the hitting sets for this collection are, a priori, merely *trial* (or *candidate*) sudoku puzzles, in the sense that a hitting set will not usually be a proper 16-clue

puzzle but every proper 16-clue puzzle will be found in this way. For this reason we need to perform the following extra step: each hitting set enumerated, i.e., each trial 16-clue sudoku puzzle produced, is run through a sudoku solver procedure, to see if there are multiple solutions; this effectively checks if there are any unavoidable sets in the grid having empty intersection with the hitting set in question. Thus the overall strategy we use in our programme *checker* to exhaustively search for 16-clue puzzles in a sudoku solution grid may be summarized as follows:

(C1) find a sufficiently powerful collection of unavoidable sets for the given grid;

(C2) enumerate all hitting sets of size 16 for this collection, i.e., find every possible combination of 16 clues that intersect all the unavoidable sets found in Step C1;

(C3) check if any of the hitting sets found in the previous step is a proper 16-clue puzzle, i.e., test if any of these hitting sets uniquely determine the given grid, by running each hitting set through a sudoku solver procedure.

Importantly, these three steps do not equally contribute to the running time of *checker*. It is Step C2, the enumeration of hitting sets, that typically consumes the bulk (around 95 per cent) of CPU cycles, which however should come as no surprise as the hitting set problem is NP-complete [38]. Of course, being NP-complete is an asymptotic statement, so it does not really say anything about the difficulty of a problem of a fixed size. On the other hand, experience suggests that $k = 16$ is often large enough a hitting set size for a standard backtracking algorithm to be too inefficient to solve billions of instances of the hitting set problem, as it indeed turned out. We will now provide an introductory description of Steps C1, C2 and C3; all details are given in Sections 5–7.

4.1 Unavoidable sets

This subsection provides a sketch of Step C1. The idea of an *unavoidable set* in a completed sudoku grid is easily explained by example.[5] So let us consider the following grid:

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

Figure 3: Two unavoidable sets in a sudoku solution grid

The reader can see that if the digits 5 and 9 are interchanged *among the four orange cells only* (rows 1 and 2, columns 1 and 5), then a different correctly completed grid is obtained. In other words, deleting

[5] Later we learned that the analogous concept had been defined previously in the study of Latin squares, where what we refer to as unavoidable sets are called *Latin trades*.

these four digits results in a 77-clue sudoku puzzle with two completions. Accordingly, in any puzzle with the above grid as the only possible answer, one of the four orange cells must contain a clue since a puzzle not having any of these four as a clue would have at least two completions and therefore not be a valid puzzle. We say that the set of the orange cells is *unavoidable* — we cannot avoid having a clue from it. Another example may be found in columns 6 and 8. Replacing the contents of the three green cells in column 6 with those of the three green cells in column 8, i.e., swapping the digits in the cells in the middle band in column 6 with those in column 8, again yields a different sudoku solution grid. So the six green cells (rows 4–6, columns 6 and 8) also form an unavoidable set.

In general, a subset of a sudoku solution grid is called unavoidable if it is possible to permute the contents of its cells, leaving the other cells unchanged, such that a different grid results. (We will give a more formal definition in Section 5.) Directly from this definition, if a set of clues in a grid does not intersect every unavoidable set, then it cannot be used as the set of starting clues for a sudoku puzzle since there will be more than one solution. Equivalently, any set of clues for a proper puzzle must use at least one clue from every unavoidable set. By the way, the proof that seven clues are never enough for a unique completion (given in Section 1) can be seen as follows: the $\binom{9}{2}$ subsets consisting of the eighteen occurrences of two digits in a grid are all unavoidable, and seven clues must always miss one of these sets.

Finding unavoidable sets

We find unavoidable sets in a given completed sudoku grid using a pattern-matching algorithm — in *checker*, there is a list of essentially different (with respect to the equivalence transformations defined in Section 3) types of unavoidable sets of size 12 or below; the actual search for unavoidable sets in a sudoku grid goes by effectively generating all grids equivalent to the given grid, and then comparing the list of essentially different types of unavoidable sets with each grid generated. In case of a match, the corresponding unavoidable set in the original (given) sudoku solution grid is obtained by applying the inverse of the transformation used to generate the grid that gave the match. Our *checker* of 2006 took about 30 seconds per grid to find these unavoidable sets. However, back then a lot of effort was duplicated by the procedure $\mathrm{FindUnavoidableSets}$. The logic in the new *checker* eliminates most of this redundancy (see Section 5.2), and takes less than one twentieth of a second for the same task.

How the unavoidable sets influence the running time of checker

As mentioned earlier, when searching a sudoku solution grid for 16-clue puzzles using *checker*, most of the compute time is usually spent on enumerating hitting sets for the collection of unavoidable sets. So the family of unavoidable sets — the *structure* of the grid — is what determines how computationally intensive each grid is. A typical grid has around 360 unavoidable sets with up to twelve elements. However, there is considerable variability in the structure between grids — some grids have as many as 500 unavoidable sets (of size 12 or below), while others have fewer than 200 — and thus the time it takes to search for a 16-clue puzzle also varies considerably among grids. One might think that those grids having the most unavoidable sets are the quickest to search through, but that is not actually the case. Interestingly, it turns out that grids with many unavoidable sets often have relatively few small unavoidable sets (of either size 4 or size 6), and the latter are an important factor in the running time of *checker*.

4.2 Hitting sets

In this subsection we provide a sketch of Step C2, i.e., we will outline how we enumerate hitting sets. Assume therefore that we have already found a sufficiently powerful family of unavoidable sets for a given sudoku grid. As explained in the beginning of this section, if a 16-clue puzzle exists within this grid, then it intersects (hits) each of the unavoidable sets. So we are faced with a common situation in computational discrete mathematics — we need to solve an instance of the *hitting set problem*. Recall that the hitting set problem is formally defined as follows: given a finite set U (the *universe*), a family $\mathcal{F}$ of subsets of U, and a positive integer k, the task is to find all hitting sets for $\mathcal{F}$ having k elements, where a hitting set for $\mathcal{F}$ is simply a subset of U that intersects every one of the sets in $\mathcal{F}$. In our case, the universe U is the given sudoku solution grid, the family $\mathcal{F}$ of subsets of U consists of some of the unavoidable sets for this grid, and $k = 16$ as we are interested in hitting sets having 16 elements. We reiterate that for *checker* to work as expected, i.e., for the search to be exhaustive, it is not essential to use *all* unavoidable sets; any subcollection, no matter how small, of the collection of all unavoidable sets will produce correct results, by Step C3. For the actual computations we of course worked with a collection that minimizes the running time of *checker* — it turned out that the family of unavoidable sets with at most twelve elements gives the overall best results.

So we needed an efficient hitting set algorithm for this project. In the original version of *checker* we used the obvious backtracking algorithm, modified such that every hitting set is enumerated once only; full details of that are given in Section 6. The hitting set algorithm in the new *checker* has three main improvements over the original algorithm, which are explained in Section 7. The most important of these three improvements will be introduced now. It is a technique to prune the search tree so as to abandon the traversal of a particular branch earlier than usual — with a standard hitting set algorithm, detection of failure, and hence backtracking, does not begin until after the last (k^{th}) element has been added to the hitting set under construction.

Pruning the search tree using higher-degree unavoidable sets

There is a natural generalization of the concept of an unavoidable set in a sudoku solution grid, which we call higher-degree unavoidable set — a *higher-degree* unavoidable set is a subset of the given grid from which more than one clue (say m clues) is needed for a uniquely completable puzzle; the *degree* of such a set is m. For example, an unavoidable set of degree 3 is an unavoidable set such that every puzzle in the grid in question must have at least three elements from that set in order to have a unique solution. So suppose that, when enumerating hitting sets of size $k > 1$ for a family $\mathcal{F}$ of unavoidable sets in a sudoku grid G, we are further given families $\mathcal{F}^{(2)}, \ldots, \mathcal{F}^{(k)} \subset 2^G$ such that each member of $\mathcal{F}^{(d)}$ is unavoidable of degree d, $d = 2, \ldots, k$. We will now show how these families of higher-degree unavoidable sets provide a very simple way to facilitate earlier backtracking, thereby speeding up the algorithm quite significantly. Recall that, in general, backtracking begins as soon as it is clear that the solution being constructed (partial solution) cannot possibly be completed to an actual solution to the problem in question.

Assume that $k - 1$ clues have been selected already, i.e., suppose that $k - 1$ elements have so far been added to the hitting set under construction. If there exists a member of $\mathcal{F}^{(2)}$ (an unavoidable set of degree 2) not containing any of the $k - 1$ clues of our partial hitting set, then we may immediately stop and backtrack — we already know that this $(k - 1)$-clue set can never be completed to a k-clue

hitting set, because any unavoidable set of degree 2 requires by definition at least two clues but there is only one more clue left to draw. (In other words, no matter how the k^{th} clue is chosen, the resulting puzzle will have multiple completions.) Similarly, if after selecting $k-2$ clues there is a set in $\mathcal{F}^{(3)}$ that does not contain any of the clues picked so far, then we can again backtrack immediately as each element of $\mathcal{F}^{(3)}$ is an unavoidable set of degree 3 and therefore requires at least three clues; with only two clues to go, there is no point continuing. More generally, if $k-d+1$ clues have been drawn for some d, $1 < d \leq k$, and there is a set in $\mathcal{F}^{(d)}$ that does not contain any of these clues, i.e., if there is an unavoidable set of degree d not yet hit, then we do not need to branch any further but may instead backtrack at once.

How do we actually obtain higher-degree unavoidable sets? By taking disjoint unions of ordinary unavoidable sets. As an example, consider once again the grid shown at the beginning of Section 4.1. In this grid, two unavoidable sets are highlighted, one having four elements (the orange cells), and the other having six elements (green cells). Note that these two unavoidable sets are disjoint, so that it is impossible to hit both with just a single clue. Therefore at least two clues are required, i.e., the union of these two sets is an unavoidable set of degree 2. We call a pair of disjoint unavoidable sets a *clique of size 2*. The general case is if there are $d > 1$ pairwise disjoint unavoidable sets (a *clique of size* d); at least d clues are then necessary to hit them all, and thus the union of these d sets is an unavoidable set of degree d. Hence, after *checker* has found some degree 1 unavoidable sets, all we need to do to obtain a sizeable collection of higher-degree unavoidable sets is search for cliques. Say ℓ unavoidable sets of degree 1 have been found. Then for each $d = 2, \ldots, k$, where k is again the size of the desired hitting set, we simply consider all $\binom{\ell}{d}$ combinations of d unavoidable sets and check for each if the d unavoidable sets in question are pairwise disjoint. If so, then we have found a clique of size d, and so we just take the union of these d unavoidable sets to produce a degree d unavoidable set. In practice, it turns out to be optimum to use cliques of size 2, 3, 4 and 5 only, so that detection of failure does not happen until at least twelve clues have been picked.

It may be surprising that such a simple observation can actually make an important difference, but with *checker*, on the algorithmic side (as opposed to the implementation side), this was really the key ingredient in making the exhaustive search for a 16-clue sudoku puzzle feasible.[6] Finally, we refer to these cliques as *trivial* higher-degree unavoidable sets. Actually, there are also examples of *nontrivial* higher-degree unavoidable sets, e.g., unavoidable sets of degree 2 that are not merely the union of two disjoint unavoidable sets of degree 1, see Section 5.3. Owing to time constraints, however, we did not fully investigate the usefulness of such nontrivial higher-degree unavoidable sets, and in the end these did not play a very important role in the computation.

