There is no 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem

Principal Investigator: Gary McGuire*
Project Collaborator: Bastian Tugemann
Project Contributor: Gilles Civario

January 1, 2012

Abstract

We apply our new hitting set enumeration algorithm to solve the sudoku minimum number of clues problem, which is the following question: What is the smallest number of clues (givens) that a sudoku puzzle may have? It was conjectured that the answer is 17. We have performed an exhaustive search for a 16-clue sudoku puzzle, and we did not find one, thereby proving that the answer is indeed 17. This article describes our method and the actual search.

The hitting set problem is computationally hard; it is one of Karp's twenty-one classic NP-complete problems. We have designed a new algorithm that allows us to efficiently enumerate hitting sets of a suitable size. Hitting set problems have applications in many areas of science, such as bioinformatics and software testing.

*School of Mathematical Sciences, University College Dublin, Ireland. E-mail: gary.mcguire@ucd.ie

Contents

1 Introduction **3**

2 History **6**

2.1 Our Preliminary Work 6

2.2 Previous Work by Others 6

2.3 Heuristic Arguments That 16-Clue Puzzles Do Not Exist 9

3 Summary Description of Method **11**

4 The Catalogue of All Grids **12**

4.1 Equivalence Transformations 12

4.2 Applying Burnside's Lemma 13

4.3 Enumerating Representatives 13

5 Checker **14**

5.1 Overall Strategy of Checker 14

5.2 Unavoidable Sets For an Individual Grid 14

5.2.1 Finding minimal unavoidable sets 16

5.2.2 Higher-degree unavoidable sets 20

5.3 Enumerating the Hitting Sets For an Individual Grid 23

5.3.1 Algorithm of the original checker 23

5.3.2 Algorithm of the New checker 28

5.4 Using A Sudoku Solver 33

5.5 Tradeoffs 33

5.6 Some remarks on the implementation of the new checker 34

5.7 Testing checker 34

6 Running Through All Grids: Further Details about Code Performance and the Computation **36**

6.1 Platform 36

6.2 Load Balancing and Task Farming 36

6.3 The Actual Computation 37

7 Acknowledgements **38**

1 Introduction

In sudoku, the puzzle solver is presented with a 9×9 grid, some of whose cells already contain a digit between 1 and 9. The puzzle solver must complete the grid by filling in the remaining cells such that each row, each column, and each 3×3 box contains all digits between 1 and 9 exactly once. It is always understood that any proper (valid) sudoku puzzle must have only one completion. In other words, there is only one solution, only one correct answer.

In this article we consider the issue of *how many* digits are given to the solver in the beginning. There are 81 cells in the grid. Typically, in newspapers and magazines around 25 clues (digits) are given. If too few clues are given then there are multiple completions, i.e., the puzzle becomes invalid. It is natural to ask how many clues are always needed. This is the minimum number of clues problem:

What is the smallest number of clues that can possibly be given
such that a sudoku puzzle still has only one solution?

More informally — what is the smallest number of clues that you could possibly have? There are puzzles known with 17 clues, here is an example:

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

However, nobody has found any 16-clue puzzles, and it was conjectured that the answer to the sudoku minimum number of clues problem is 17. We have proved this conjecture. In this article we present our method.

The strategy we used to finally solve this problem is an obvious one — exhaustively search through all possible solution grids, one by one, for a 16-clue puzzle. So we took the point of view of considering each particular completed sudoku grid one at a time, and then looking for puzzles whose solution is that particular grid. We think of these puzzles as being “contained in” that particular grid. Our search turned up no proper 16-clue puzzles, but had one existed, then we would have found it.

A brute force exhaustive search would not have been feasible, but we have developed a novel algorithm that made the exhaustive search possible. Our programme, named *checker*, improves greatly over the original open-source version of checker from 2006, which would have been far too slow to search all sudoku solution grids. Indeed, the paper [18] estimates that our original version would take over 300,000 years on one computer to finish this project.

It is worth noting that there have been attempts to solve the minimum number of clues problem using mathematics only, i.e., not using a computer. However, nobody has made any serious progress. In fact, while it is very easy to see that a sudoku puzzle with seven clues will always have multiple completions, because the two missing digits can be interchanged in any solution, finding a theoretical reason why eight clues are not enough for a unique solution already seems hard. This is far from the conjectured answer of 17, so a purely mathematical solution of the minimum number of clues problem is a long way off.

Other Applications

Solving the minimum number of clues problem serves as a way to introduce our algorithm for efficiently finding all hitting sets of size k, where k is a given positive integer. This new algorithm has many other potential applications than just sudoku. To begin with, it is applicable to any instance of the hitting set problem, and such situations frequently occur in Bioinformatics (e.g., gene expression analysis), Computer Networks and Software Testing [11]. We note that the vertex cover problem from graph theory can be reduced to the hitting set problem, and so can the set cover problem, so both of these are actually equivalent to the hitting set problem. The set cover problem has applications to interference in cellular networks. Another situation where our algorithm could be applied is the quasi-group (Latin square) completion problem from combinatorics. Indeed, the minimum number of clues problem is an instance of a whole family of problems in combinatorics, where one studies *critical sets*. Critical sets are subsets of a structure that uniquely determine the whole structure. In other words, the entire structure can be reconstructed from the critical set. It is natural to look for the smallest possible critical set, which is what we are doing here for sudoku.

Other Comments

The computation was carried out on the Stokes cluster which is owned and run by the Irish Centre for High-End Computing (ICHEC). For further details see Section 6.

This article is *not* about any of the following topics: how to solve sudoku puzzles, how to create sudoku puzzles, how to rate the difficulty of sudoku puzzles, how to write a sudoku solver program.

Finally, we emphasize that we are not saying that all completed sudoku grids contain a 17-clue puzzle (in fact, only a few do). We *are* saying that no completed sudoku grid contains a 16-clue puzzle.

2 History

2.1 Our Preliminary Work

The work on this project began over six years ago. Back in August 2005, we started writing *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 last release of this original, open-source version of checker that we posted on the Internet is of November 2006 [2]. In 2009 we started working on a completely new implementation of checker optimized for the case $n = 16$, which up to now we had never published anywhere. This new version of checker takes only a few seconds to search an average sudoku solution grid for all 16-clue puzzles, whereas the last release from 2006 takes about an hour per grid on average.

In July 2009 we were awarded a Class C project from ICHEC to do some early testing on their clusters. In September 2009 we also successfully applied for a PRACE prototype award [3]. We were granted almost four million core hours on JUGENE, as well as time on a Cray XT5 cluster at CSC in Finland, an IBM Power 6 cluster at SARA in the Netherlands, and a Bull cluster at CEA in France. The main goal of this project was to evaluate different strategies for the load-balancing (see Section 6) of the hitting set problem. The actual code we ran was an early version of the new checker.

2.2 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 have surely