4.3 Running all 16-clue candidate puzzles through a sudoku solver

In this subsection we explain Step C3. We required a sudoku solver module (procedure) for *checker*, because we had to test all hitting sets found for a unique completion, the reason being that we worked with a subcollection of the collection of all unavoidable sets. To avoid doing something in a mediocre way that others had already excelled at, we did not actually develop our own sudoku solver. Instead,

[6] "In a [sudoku] puzzle, what you haven't looked at yet is probably where you're meant to make progress. And in science it's what you haven't looked at yet often times." —Thomas Snyder (a.k.a. *Dr. Sudoku*), three-times sudoku world champion, explaining similarities between solving a sudoku and carrying out scientific research (Source: http://www.youtube.com/watch?v=WMNal53nBtE from 3:25 to 4:18)

in the original *checker*, we used the public domain solver *suexk* written by Guenter Stertenbrink [41]. However, for the new version of *checker*, we switched to the open-source solver *BBSudoku*, by Brian Turner [42]. At the time we had to decide which solver to use, Turner's was the fastest one available, to our knowledge. Note that Stertenbrink's algorithm is based on an exact cover problem solver, while Turner's approach is to solve a sudoku puzzle much like a human would: at every step, the idea is to first look for naked singles, then hidden singles, then locked candidates, then to check for an X-Wing, Swordfish, etc., and finally to make a guess, i.e., perform trial and error. For the actual computations, we slightly modified Turner's solver. We removed all of the advanced solving methods in order that the logic would be as simple as possible. The modified solver consisted of only about 150 statements, yet was capable of checking over 50,000 16-clue trial puzzles for a unique completion per second, a testament to Turner's design. In fact, not making use of the advanced solving techniques produces the fastest possible version of this solver, as the author also remarks in a comment in the source code.

5 Some theory of unavoidable sets

In Section 4.1 we gave a brief introduction to unavoidable sets, and outlined how we go about finding them in a given completed sudoku grid. We said that the algorithm in the original version of *checker* did a lot of redundant work compared with the new *checker*. We will now make all this precise, and in particular we explain how investigating the theory of unavoidable sets makes it possible to remove the aforementioned redundancy relatively easily.

5.1 Definition and elementary properties

Throughout this section, a sudoku solution grid will formally be a function

$$\{0, \dots, 80\} \to \{1, \dots, 9\},$$

and when we say "let X be a subset of a sudoku solution grid G" we identify G with the corresponding subset of the cartesian product $\{0, \dots, 80\} \times \{1, \dots, 9\}$.

Definition. Let G be a sudoku solution grid. A subset $X \subset G$ is called an *unavoidable set* if $G \backslash X$, the complement of X, has multiple completions. An unavoidable set is said to be *minimal* if no proper subset of it is itself unavoidable.

So if a subset of a sudoku solution grid does not intersect every unavoidable set contained in that grid, then it cannot be used as a set of clues for a sudoku puzzle because there will be more than one solution. In other words, a set of clues for a valid puzzle *must* contain at least one clue from every unavoidable set. In fact, the converse is true as well:

Lemma 2. *Suppose that $X \subset G$ is a set of clues of a sudoku solution grid G such that X hits (intersects) every unavoidable set of G. Then G is the only completion of X.*

Proof. If X had multiple completions, then $G \backslash X$ would be an unavoidable set not hit by X, which is a contradiction. □

In the original version of *checker*, unavoidable sets in a given sudoku solution grid were found using a straightforward pattern-matching algorithm. More specifically, *checker* contained several hundred different *blueprints*, where a blueprint is just a representative of an equivalence class of (minimal) unavoidable sets, the equivalence relation again being the one from Section 3.1. On the online forums, Ed Russell had investigated unavoidable sets and compiled a list of blueprints, which he sent us. We added all blueprints of size 12 or less from Russell's list to *checker* (525 blueprints in total). When actually finding unavoidable sets in a completed sudoku grid, *checker* would compare each blueprint against all grids in the same equivalence class as the given grid, modulo the digit permutations. That is, *checker* would generate 3,359,232 grids equivalent to the original grid, as explained in Section 3, and for every grid generated *checker* would try each blueprint for a match. In total, this yields around 360 unavoidable sets with a typical grid.

Using the algorithm just described, finding unavoidable sets takes approximately 30 seconds per grid. In 2006 we did not consider this a problem because back then *checker* took over an hour on average to scan a grid for all 16-clue puzzles, so that searching the entire sudoku catalog was completely out of question anyway. However, once we succeeded in efficiently enumerating the 16-clue hitting sets, in order to make this project feasible we also had to come up with a faster algorithm for finding the unavoidable sets. While we kept our original pattern-matching strategy, a bit of theory helped us to significantly reduce the number of possibilities to check. Before we can start developing the theory of unavoidable sets, we need one more definition.

Definition. Let G be a sudoku solution grid, and let $X \subset G$. We say that a digit d, $1 \leq d \leq 9$, appears in X if there exists $c \in \{0, \ldots, 80\}$ such that $(c, d) \in X$. Similarly we say that a cell c is contained in X if $(c, d) \in X$ for some d.

Lemma 3. *Let G be a sudoku solution grid and suppose that $U \subseteq G$ is a minimal unavoidable set. If H is any other completion of $G \backslash U$, $H \neq G$, then G and H differ exactly in the cells contained in U. In particular, every digit appearing in U occurs at least twice.*

Proof. If G and H agreed in more cells than those contained in $G \backslash U$, then U would not be minimal as it would properly contain the unavoidable set $G \backslash (G \cap H)$. So when moving between G and H, the contents of the cells in U are permuted such that the digits in all cells of U change. If there was a digit d contained in only one cell of U, that digit could neither move to a different row nor to a different column, since otherwise the row respectively column in question would not contain the digit d anymore at all. That is, the digit d stays fixed, in contradiction to what we just noted. □

For our first corollary, recall that a *derangement* is a permutation with no fixed points, i.e., an element $\sigma \in S_n$, $n \in \mathbb{N}$, such that $\sigma(k) \neq k$, for all $k = 1, \ldots, n$.

Corollary 4. *Let G be a sudoku solution grid and suppose that $U \subseteq G$ is a minimal unavoidable set. If H is any other completion of $G \backslash U$, $H \neq G$, then H may be obtained from G by a derangement of the cells in each row (column, box) of U. Hence the intersection of U with any row (column, box) is either empty or contains at least two elements.*

Proof. Follows directly from the last lemma and since the rules of sudoku would be violated otherwise. □

5.2 Finding minimal unavoidable sets efficiently using blueprints

The basic idea how to make the actual search for minimal unavoidable sets in a given sudoku solution grid faster is to replace the blueprints from Russell's list by appropriate members of the same equivalence class. These members are chosen such that only a fraction of grids equivalent to the given one need to be checked for a match. We will call a blueprint an $m \times n$ blueprint if it hits m bands and n stacks of the 9×9 matrix, and we treat the blueprints according to the number of bands and stacks they hit. By taking the transpose if necessary, it is no loss of generality to assume that $m \leq n$.

Suppose that $m = 1$, i.e., suppose that a blueprint hits only a single band. First observe that by Lemma 3, a 1×1 blueprint cannot exist, as any digit in a minimal unavoidable set appears at least twice. So $n \geq 2$. By swapping bands if necessary, we may then assume that (up to equivalence) only the top band is hit. Moreover, after possibly permuting some of the rows and/or columns, we may in fact assume that two of the cells of the blueprint are as follows, again because any digit appearing in a minimal unavoidable set occurs at least twice:

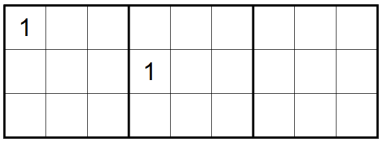


Figure 4: Two digits in any $1 \times n$ blueprint

For a 1×3 blueprint we further choose, if possible, a representative that has no cells in either the middle or the right column of the right stack.

When actually searching for instances of 1×2 blueprints in a given grid, i.e., when generating those representatives of the given grid that need to be considered in order to find all occurrences of a 1×2 blueprint, there will be three possibilities for the choice of top band as well as six permutations of the three rows within the top band (once a band has been chosen as the top band). So in total there are 18 different arrangements of the rows. Compare this to 1,296 row permutations with the original algorithm. For the columns, a priori, we have to consider all six possible arrangements of the three stacks, as well as all six permutations of the columns in the left stack. However, the majority of 1×2 blueprints have a stack containing cells in only one column of that stack. Therefore, if we choose such a stack as the middle stack, then only the left column in the middle stack will contain cells of the blueprint. So we do not in fact consider all six permutations of the three columns in the middle stack; rather, we try each of the three columns as the left column once only, which means that we use only one (random) arrangement of the two remaining columns in the middle stack.

Of course this will, with probability 0.5, miss instances of those blueprints that have digits in the other two columns of the middle stack, which is why all such blueprints are actually contained twice in *checker*, with the respective columns swapped. Perhaps this is best explained by the following example of a 1×2 blueprint of size 10, which is saved twice in *checker*'s table of blueprints:

Figure 5: A 1×3 blueprint of size 10 that occurs twice in *checker*'s list of blueprints

In total, 108 different permutations of the columns will be considered. However, only in one out of six cases do we actually need to match all the 1×2 blueprints against the corresponding grid, since all our blueprints have the same digit in the top-left cell and in the fourth cell of the second row as explained at the beginning of this subsection (see Figure 4). Summing up the above discussion, in order to find all 1×2 unavoidable sets in a given sudoku solution grid, instead of having to generate 3,359,232 grids, in actual fact we only need to generate 648 grids.

We search for 1×3 unavoidable sets in a very similar manner; the only additional effort required is that we also need to permute the columns in the right stack of the grid. Like with the middle stack, we do not actually try all six permutations, but only three permutations — each of the three columns in the right stack is selected as the left column exactly once. Since this would again miss half of those unavoidable sets having clues in multiple columns of the right stack, the respective blueprints also appear twice in *checker*. In particular, 1×3 blueprints having clues in multiple columns of *both* the middle *and* the right stack actually appear four times in *checker*'s table of blueprints, e.g., this one here of size 12:

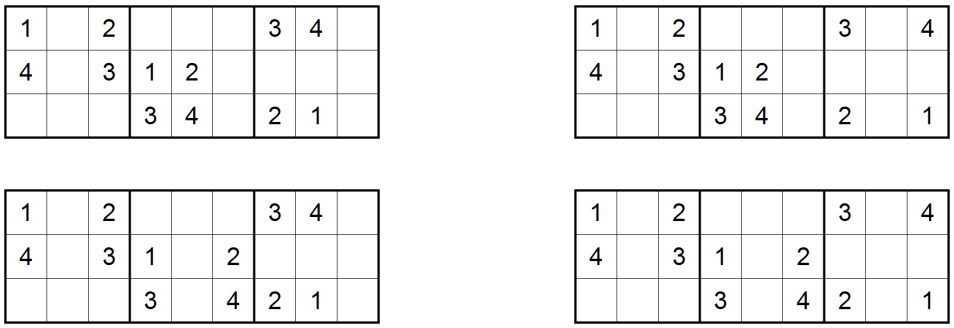
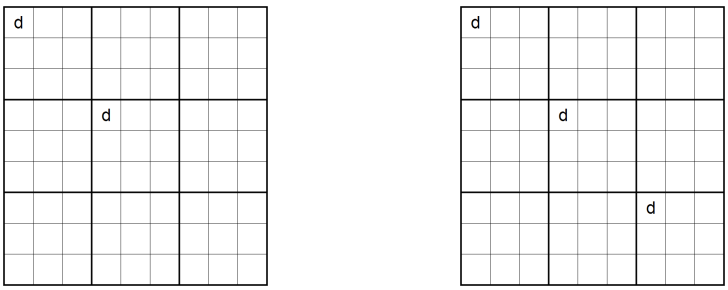


Figure 6: A 1×3 blueprint of size 12 that occurs four times in *checker*'s list of blueprints

In order to tackle 2×2, 2×3 and 3×3 blueprints, we need the following result, whose proof is quite long (although not very difficult).

Proposition 5. *Every blueprint is equivalent to one containing the same digit twice in the same band.*

Proof. First note that for $1 \times n$ blueprints there is nothing to prove as the claim follows directly from the fact that any digit in a minimal unavoidable set appears at least twice, see Lemma 3. So assume that we are given an $m \times n$ blueprint B such that $n \geq m \geq 2$. We begin by showing that there is either a band or a stack containing the same digit twice. Let d be any digit occurring in B. Suppose for the sake of contradiction that d does not appear twice in either the same band or the same stack. Without loss of generality, the only two possibilities for *all* occurrences of d are therefore as follows, recalling that d appears at least twice:



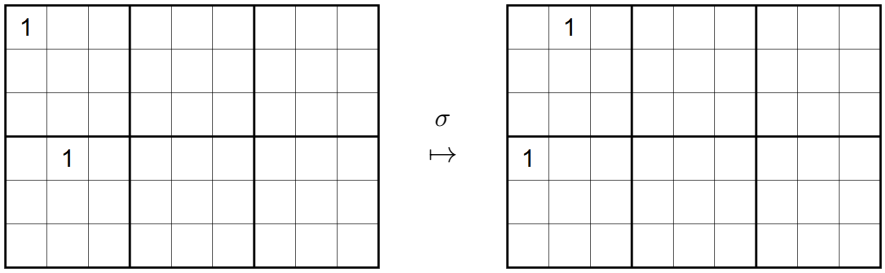
By Corollary 4, if we number the cells in a grid from 0 to 80 inclusive, left-to-right and top-to-bottom, then there exists a permutation $\sigma\colon \{0,\dots,80\} \to \{0,\dots,80\}$ such that

$$\begin{cases} \sigma(c) \neq c \text{ but } \operatorname{box}(\sigma(c)) = \operatorname{box}(c), & c \in B, \\ \sigma(c) = c, & c \notin B, \end{cases}$$

and whose application to the blueprint B respects the rules of sudoku. Here, the function box evaluates to the index of the 3×3 box the given cell belongs to. (Formally, $\operatorname{box}\colon \{0,\dots,80\} \to \{0,\dots,8\}$ is defined by $c \mapsto \lfloor c/27 \rfloor \times 3 + \lfloor c \pmod 9 /3 \rfloor$.) But then in both of the above cases, by the properties of σ just stated, any instance of the digit d would have to stay fixed as the cell containing it is only allowed to move inside its box, but if that cell actually left either its original row or column, the rules of sudoku would be violated, which is the desired contradiction. In detail, if a cell of B containing the digit d left, e.g., its row under σ, then the row in question would lack d, as any other appearance of d that might compensate for the "loss" of the original instance of d in the affected row would have to be contained in the same band, because, after applying σ, all cells are still in the same box as they originally were. Similarly if a cell of B containing d left its original column under σ.

This proves our first claim, that d necessarily appears twice in either the same band or the same stack. If there is a band containing the digit d twice, we are done. Otherwise, for a 2×2 or a 3×3 blueprint we simply take the transpose, and again the claim is proven. We are left with the case that B is a 2×3 blueprint such that every digit appears exactly twice in the same stack — by the pigeon hole principle, no digit of B can appear three or more times, because then at least one of the bands that B contains cells of would have that digit multiple times.

Without loss of generality, two digits of B are as shown on the left; the respective digits of $\sigma(B)$ are then necessarily as shown on the right:



This illustrates a general principle — when applying σ to B, *every* cell of B stays in its original row, once again because no digit appears twice in the same band of B and as all cells of B stay in their original boxes under σ. Hence if d is any digit appearing in B and if c and c' are the two cells of B containing d, then

$$\begin{array}{rclcrclcrcl} \operatorname{box}(\sigma(c)) & = & \operatorname{box}(c), & \qquad & \operatorname{row}(\sigma(c)) & = & \operatorname{row}(c), & \qquad & \sigma(c) & \neq & c, \\ \operatorname{box}(\sigma(c')) & = & \operatorname{box}(c'), & & \operatorname{row}(\sigma(c')) & = & \operatorname{row}(c'), & & \sigma(c') & \neq & c'. \end{array}$$

It now follows immediately that

$$\operatorname{col}(\sigma(c)) \;=\; \operatorname{col}(c') \qquad \text{and} \qquad \operatorname{col}(\sigma(c')) \;=\; \operatorname{col}(c), \tag{3}$$

since the rules of sudoku would be violated otherwise. (The functions row and col are defined in the obvious way.) We summarize the above observations as follows.

Every digit appearing in B occurs exactly twice, and both instances are in the same stack; moreover, every cell of B stays in its original row and box under σ but moves to the original column of the other cell of B containing the same digit.

We will complete the proof by showing that this contradicts minimality of B — recall that any blueprint by definition is an instance of a *minimal* unavoidable set. In brief, it suffices to notice that the subset of cells of B contained in the left stack, which is a proper subset of B because B also has cells in the other two stacks of the grid, already is itself unavoidable. We shall now make this precise. To this end, let G be any completion of B, and let $G^{(0)}$, $G^{(1)}$ and $G^{(2)}$ denote the three stacks of G. That is to say, for each $s = 0, 1, 2$, we think of $G^{(s)}$ as a function

$$\{9i + j + 3s \mid i = 0, \ldots, 8, j = 0, 1, 2\} \;\to\; \{1, \ldots, 9\},$$

and we again identify $G^{(s)}$ with the corresponding subset of $\{0, \ldots, 80\} \times \{1, \ldots, 9\}$. Recall that in the beginning of the proof we chose σ such that $\mathrm{box}(\sigma(c)) = \mathrm{box}(c)$, for all $c = 0, \ldots, 80$. So when σ is applied to G, any cell of the grid stays in its original box, in particular, any cell stays in its original stack. Therefore, the set $\sigma(G^{(s)}) = \{(\sigma(c), d) \mid (c, d) \in G^{(s)}\}$ is still the stack of index s (of a different grid), for each $s = 0, 1, 2$, and so there is no ambiguity to set

$$H \;=\; \sigma(G^{(0)}) \cup G^{(1)} \cup G^{(2)}.$$

Note that H is a valid sudoku solution grid:

- By the above, no cell of B moves to a different row under σ (recall also that cells in $G \backslash B$ are fixed by σ anyway). Therefore, if d_1, d_2, d_3 are the three digits contained in any row of the stack $G^{(0)}$, then the corresponding row of the stack $\sigma(G^{(0)})$ will also contain the same three digits d_1, d_2, d_3, though possibly in a different order. In any case, each row of H contains every number from 1 to 9 exactly once.

- Although some cells of $G^{(0)}$ do change columns when σ is applied, every column of H nonetheless contains only distinct entries — as if $c \in G^{(0)}$ is such that $\mathrm{col}(\sigma(c)) \neq \mathrm{col}(c)$, then $c \in B$ and from (3) there exists $c' \in G^{(0)}, c' \neq c$, such that the cells c and c' carry the same digit and $\mathrm{col}(\sigma(c')) = \mathrm{col}(c)$, i.e., c' acts as the replacement for c in the original column of c (and conversely).

- Because σ permutes the cells in each box, it is immediately clear that every box of $\sigma(G^{(0)})$ still contains nine different numbers.

Observe now that the set $U := G \backslash (G \cap H)$ is a proper subset of B that is in fact unavoidable. First, U is a subset of B as, trivially, $G \supseteq G \backslash B$, and also $H \supseteq G \backslash B$ since σ fixes cells not contained in B. So $G \cap H \supseteq G \backslash B$, and taking complements we therefore get $U \subseteq B$. Second, U is a *proper* subset of B because $G \cap H \supseteq G^{(1)} \cup G^{(2)}$ but $B \cap (G^{(1)} \cup G^{(2)}) \neq \emptyset$, recalling that B contains cells of the middle and the right stack. Finally, directly from the definition, U is unavoidable as its complement has two different completions, namely G and H. This completes the proof of Proposition 5. □

So for *any* blueprint it is no loss of generality to assume that two digits in the top band are as shown in Figure 4, not just for $1 \times n$ blueprints. The actual algorithm used when searching for 2×2, 2×3, and 3×3 unavoidable sets is very similar to the one for $1 \times n$ unavoidable sets — the only difference

is that we further arrange for each blueprint to be of one of the following three types, again to reduce the number of possibilities to check (note that the first and the third type overlap):

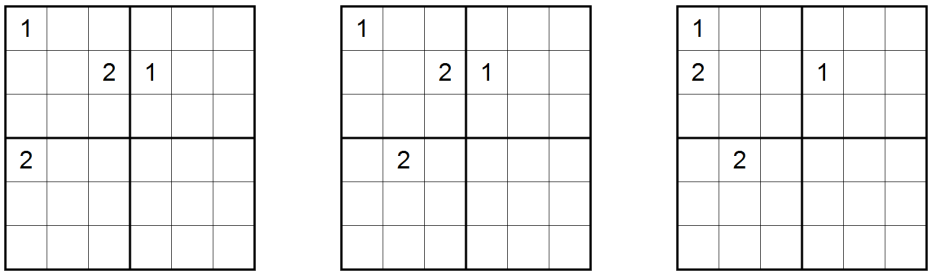


Figure 7: The three types of $m \times n$ blueprint with $m \geq 2$ (only the top-left 6×6 subgrid is shown)

With the above ideas implemented, finding all unavoidable sets of size up to 12 in a sudoku solution grid takes less than 0.05 seconds on average.

5.3 Higher-degree unavoidable sets

This section gives the details of the theory of higher-degree unavoidable sets introduced in Section 4.2. There are unavoidable sets that require more than one clue, which we call *higher-degree* unavoidable sets. Let us illustrate this with an example.

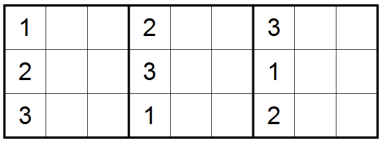


Figure 8: An unavoidable set of degree 2

Note that two clues are needed from the nine digits shown in order to completely determine these nine cells. Because, if only one is given, the other two digits may be interchanged. These nine cells form an unavoidable set requiring two clues. Technically, the above nine clues are really the union of nine minimal unavoidable sets of size six each, and the intersection of these nine minimal unavoidable sets is empty, hence one clue is not enough to hit all of them. Thus the above is an example of a degree 2 unavoidable set. If we say that an unavoidable set as defined earlier (see the definition on p. 18) is an unavoidable set of degree 1, then we may recursively define the notion of an unavoidable set of degree $k > 1$.

Definition. A nonempty subset U of a sudoku solution grid G is said to be an *unavoidable set of degree* $k > 1$ if for all $c \in U$ the set $U \backslash \{c\}$ is an unavoidable set of degree $k - 1$.

There is the following alternative characterization of higher-degree unavoidable sets.

Lemma 6. *Let G be a sudoku solution grid and let $k \in \mathbb{N}$. A nonempty subset $U \subseteq G$ of size n is unavoidable of degree k if and only if for all $\binom{n}{k-1}$ choices of distinct elements $c_1, \ldots, c_{k-1} \in U$, the set $U \backslash \{c_1, \ldots, c_{k-1}\}$ is unavoidable.*

Proof. Induction on k. For $k = 1$ the result just says that U is unavoidable of degree 1 if and only if U is unavoidable as defined earlier, so there is nothing to prove. Suppose now that $k > 1$ and the result is true for smaller values of k. Assume that U satisfies the hypothesis of the "if"-direction and let $c \in U$ be an arbitrary element; we need to show that $U\backslash\{c\}$ is unavoidable of degree $k-1$. Using induction it suffices if $(U\backslash\{c\})\backslash\{c_1, \ldots, c_{k-2}\} = U\backslash\{c, c_1, \ldots, c_{k-2}\}$ is unavoidable for all possible combinations of distinct $c_1, \ldots, c_{k-2} \in U\backslash\{c\}$, which however is true by assumption.

For the "only if"-direction suppose that U is unavoidable of degree k and let $c_1, \ldots, c_{k-1} \in U$ be distinct. By definition, $U\backslash\{c_1\}$ is then unavoidable of degree $k-1$, and from induction it thus follows immediately that $U\backslash\{c_1, \ldots, c_{k-1}\} = (U\backslash\{c_1\})\backslash\{c_2, \ldots, c_{k-1}\}$ is unavoidable, as required. □

As before, we say that an unavoidable set of degree greater than 1 is *minimal* if no proper subset is unavoidable of the same degree. Furthermore, to ease notation, we will say that U is an (m, k) unavoidable set if U is an unavoidable set of degree k having m elements. So the example in Figure 8 is a $(9, 2)$ unavoidable set that is the union of nine $(6, 1)$ unavoidable sets. One can easily construct higher-degree unavoidable sets, e.g., the union of any two disjoint degree 1 unavoidable sets is trivially an unavoidable set of degree 2. More generally, we have the following result.

Proposition 7. *Let $U \subset G$ be an (m, k) unavoidable set. If $V \subset G$ is an (n, ℓ) unavoidable set such that $U \cap V = \emptyset$, then $U \cup V$ is an $(m + n, k + \ell)$ unavoidable set.*

Proof. The claim follows directly from the last lemma. □

Repeated application of this proposition gives the following useful fact.

Corollary 8. *Suppose that $U_1, \ldots, U_t$ are unavoidable sets of a sudoku solution grid G of degree $k_1, \ldots, k_t$, respectively. Assume further that the U's are pairwise disjoint. Then $U_1 \cup \cdots \cup U_t$ is an unavoidable set of degree $k_1 + \cdots + k_t$.*

Definition. An unavoidable set U of degree $k > 1$ is said to be *nontrivial* if there does not exist an unavoidable set U_1 of degree k_1 and an unavoidable set U_2 of degree k_2 and disjoint from U_1, $k_1, k_2 \in \mathbb{N}$, such that $U = U_1 \cup U_2$ and $k = k_1 + k_2$; otherwise, we say that U is *trivial*.

So the $(9, 2)$ unavoidable set shown in Figure 8 is nontrivial. In actual fact, it is one of only two types of nontrivial $(9, 2)$ unavoidable sets, the other being this one here:

1			2			3		
2			1			4		
3			4			1		

Figure 9: A nontrivial (9,2) unavoidable set

Earlier we said that, so far, nobody has found a purely mathematical proof that 8 clues are not sufficient for a sudoku puzzle to have a unique solution. However, if one assumes that the sudoku solution grid in question has a nontrivial (9,2) unavoidable set (of either type), then it is actually easy to see why such a grid cannot contain a proper 8-clue puzzle.

We classified all minimal $(m, 2)$ unavoidable sets for $m \leq 11$. The result was that no $(m, 2)$ unavoidable sets exist for $m \leq 7$. While $(8, 2)$ unavoidable sets do exist, all these are trivial, i.e.,

any $(8,2)$ unavoidable set is the union of two disjoint $(4,1)$ unavoidable sets. Similarly, minimal $(10,2)$ unavoidable sets exist, but again, all these are trivial, i.e., the disjoint union of a $(4,1)$ and a $(6,1)$ unavoidable set. There are seven distinct types of nontrivial minimal $(11,2)$ unavoidable sets, which however we did not use in this project — observe that any minimal $(11,2)$ unavoidable set is necessarily nontrivial as there are no minimal $(m,1)$ unavoidable sets for $m = 1, 2, 3, 5, 7$. Naturally we have the following result.

Proposition 9. *Let $U \subset G$ be an (m,k) unavoidable set. Then we need to add at least k elements from U to $G\backslash U$ in order to obtain a sudoku puzzle with a unique completion.*

Proof. If we add only up to $k-1$ clues from U to the puzzle $G\backslash U$, then there will still be multiple completions, since by Lemma 6 and the assumption that U is an unavoidable set of degree k, the set $U\backslash\{c_1,\ldots,c_{k-1}\}$ is unavoidable for any choice of $c_1,\ldots,c_{k-1} \in U$. □

This last proposition is really the key result on the theory of unavoidable sets as far as the problem of making the enumeration of hitting sets more efficient is concerned, see Section 7.

Example

We close this section with the following example of a sudoku grid that requires 18 clues, at least. We can prove this fact purely mathematically using unavoidable sets of degree 2.

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

Figure 10: A grid requiring at least 18 clues

Lemma 10. *Any sudoku puzzle whose solution is this grid has at least* 18 *clues.*

Proof. Upon inspection, the grid is the union of nine pairwise disjoint $(9,2)$ unavoidable sets, as indicated by the different shadings. From Corollary 8, this grid is therefore an $(81,18)$ unavoidable set and thus requires at least 18 clues for a unique solution, by the last proposition. □

We note that some choices of 18 clues lead to a unique solution, but some do not. We also remark that results like this lemma are impossible to prove by hand for a general grid. It is possible to prove the lemma by hand only because this grid is highly structured.

6 Hitting set algorithm from the original checker

In this section we will explain how the original (2006) version of *checker* enumerated the hitting sets for a given family of unavoidable sets in a sudoku solution grid. Section 6.1 describes how we implemented the obvious hitting set algorithm, and Section 6.2 explains the one improvement we made. Formally, the hitting set problem is as follows: given a triple $(U, \mathcal{F}, k)$, where U is a finite set (the *universe*), $\mathcal{F} \subset 2^U$ is a family of subsets of U, and k is a positive integer, the task is to find all subsets $H \subset U$, $\#H = k$, such that $H \cap S \neq \emptyset$, for all $S \in \mathcal{F}$. Each set H is called a *hitting set* for $\mathcal{F}$.

The hitting set problem is NP-complete [38]. And indeed, the algorithm presented in this section, despite already having the advantage over a standard backtracking algorithm that each hitting set is enumerated once only, would not have been fast enough to solve the sudoku minimum number of clues problem using a reasonable amount of compute time until about the year 2020. Only with the improvements described in Section 7 was this project feasible in 2011. We should also mention that a common way to approach the hitting set problem is actually a greedy algorithm. However, as the result a greedy algorithm produces is only approximate there would have been no sense applying such an algorithm to the sudoku minimum number of clues problem.

6.1 Description of the basic algorithm

For our original version of *checker* from 2006 we essentially used the obvious algorithm for finding hitting sets, with one small, but powerful, improvement. The strategy of the basic algorithm may be described in one paragraph. Given a completed sudoku grid as well as a collection of unavoidable sets for that grid, candidate 16-clue puzzles are constructed by recursively adding clues from unavoidable sets. Whenever the next clue is added to the puzzle being built up to, one first finds an unavoidable set that does not contain any of the clues picked so far and then branches in all possible ways. (So if that unavoidable set has d clues, there will be d branches.) Repeat until 16 clues have been picked. If all unavoidable sets in the given collection have been hit before the 16^{th} clue is reached, the remaining clues needed are added in all possible ways. It is also understood that at each stage, in order to minimize the number of branches, one always chooses a smallest unavoidable set not yet hit.

The data structure we used to accomplish the above was as follows. Suppose that there are m members in our family $\mathcal{F}$ of unavoidable sets. Say $\mathcal{F} = \{U_0, \ldots, U_{m-1}\}$ where each $U_i \subset \{0, \ldots, 80\}$ and $\#U_i \leq \#U_{i+1}$. Then for every clue $c \in \{0, \ldots, 80\}$ there is a binary vector of length m, called the *hitting vector for* c and which we will denote $\mathrm{hitvec[c]}$, whose i^{th} slot is given by $\mathcal{X}_{U_i}(c)$, where for any subset $S \subset \{0, \ldots, 80\}$ the function $\mathcal{X}_S \colon \{0, \ldots, 80\} \to \{0, 1\}$ is defined by

$$\mathcal{X}_S(s) = \begin{cases} 1, & s \in S, \\ 0, & s \notin S. \end{cases}$$

So $\mathcal{X}_S$ is just the usual *characteristic function* (or *indicator function*) for S with respect to $\{0, \ldots, 80\}$, and the i^{th} slot of $\mathrm{hitvec[c]}$ records whether or not the clue c is contained in the unavoidable set U_i. Here is pseudocode for the procedure that sets up the hitting vectors:

```
InitHittingVectors(U,m)    // U = array of unavoidable sets, m = number of sets
  create array hitvec[0..80]
  for c from 0 to 80
    do hitvec[c] := (X_{U[0]}(c),...,X_{U[m-1]}(c))
```

We store the hitting set being constructed in the array hitset. When enumerating the hitting sets, we need to keep track of which unavoidable sets have been hit already. Therefore, there is another array of binary vectors of length m, $\mathrm{statevec}[0..16]$. Initially we set $\mathrm{statevec}[0] := (0, \ldots, 0)$, i.e., $\mathrm{statevec}[0]$ is the zero vector. If we add the clue c at the j^{th} step, $0 \leq j \leq 15$, then we simply set

```
hitset[j+1]   := c
statevec[j+1] := statevec[j] OR hitvec[c]
```

That is, we first save the clue c to the array hitset and then perform componentwise (bitwise) boolean OR on the binary vectors $\mathrm{statevec[j]}$ and $\mathrm{hitvec[c]}$ and store the result in $\mathrm{statevec[j+1]}$. Therefore $\mathrm{statevec[j+1]}$ contains a 1 in the i^{th} slot if and only if either $\mathrm{statevec[j]}$ or $\mathrm{hitvec[c]}$ has a 1 in the i^{th} slot, which is so if and only if either the set U_i was already hit, or if $c \in U_i$. We recursively do this until $j = 16$ or $\mathrm{statevec[j]} = (1, \ldots, 1)$. In the latter case, i.e., if at some stage all unavoidable sets in our collection have been hit, we add $16 - j$ more clues to the hitting set in all possible ways.

6.2 Using the dead clue vector to prevent multiple enumeration of hitting sets

We describe here the one improvement (over the obvious hitting set algorithm) that we used in the original release of *checker*. It addresses one clear shortcoming of the above, naive, algorithm, namely the fact that this algorithm will enumerate most candidate 16-clue puzzles multiple times. For example, suppose that the first three sets in our family $\mathcal{F}$ of unavoidable sets to be hit are

$$\begin{aligned} U_1 &= \{0, 3, 9, 12\}, \\ U_2 &= \{0, 1, 27, 28\}, \\ U_3 &= \{3, 4, 66, 67\}. \end{aligned}$$

When we choose $0 \in U_1$ as the first clue of the hitting set under construction, then U_2 is also hit automatically, so we will use U_3 as the set for drawing the second clue from. The first element in U_3 is 3, so one possibility for the first two clues is $\{0, 3\}$. On the other hand, when the algorithm later chooses $3 \in U_1$ as the first clue of the hitting set, then U_2 is still unhit, so will be used for drawing the second clue from. However $0 \in U_2$, so again one possibility for the first two clues is $\{0, 3\}$. The following illustration shows the search tree for the example just given:

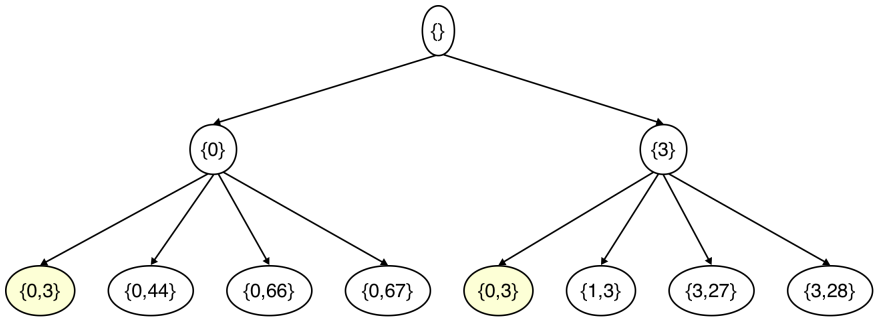


Figure 11: Sample search tree produced by a standard backtracking algorithm

To avoid duplicating work as in the example just given, we incorporated the *dead clue vector* into the original *checker* to ensure that every 16-clue puzzle is enumerated once only. The basic idea is that,

whenever we add a clue to the hitting set from an unavoidable set, we consider all smaller clues from that unavoidable set as *dead*, i.e., we exclude these smaller clues from the search (in the respective branch of the search tree only). We again use a binary vector, of length 81, to keep track of which clues are dead, and whenever we add a clue, we check immediately beforehand whether or not that clue was already excluded earlier. Going back to the example just given, when we choose $3 \in U_1$ as the first element of the hitting set, then we also mark the element $0 \in U_1$ as dead. The next unavoidable set not hit is U_2, and normally we would now try 0 as the second clue in the hitting set. However we excluded 0 as a possible future clue in the previous step, so there will be only three branches in this case. Only 1, 27, and 28 are actually tried as the second clue of the hitting set being constructed. The pseudocode for the procedure that initially sets up the required binary vectors is as follows:

```
Initialize(U,m)            // U = array of unavoidable sets, m = number of sets
  for i from 0 to m-1      // first set up vectors needed for the dead clues
    do for each c in U[i]
         do U[i].deadclues[c] := (0,...,0)
            for each d in U[i]
              do if d <= c
                   then U[i].deadclues[c].SetBit(d)
  create arrays deadvec[0..16] and statevec[0..16]
  deadvec[0] := statevec[0] := (0,...,0)
  InitHittingVectors(U,m)    // finally set up hitting vectors as in Section 6.1
```

The actual procedure that recursively adds clues to a hitting set then looks like this:

```
AddClues(j,U,m,hitset,statevec,deadvec)
  if statevec[j] = (1,...,1)
    then GeneratePuzzles(j,hitset,deadvec[j]) // all unav'ble sets are hit, add
                                              // 16-j clues in all possible ways
    return
  else if j=16
     then return  // after drawing 16 clues some unav'ble sets are still not hit
  i := statevec[j].GetIndexOfLowestZeroSlot() // pick first unav'ble set not hit
  for each c in U[i]
    do if deadvec[j].GetBit(c) = 0 // first verify that this clue is still alive
      then hitset[j+1] := c        // add clue to hitting set
           statevec[j+1] := statevec[j] OR hitvec[c]    // update state vector
                                                        // exclude smaller clues
           deadvec[j+1] := deadvec[j] OR U[i].deadclues[c]
           AddClues(j+1,U,m,hitset,statevec,deadvec)    // add more clues
```

We now convince the reader that the above algorithm does not miss any hitting sets. Suppose that G is a sudoku solution grid containing a proper 16-clue puzzle P. We need to show that algorithm just presented will find P. Let U be the member of the family $\mathcal{F}$ of unavoidable sets used for drawing the first clue from. Since P is a proper puzzle, it intersects U, so we may set $c = \min P \cap U$, i.e., we let c be the smallest clue of P contained in U. When we add c to our candidate hitting set, only clues smaller than c will be excluded from the search, however, as c is the smallest clue of P contained in U, no clues of P will actually be excluded. The exact same is true for the second, third, etc., clue we add — at each stage, when we add the smallest clue of P also contained in the unavoidable set in question, only clues not appearing in P will be marked 'dead'. After adding the 16^{th} clue in this way, our hitting set will equal P. Since P hits all unavoidable sets, P hits all unavoidable sets in the family $\mathcal{F}$ that *checker* uses, and therefore no further unavoidable sets are available, so that *checker* will test the set P for a unique completion (see Section 4), and thus find the 16-clue puzzle P.

As a closing remark, we originally added the dead clue vector to *checker* because we wanted to search this special sudoku grid

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

Figure 12: A grid containing 29 proper 17-clue puzzles

for all 17-clue puzzles. To this day, this grid holds the record as the grid having the largest known number of 17-clue puzzles (29, all found by Gordon Royle). In 2005, this grid was considered a likely candidate to have a 16-clue puzzle, but using *checker* we were the first to prove that there is no 16-clue puzzle in this grid [17]. Of course, we also wanted to know exactly how many 17-clue puzzles it contained, but the very first *checker* would have taken several months of CPU time to answer that question. After we had implemented the dead clue vector in 2006, we were able to exhaustively search this grid in less than a week for all 17-clue puzzles. The result was that Royle had already found all of them, i.e., it was now known that there are *exactly* 29 proper 17-clue puzzles contained in this grid.

7 Hitting set algorithm of the new checker

This section is a detailed version of the outline provided in Section 4.2. We will explain our new algorithm, the algorithm that we used to enumerate the hitting sets of size 16 given a family of unavoidable sets for a particular fixed sudoku solution grid. As far as this hitting set algorithm is concerned, there are really three improvements over the original *checker* described in the last section. In this section we describe these improvements.

The first improvement is that we added higher-degree unavoidable sets to *checker* so as to obtain an early "no" during the enumeration of hitting sets whenever possible, i.e., in order to abandon the search of a branch as soon as possible. For instance, if, after drawing 15 clues, there is an unavoidable set of degree 2 that is not yet hit, then we do not have to continue and draw the 16^{th} clue as we know that at least two more clues are required for a hitting set. We outlined this improvement in Section 4.2, and we give the details here in Section 7.1.

The second difference is that we discard all those unavoidable sets that have been hit after drawing the first few clues, so that, when adding the remaining clues, we are working with shorter vectors (i.e., a smaller amount of data). For instance, initially we begin with (up to) 384 minimal unavoidable sets, and after drawing the first seven clues, we check which unavoidable sets have been hit and continue with only the smallest (up to) 128 unavoidable sets. So when picking the last nine clues, for tracking the minimal unavoidable sets we are using binary vectors of length 128 only, not binary vectors of

length 384 as with the first seven clues. Similarly for the higher-degree unavoidable sets. We describe this technique in Section 7.2.

The third improvement is that, when choosing which unavoidable set to use for drawing the next clue from, we now invest some effort to make the best, or at least a better, choice. Recall that with the original *checker*, we selected the unavoidable set to use for drawing the next clue from in a greedy fashion — we simply used one of smallest size. However, this is not generally optimum. A different unavoidable set of the same size, or even a bigger set, may be a better choice since some of its clues may have been excluded from the search already, so that its *effective* (or *real*) size, and hence the number of branches to be taken, may actually be smaller. Therefore, when choosing unavoidable sets for drawing clues from, the new *checker* also takes the dead clue vector into account. We describe this in Section 7.3.

The first of the above changes — the idea to use higher-degree unavoidable sets — is certainly the most important one, and without it this computation would not have been feasible for several years. However, the other two changes, too, saved us a considerable amount of CPU cycles.

7.1 Improving backtracking using higher-degree unavoidable sets

Here we will explain the most important of the three improvements we made to our hitting set algorithm compared to the version of 2006. It is about how the use of higher-degree unavoidable sets enables us to considerably prune the search tree. We begin with the following definition.

Definition. A collection of pairwise disjoint unavoidable sets in a sudoku solution grid is called a *clique*. If there is no clique having a greater number of unavoidable sets, then we further say that the clique is *maximum*.[7]

Recall that, by Proposition 9, if after drawing j clues, there is an unavoidable set of degree $17 - j$ that is not hit, then we do not need to traverse the respective branch of the search tree as we already know that it cannot contain any proper 16-clue puzzles. On the other hand, from Corollary 8, the union of the sets in a clique of size d is an unavoidable set of degree d. Therefore, right before we begin the enumeration of hitting sets, we obtain a (usually quite good) collection of unavoidable sets of degree 2, 3, 4, 5, simply by finding cliques of size 2, 3, 4, 5.

We track these higher-degree unavoidable sets during the enumeration of hitting sets just like the ordinary (degree 1) unavoidable sets, i.e., through the use of state and hitting vectors for each degree. After twelve clues have been drawn, if there is an unavoidable set of degree 5 in our collection that is not hit, then we may abandon the search and backtrack immediately. Similarly if there is an unavoidable set of degree 4, 3, or 2 in our collection that is not hit after drawing 13, 14, or 15 clues, then we may abandon the search and backtrack immediately.

This may seem like an obvious way to prune the search tree with the hitting set problem. However, back in 2008 when we first realized that this idea would allow us to dramatically speed up *checker*,

[7] This terminology comes from graph theory — in the original *checker*, a maximum clique was found by setting up an undirected graph whose vertices were the minimal unavoidable sets, and where two vertices were adjacent if the corresponding unavoidable sets were disjoint. Hence the term "max clique number", or *MCN* for short — the biggest number of pairwise disjoint unavoidable sets that a grid possesses. In particular, a grid whose MCN is m cannot have a puzzle with fewer than m clues.

this was not yet described anywhere in the literature. This is surprising because, like the other improvements we made, it is not at all specific to sudoku but applies in an equal manner to the general hitting set problem. The first public mention of higher-degree unavoidable sets, to our knowledge, was in a posting of 23^{rd} July 2010 to the sudoku programmers' forum by Mladen Dobrichev, who had just released the first version of his open-source tool *GridChecker* [24]:

> "UA set is a region of the grid where we know at least one clue must exist. [...] Additionally there are regions where at least two clues must exist. A trivial example of such region is the union of 2 mutually disjoint UA sets — UA sets which have no cell in common. But, it is not necessary such regions to consist of disjoint UA. For example 3 UA of size 6 could form region of size 9 requiring at least 2 clues. [...] Similarly there are regions where at least 3, 4, 5, etc. clues must exist."

It is remarkable how Dobrichev even used the term *trivial* unavoidable set. However, although *GridChecker* does use higher-degree unavoidable sets, at the time of our work it used a relatively limited collection of such sets, namely those coming from the members of a maximum clique.[8]

There was another public mention of this idea and how it may be used to prune the search tree, in November 2010, when Hung-Hsuan Lin and I-Chen Wu published the paper [40] (for an updated version, see [1]). In this work, in Lemma 2 it is shown that the search for a k-clue puzzle in a sudoku grid may be stopped after selecting j clues if there are at least $k - j + 1$ unavoidable sets that are not yet hit ("active" unavoidable sets, in the language of that paper) and which are pairwise disjoint. The authors further describe how they used this observation to speed up our original *checker* by a factor of 129 and hence achieved a running time of 13.9 seconds per grid on average.

As far as the problem of finding a clique of a certain size of active unavoidable sets is concerned, they point out that the maximum clique problem is itself NP-complete (like the hitting set problem), and that they therefore use a greedy algorithm for attempting to construct cliques of the desired size. This is not the most efficient way to construct cliques, however, and it is likely to be the main reason why our own, new *checker* is about twice as fast as the *checker* written by Lin and Wu. For, constructing cliques over and over again, even just small ones, means duplicating effort. In contrast, with our new *checker* we compute a large number of cliques of all sizes less than or equal to 5 *exactly once* at the beginning of the search, and keep track of which ones are hit (become inactive) as we add clues.

The consequence is that, e.g., after drawing twelve clues, we merely have to do a boolean OR of two binary vectors and then check if the resulting vector has a 1 in every slot in order to find out if there is a clique of size 5 not yet hit. Note that, the moment we find that one slot has a 0, we do not need to compute the remaining slots. In other words, it is actually sufficient to do the boolean OR on just part of the vectors involved at first. In the case of the degree 5 unavoidable sets, we compute the required vector in three steps, checking for a slot containing a zero at the end of each step. All that can be done efficiently using SIMD programming, whereas constructing a clique of size 5 from scratch is certainly more work and in particular involves more dependencies (where an operation requires the output of the previous one) and is therefore not very suitable for SIMD programming. Of course, with our method, more work has do be done upfront (while the first eleven clues are picked), but on the

[8] *GridChecker* also uses a maximum clique, however, it does so in a more clever way than our original *checker*. Moreover, though a powerful collection of unavoidable sets of degree 2 is actually being used, it seems that it is only fully deployed in the method $\mathrm{chunkProcessor::iterateClue}$, for which it was "rare", in the words of Dobrichev, that the last clue was being picked there. (In *GridChecker*, the last clue is usually drawn, if at all necessary, in $\mathrm{chunkProcessor::iterateClueBM}$.)

other hand, a greedy algorithm will often miss cliques of the required size even though they exist. On balance, it seems that ours is the more efficient approach.

Summary

Now is a good time to summarize exactly what we do. We will walk through the case of the cliques of size 4; the other ones are similar. So suppose that there are m sets $\mathrm{U}[0], \ldots, \mathrm{U}[\mathrm{m}-1]$ in our initial family of unavoidable sets. We add the following statements to the procedure $\mathrm{InitHittingVectors}$, see the pseudocode in Section 6.1:

```
var   count := 0;
const START := 27;
create array CLQ4[0..32767]
for i from START to m-1
  do for j from START-1 to i-1
       do if U[i] intersects U[j]
            then continue
          for k from START-2 to j-1
            do if U[k] intersects either U[i] or U[j]
                 then continue
               for l from START-3 to k-1
                 do if U[l] intersects none of U[i],U[j],U[k]
                      then // found a clique of size 4
                        CLQ4[count] := union(U[i],U[j],U[k],U[l])
                        inc(count)
                        if count = 32768
                          then goto SetUpHittingVectors
SetUpHittingVectors:
 create arrays quadhitvec[0..80] and quadstatevec[0..80]
 for c from 0 to 80
   do quadhitvec[c] := (X_{CLQ4[0]}(c),...,X_{CLQ4[count-1]}(c))
 quadstatevec[0] := (0,...,0) // initially, no clique of size 4 is hit
```

Here $\mathrm{CLQ4}$ is an array of sets, $\mathrm{quadhitvec}$ is an array of binary vectors of length 32,768, and

$$\mathcal{X}_{\mathrm{CLQ4[i]}} \colon \{0, \ldots, 80\} \to \{0, 1\}$$

is again the characteristic function for the set $\mathrm{CLQ4[i]}$ with respect to $\{0, \ldots, 80\}$, see Section 6.1. The purpose of the constant START is to ensure that we do not collect cliques that will be hit anyway after drawing 13 clues — obviously there is no point using cliques involving $\mathrm{U}[0]$, for example, since the first unavoidable set in our family will always be used for drawing the first clue from, so it will certainly be hit by the time we check if there is an unhit clique of size 4.

When adding clues to the hitting set under construction, we need to add one statement (see the pseudocode for the procedure $\mathrm{AddClues}$ in Section 6.2), namely

```
quadstatevec[j+1] := quadstatevec[j] OR quadhitvec[c]
```

Recall that j is the index of the clue we add to the hitting set and c is the clue itself. Finally, we need to add logic so as to check if there are any cliques of size 4 not hit after we have drawn the 13^{th} clue:

```
if j = 13 and not quadstatevec[13] = (1,...,1)
  then return // a clique of size 4 is unhit after 13 clues have been added
```

The return-statement means that we backtrack.

7.2 Consolidating binary vectors for the innermost loops

The second improvement we made to the hitting set algorithm is that, with the unavoidable sets, after picking the first few clues, we discard those sets that have already been hit. The reason for this is that we then have to carry fewer data, i.e., we can use shorter vectors in the innermost loops of *checker*, where most of the compute time is spent. For instance, when enumerating hitting sets, initially we use a binary vector of length 32,768 to track the unavoidable sets of degree 4, but after the first five clues have been drawn, we switch to a vector of length 1,536. Of course we also have to update the respective hitting vectors — we refer to this process as *consolidating* the hitting vectors. However, this is not particularly advanced never mind original; what we are doing here is really just a standard *gather* operation. The key part was to realize that consolidating (gathering) the hitting vector actually helps here.

Continuing with the example of the degree 4 unavoidable sets from Section 7.1, in detail, what we do is as follows. After five clues have been added to the candidate hitting set, most of the unavoidable sets of degree 4 will usually have been hit, because the degree 4 unavoidable sets we use typically have around 30 elements. The probability that a given subset of size 30 of a sudoku solution grid does not contain five randomly chosen clues is

$$\frac{\binom{81-30}{5}}{\binom{81}{5}} \;=\; \frac{\binom{51}{5}}{\binom{81}{5}} \;\approx\; 0.092.$$

Therefore we expect that only about one in eleven unavoidable sets of degree 4 is *not* hit after selecting five clues. In other words, the state vector for the degree 4 unavoidable sets will carry 1's in approximately ten out of eleven places, on average. (Recall that a 1 in the i^{th} slot of this vector means that the i^{th} unavoidable set of degree 4 has been hit already.) For all future clues to be drawn, we only need to keep track of the unavoidable sets corresponding to the 0's in the state vector, since only those unavoidable sets have not yet been hit. One way to do this is to recompute the 81 hitting vectors for this much smaller collection of degree 4 unavoidable sets, just as we did with the original collection right before the enumeration of hitting sets began.

A more efficient method is to take the original hitting vectors and shrink them to vectors of length at most 1,536, where the i^{th} slot, $0 \leq i \leq 32767$, is erased according as the state vector carries a 1 or a 0 in this slot. Pseudocode for the obvious algorithm to accomplish this — which processes one slot at a time — looks as follows:

```
var j := 0 // j is the index of an unavoidable set in the new collection
create array newquadhitvec[0..80]
for c from 0 to 80  // set all the new (shorter) hitting vectors to 0
  do newquadhitvec[c] := (0,...,0)
for i from 0 to 32767 // i is the index of a set in the original collection
  do if not quadstatevec[5].IsBitSet(i)
       then
         // the set of index i is not yet hit after picking five clues, so
         // it becomes the set of index j of the new collection
         // ==> for each clue, copy the i-th bit of the original hitting
         // vector to the j-th bit of the new hitting vector:
         for c from 0 to 80
           do newquadhitvec[c].SetBit(j,quadhitvec[c].GetBit(i))
         inc(j)
         if j = 1536
           then return // we have found 1536 unhit unav'ble sets of degree 4
```

However, it is possible to do considerably better than that, by processing more than one slot (bit) at a time. For, suppose that we are given a consolidation (gathering) function

$$\mathrm{Con}\colon \{0,1\}^8 \times \{0,1\}^8 \to \{0,1\}^8$$

that shrinks the second input vector such that precisely those slots are eliminated in which the first of the input vector carries a 1. For example,

$$\mathrm{Con}((0,1,1,0,1,0,1,0),(1,1,0,1,0,0,1,0)) \;=\; (1,1,0,0,0,0,0,0),$$

because the first input vector has 0's in slots 1, 4, 6 and 8 and the second vector has the digits 1, 1, 0 and 0, respectively, in these four slots, which therefore become the first four slots of the output vector; naturally slots 5–8 of the output are filled with 0's. Suppose further that we are given a function

$$\mathrm{hamwt}\colon \{0,1\}^8 \to \{0,\ldots,8\}$$

that computes the Hamming weight of a binary vector of length 8. Then, given these two functions Con and hamwt, we can now easily process eight slots at a time:

```
var cnt := 0 // 'cnt' is the no. of unavoidable sets in the new collection, so far
var tmpvec   // a temporary binary vector of length 1536
for c from 0 to 80
  do newquadhitvec[c] := (0,...,0)
for i from 0 to 4095 // 4096 = 32768 / 8
  do for c from 0 to 80
    do tmpvec := Con(quadstatevec[5].GetByte(i),quadhitvec[c].GetByte(i))
       tmpvec.ShiftRight(cnt)
       newquadhitvec[c] := newquadhitvec[c] OR tmpvec
  cnt := cnt + (8 - hamwt(quadstatevec[5].GetByte(i)))
  if cnt >= 1536
    then return
```

Here, the method $\mathrm{GetByte}$ is similar to GetBit, except that it returns 8 bits at a time, while $\mathrm{ShiftRight}$ shifts the respective binary vector by the specified number of places, inserting 0's at the front.

Implementation notes

In *checker*, the function Con is implemented using the precomputed table $\mathrm{confunctab}$, while hamwt is already provided by the hardware through the POPCNT instruction. So the above is really an implementation of 8-bit *software* scatter. Note that with the Haswell microarchitecture, Intel processors support (64-bit) *hardware* scatter by way of the PEXT instruction. Therefore, rewriting the procedure $\mathrm{ConsolidateHitvec}$ in *checker* to take advantage of this instruction should result in a very significant performance increase on Haswell CPUs.

With the actual (Nehalem-optimized) code in *checker*, we further performed partial loop unrolling with the inner of the above two loops, by always processing the hitting vectors for six clues in parallel. Finally, one more slight improvement is gained by considering (shrinking) only those hitting vectors that correspond to clues that are not yet dead, and ignoring the other ones.

7.3 Taking the effective size of the minimal unavoidable sets into account

In this section we describe the final improvement over our original hitting set algorithm. Recall that, at each stage, our hitting set algorithm adds clues from the first unavoidable set that is not yet hit, as described in Section 6.1. Since the unavoidable sets are ordered by size, the set in question will always be one of smallest size. However, this is not usually the best choice. For instance, if the unavoidable set of lowest index that is not yet hit has empty intersection with the set of currently dead clues, and the unavoidable set of second lowest index that is not yet hit has the same size but one of its clues has been marked 'dead' earlier, then it is obviously better to use the unavoidable set of second lowest index for drawing clues from. For this reason, in the new *checker*, when selecting the first ten clues we always use an unavoidable set of minimum effective size.[9] Here, the *effective size* of an unavoidable set is the number of clues it has that are not yet dead.

Implementation notes

The way to efficiently accomplish this is to first invert the vector of dead clues, so that we obtain the *vector of alive clues*, i.e., the binary vector that has a 1 in slot i precisely if the clue i is still alive. Then, for each unavoidable set that is still unhit, we take the boolean AND of the vector of alive clues with the vector that has a 1 in exactly those slots corresponding to the clues this unavoidable set contains. In other words, in the latter vector we simply set all slots to zero that correspond to clues that are dead. We finally obtain the Hamming weight of the resulting vector, which is equal to the effective size of the unavoidable set in question. We do this for all unavoidable sets, and we always remember the index of the first set that had the smallest effective size, so far.

8 Running through all grids: the final computation

With the catalogue of all essentially different sudoku grids and *checker* in hand, the actual search for 16-clue puzzles involved running *checker* on each grid in the catalogue. We outlined these steps in Section 1.1. Because of the number of grids to check (about 5.5 billion) this had to be done using a large number of processors. In this section we discuss the details of this computation.

8.1 Some remarks on the implementation of the new checker

Following our successful PRACE prototype project application in 2009, we were able to test an early version of the new *checker* on four different hardware platforms (AMD Istanbul, IBM Blue Gene/P, IBM Power 6, Intel Nehalem), and we found that Nehalem is the best for us. Hence we optimized *checker* for Nehalem, to the extent of rewriting critical routines in assembly language. We did this in such a way so as to maximize simultaneous use of the different execution units of a Nehalem core (instruction-level parallelism). We further heavily used SSE to facilitate data-level parallelism. Also, using machine language allowed us to retain key data *checker* frequently uses inside the processor's registers, thereby further reducing overhead. During the development of these assembly routines, we

[9] For the eleventh clue we still find the unavoidable set of minimum effective size among the first 64 unavoidable sets in our collection, and for the twelfth clue we find the unavoidable set of minimum effective size among the first five unavoidable sets not yet hit. For drawing the remaining four clues we always simply use the first unavoidable set not yet hit. The reasons for this are explained in Section 8.1.

found the optimization manuals by Agner Fog [39] to be most helpful. Moreover, we took advantage of the SMT mode of a Nehalem/Westmere CPU (hyper-threading), which was enabled on the cluster *Stokes* at ICHEC that we used for this computation. We give more details on the parallelisation later in this section.

Apart from the above, the main change to the implementation of the new *checker* (compared to the version from 2006) is that we expanded all function calls when enumerating hitting sets. So there are no recursive function calls when drawing more clues in the new *checker*, rather, there are now sixteen nested loops, one for each clue.

One other implementation detail is that instead of performing a boolean OR (as described in the earlier sections) for updating the binary vectors that track the unavoidable sets of the various degrees, and later test if a vector is all 1's, we actually perform a boolean AND and check if the vector is all 0's. The reason for this is that with SSE 4.2, it is slightly easier to test if an XMM-register (128 bits) is all 0's than if it is all 1's.

Tradeoffs

During the development of *checker*, there were several design choices that involved tradeoffs, i.e., there were often different alternatives that each had their advantages, so we had to find out the overall best (fastest) by experimentation. These include:

- The number unavoidable sets to use, both minimal and of higher degree. Obviously using more unavoidable sets results in fewer hitting sets being found, however, there is of course a price for having access to a larger selection of unavoidable sets. We found that initially starting out with up to 384 minimal unavoidable sets was the optimum. We also tried adding unavoidable sets of size 13 to *checker*, but it seemed to make almost no difference to the average running time one way or another.

- Which higher-degree unavoidable sets to use. There does not seem to be a big difference whether or not one uses unavoidable sets of degree up to 5 only, or also unavoidable sets of degree 6. However, adding unavoidable sets of degree 7 appears to slow down the computation.

- The point at which we consolidate the hitting vectors. Doing that later means a more powerful collection of unavoidable sets in the innermost loops of *checker*, where most of the running time is spent. However, doing that later also means having to do it more often. The best combination could only be found through experimentation.

- Finding the best unavoidable set to use for drawing clues from. In principal it would be best to always use the unavoidable set having the least effective size, but obviously not so if the performance hit incurred by finding this particular unavoidable set outweighs the gain. Again, some experimentation was required.

- The number of chunks into which we divide a binary vector (of large length) when we test to see if every slot is 1. Having more chunks means that we can potentially save work — in case one of the slots of the vector is 0 — but on the other hand it also means more if-statements (to check whether or not a chunk is all 1's), i.e., more (mispredicted) branches.[10]

[10] In his document *The microarchitecture of Intel, AMD and VIA CPUs*, Agner Fog states that according to his measure-

8.2 Parallelisation strategy for the grid search

The parallelisation strategy we used with this project was driven by the simple fact that each sudoku solution grid could be checked independently. Therefore, as long as we ensured that all grids were actually processed, they could be handled in any order. We focused our parallelisation effort on a HPC infrastructure. In this domain, the main parallelisation paradigm is MPI. This *Message Passing Interface* library gave us all the flexibility we needed to achieve an efficient and reliable parallelisation.

Loadbalancing and task-farming

The final design of the parallel code was based on a master/slave architecture. One group of master processes manages the reading of the grids to search, distributes the work to a pool of slave processes, collects the results, and finally writes them to disk. The exact ratio between masters and slaves, along with the work distribution policy and the resulting printing pattern, was unclear at first. That is why we made an initial version of the code very configurable in this regard, and we experimented with various options to find the most efficient one. This took place during our access to the PRACE infrastructure for a prototype machine evaluation project. Following our PRACE tests, the configuration we finally selected consists of one single master process per run. This proved sufficient for the jobs we typically ran, which had less than 500 slave processes. When using more than 500 slaves, running several independent jobs with one master process each proved more efficient at the batch scheduler level than increasing the size of the slaves' pool for one single job.

For distributing the work to the slave processes, the loadbalancing (which refers to the distribution, or *task-farming*, of the tasks by a master node to the slave nodes) algorithm we implemented was as follows: a global list of sudoku grids to search was first read from disk by the master. Next a (small) batch of grids was sent to each slave in a round-robin fashion. Then the master would wait for a slave to contact it upon finishing. Each slave process would run *checker* on the batch of grids it had been assigned and then send the results back to the master. The master would answer whichever slave tried to contact it first, collect its results, store the results on disk, check if more work was available, and either send new grids to search to the slave or command it to stop, according as sufficient (job) time was still available. The master then would wait for the next slave to contact it, until all slaves had been commanded to stop. Finally the master itself would stop and hence the job would terminate. For optimum efficiency, we naturally wanted all slave nodes to finish around the same time.

This is the ideal case, and the job finishes properly. However, it might happen that a job finishes unexpectedly. For example, this would be the case if the batch scheduler killed the job for some reason, like having reached the wall-time limit. In this case, we could restart the very same job, and all the previously processed sudoku grids would be automatically removed from the pool of work to do, avoiding any waste of resources. Only grids that had either been partially processed by a slave, or finished but for which no acknowledgment had been transmitted to the master process, would be re-processed. Because searching a grid usually takes only a few seconds, and a job duration is typically 24 to 84 hours, the potential number of grids to re-compute compared to the total number of grids checked in one compute job is tiny. So the usage of the CPU cores was near-optimal, i.e., almost no processor time was lost at the end of a job. Moreover, as the missing grids from a previous job would be the first ones to be computed at restart, if for some reason one grid takes an unusually long time to

ments, the penalty for a mispredicted branch on Nehalem is at least 17 clock cycles [39].

be searched by *checker*, which increases the likelihood that the job will get killed for time-out during its computation, that grid would be re-processed right at the start of the next job and so there would be more than enough time to complete it.

The parallelisation technique we used was therefore well suited for our kind of workload, where the various individual jobs would take an unpredictable time to finish. Having a first-come first-served work distribution policy ensured a natural load re-balancing between the slave processes, where the ones dealing with harder grids would simply report less often to the master process than the one with easier grids.

Extra performance refinements

Selecting Intel Nehalem as our target platform gave us additional opportunity for performance tuning. Indeed, this processor architecture supports a feature called SMT mode (hyper-threading). This *Simultaneous Multi-Threading* mode allows two processes to run concurrently on each CPU core. Exploiting the SMT mode proved to be effective for us, leading to a 26.5 per cent performance gain at no cost. To cope with the scheduling policy constraints on the *Stokes* cluster, we had to further parallelise our code using OpenMP. That way, we were able to pin each MPI process to one physical CPU core, where two OpenMP threads were running concurrently using the 2-way SMT mode available there.

8.3 Ensuring the correctness of our programme checker

Certainly the most important aspect of this project is correctness. At first glance, it may appear difficult to verify the correctness of our programme *checker* on the grounds of the lack of available test cases (no sudoku grids containing any 16-clue puzzles). To be able to do some testing anyway, we produced a version of *checker* that searches for 17-clue puzzles. This version had minimal changes over the version that searches for 16-clue puzzles — we made only those changes that were absolutely necessary. We ran this *checker* on all known grids having at least one 17-clue puzzle, and with every such grid, all the 17-clue puzzles that grid was known to contain were found by *checker*.

As far as white-box testing goes, using the debugger we stepped through every line of the code, carefully verifying that each instruction does exactly what we thought it would do. We also added pre- and post-conditions throughout the code (in the form of `assert`-statements), to check the internal consistency of the data structures and to make sure that parameters passed were within their respective valid range. For further debugging, we used the tool *valgrind*, as well as its companion *cachegrind*, the latter mainly for optimizing memory accesses. Moreover, all changes to the code during development were tracked using a version control system.

We implemented two safety checks in *checker*. Firstly, after the procedure that finds all the minimal unavoidable sets in a grid is finished, we again test each set found, by running its complement through the solver, to make sure that the set is really unavoidable. Secondly, we also double-check the answer the solver produces. Each time a puzzle is run through the solver, we have the solver save the first two completions found. Then we check that both completions are in fact valid completions of the given puzzle, and we further verify that they are actually different. Should that ever not have been the case, the grid in question would have been logged. However, this never actually happened, i.e., the solver always produced correct answers.

We technically used six different versions of *checker* throughout the computations. That is to say,

we updated our code five times during the computations, where each version was a small improvement and a little faster than the previous version. It would have been optimal to use a single version of the code for the entire computation, however, we were under time pressure due to the competitive nature of this project, and we needed to start running compute jobs as soon as was possible. Had we used just the first version of the new *checker* for the entire computation (as it was at the time we got access to the cluster *Stokes*), then we would have needed at least 1.5 million additional compute hours on top of what we actually used. Only through updating *checker* were we able to finish the computation before the end of 2011. Note however, that the difference between one version of *checker* and the next one was usually just one procedure that was rewritten, i.e., the changes between updates were relatively minor, and of course we fully tested each version before the upgrades, as described in the beginning of this subsection.

8.4 The actual computation

The entire computation took about 7.1 million core hours on the *Stokes* machine at ICHEC. *Stokes* is an SGI Altix ICE 8200EX cluster with 320 compute nodes. Each node has two Intel (Westmere) Xeon X5650 hex-core processors and 24 GB of RAM. We divided the computation up into several hundred jobs. We started running jobs in January 2011, and we finished in December 2011. For each grid, we recorded the number of hitting sets found, and the time taken to search that grid. This data is stored in our log files. The average running time of our final version of the new *checker* is about 3.6 seconds per grid (on a single core of the above model of CPU, with hyper-threading enabled).

8.5 GPU computing opportunities

The opportunity of using Graphic Processing Units (GPUs) for this project has been examined and rejected for the reason that at the time we could have begun porting the code to a GPU, we were already more than half-way through the computations. Changing the code would have required development and validation time which, even with a dramatic speed-up, would not have permitted us to finish the project earlier. Moreover, it is not clear to us if our hitting set algorithm is a suitable application for a GPU, at all.

For starters, one would need to come up with a very different parallelisation strategy — exploiting the intrinsic parallel nature of the global workload is definitely not sufficient to perform an effective parallelisation at the GPU level. The reason is that, due to the specific architecture of GPUs, a large number of threads would all need to work on a single grid, rather than on different grids. This would necessitate parallelism at the code level itself, not just at the workload level.

At first, one might think this to be possible because checker mainly performs vector operations such as boolean-OR and reductions, which are well-suited for a GPU, provided the respective vectors are long enough. Unfortunately, if the GPU is used only as a coprocessor for vector operations, then performance benefits are likely to be limited, because this approach would involve a lot of communications between the host and the GPU, i.e., the PCIe channel would almost certainly be a major bottleneck. With that said, if larger chunks of the computations could be kept on the GPU, even if this means less (relative) efficiency there than on a CPU, then it might actually be possible to reap a performance gain.

Acknowledgements

This work has built on ideas and work of many other people. It began from reading posts on the sudoku forums, and we thank many of the posters. They include Guenter Stertenbrink, Gordon Royle, Ed Russell, Glenn Fowler, Roger Wanamo, who helped at various stages. We also thank Konstantinos Drakakis and ICHEC for some extra CPU hours. We thank the staff of ICHEC who were very supportive throughout the project. Finally, we are grateful to Sascha Kurz for his comments on an earlier version of the manuscript.

References

[1] Hung-Hsuan Lin, I-Chen Wu, *An Efficient Approach To Solving The Minimum Sudoku Problem*, ICGA Journal, Vol. 34, No. 4 , pp. 191–208, 2011.

[2] Kenneth Appel, Wolfgang Haken, *Every Planar Map is Four Colorable Part I. Discharging*, Illinois Journal of Mathematics 21:429–490, 1977.

[3] Neil Robertson, Daniel P. Sanders, Paul Seymour, Robin Thomas, *The Four-Colour Theorem*, J. Combin. Theory Ser. B 70(1): 2–44, DOI:10.1006/jctb.1997.1750, 1997.

[4] Thomas C. Hales, *A proof of the Kepler conjecture*, Annals of Mathematics. Second Series 162 (3): 1065–1185, doi:10.4007/annals.2005.162.1065, 2005.

[5] Harald A. Helfgott, *Major arcs for Goldbach's theorem*, http://arxiv.org/abs/1305.2897, 2013.

[6] Danielle Passos de Ruchkys, Siang Wun Song, *A parallel approximation hitting set algorithm for gene expression analysis*, in *14th Symposium on Computer Architecture and High Performance Computing*, Vitoria, Espirito Santo, Brazil, pp. 75–81, 2002.

[7] Faisal Abu-Khzam, *Kernelization Algorithms for d-Hitting Set Problems*, in *Proceedings of the 10th Workshop on Algorithms and Data Structures (WADS 2007)*, LNCS, Vol. 4619, pp. 434–445.

[8] Alexei Vazquez, *Optimal drug combinations and minimal hitting sets*, BMC Systems Biology 2009, 3:81. doi:10.1186/1752-0509-3-81

[9] Drew Mellor, Elena Prieto, Luke Mathieson, Pablo Moscato, *A Kernelisation Approach for Multiple d-Hitting Set and Its Application in Optimal Multi-Drug Therapeutic Combinations*, PLoS ONE 5(10): e13055. doi:10.1371/journal.pone.0013055

[10] Leonid Khachiyan, Endre Boros, Khaled Elbassioni, Vladimir Gurvich, *A New Algorithm for the Hypergraph Transversal Problem*, in *Computing and Combinatorics, 11th Annual International Conference, COCOON 2005*, LNCS, Vol. 3595, pp. 767–77.

[11] Rolf Niedermeier, Peter Rossmanith, *An efficient fixed-parameter algorithm for 3-Hitting Set*, Journal of Discrete Algorithms 1, pp. 89-102, 2003.

[12] Fabian Kuhn, Pascal von Rickenbach, Roger Wattenhofer, Emo Welzl, Aaron Zollinger, *Interference in Cellular Networks: The Minimum Membership Set Cover Problem*, in *Computing and Combinatorics, 11th Annual International Conference, COCOON 2005*, LNCS, Vol. 3595, pp. 188–198.

[13] Gary McGuire, *Gary McGuire's Sudoku Page*, http://www.math.ie/checker.html

[14] Gary McGuire, *Gary McGuire's Sudoku Page* (the original webpage from late 2006), http://www.math.ie/checkerold.html

[15] Gordon Royle, *A collection of 49,151 distinct Sudoku configurations with 17 entries*, http://mapleta.maths.uwa.edu.au/~gordon/sudokumin.php

[16] http://forum.enjoysudoku.com/ and http://www.setbb.com/sudoku/

[17] Jean-Paul Delahaye, *The Science behind Sudoku*, Scientific American, Vol. 294, No. 6, pp. 80–87, 2006.

[18] Brian Hayes, *Unwed Numbers*, American Scientist, Vol. 94, No. 1, pp. 12–15, 2006.

[19] Agnes Herzberg, M. Ram Murty, *Sudoku Squares and Chromatic Polynomials*, Notices of the AMS, Vol. 54, No. 6, pp. 708–717, 2007.

[20] Ed Pegg Jr., *Sudoku Variations*, Math Games, 2005, http://www.maa.org/editorial/mathgames/mathgames_09_05_05.html

[21] Laura Taalman, *Taking Sudoku Seriously*, Math Horizons, Vol. 15, Sept. 2007, pp. 5–9.

[22] http://dist2.ist.tugraz.at/sudoku/

[23] Max Neunhöffer, http://www-groups.mcs.st-and.ac.uk/~neunhoef/Publications/talks.html

[24] Mladen Dobrichev, *Sudoku GridChecker*, http://sites.google.com/site/dobrichev/

[25] Jason Rosenhouse, Laura Taalman, *Taking Sudoku Seriously: The Math Behind the World's Most Popular Pencil Puzzle*, Oxford University Press, USA, 2012.

[26] Ed Russell, forum post (MLE for the total number of 17-clue puzzles), January 2006, http://forum.enjoysudoku.com/pseudo-puzzles-t2747.html#p17640

[27] Richard Bean, *The size of the smallest uniquely completable set in order 8 Latin squares*, http://arxiv.org/abs/math/0403005

[28] *Mathematics of Sudoku*, Section 3.1, Wikipedia, http://en.wikipedia.org/wiki/Mathematics_of_Sudoku#Sudoku_with_rectangular_regions

[29] Ed Russell, forum post (proof that no 7-clue puzzle exists for 6×6 sudoku), March 2006, http://forum.enjoysudoku.com/sudoclues-max-min-forest-leaves-t3351-15.html#p23005

[30] Christoph Lass, *Minimal number of clues for Sudokus*, Central European Journal of Computer Science, Volume 2, Issue 2, pp. 143–151, June 2012.

[31] Christoph Lass, *Sudokus und das 16er Problem*, (updated) slides of a talk of Oct. 2010.

[32] Ebadollah Mahmoodian, G. H. John van Rees, *Critical sets in back-circulant Latin Rectangles*, Australasian Journal of Combinatorics 16 (1997), 45–50.

[33] Bertram Felgenhauer, Frazer Jarvis, *Mathematics of Sudoku I*, Mathematical Spectrum, Vol. 39, No. 1, pp. 15–22, 2006.

[34] Ed Russell, forum post (determination of the possible automorphism groups), February 2009, http://forum.enjoysudoku.com/about-red-ed-s-sudoku-symmetry-group-t6526-195.html#p66833

[35] Ed Russell, Frazer Jarvis, *Mathematics of Sudoku II*, Mathematical Spectrum, Vol. 39, No. 2, pp. 54–58, 2007.

[36] Glenn Fowler, forum post (list of all essentially different sudoku solution grids having at least one nontrivial automorphism), June 2010, http://forum.enjoysudoku.com/about-red-ed-s-sudoku-symmetry-group-t6526-195.html#p201053

[37] Glenn Fowler, *A 9x9 sudoku solver and generator*, http://research.att.com/~gsf/sudoku/

[38] Richard Karp, *Reducibility Among Combinatorial Problems*, in R. E. Miller, J. W. Thatcher (editors), *Complexity of Computer Computations*, New York: Plenum, pp. 85–103, 1972.

[39] Agner Fog, *Software optimization resources*, http://www.agner.org/optimize/

[40] Hung-Hsuan Lin, I-Chen Wu, *Solving the Minimum Sudoku Problem*, TAAI, pp. 456–461, 2010 International Conference on Technologies and Applications of Artificial Intelligence.

[41] Guenter Stertenbrink, *suexk* (sudoku solver), http://magictour.free.fr/sudoku.htm

[42] Brian Turner, *BBSudoku* (bit based sudoku solver), v1.0, October 2009, https://sites.google.com/site/bbsudokufiles/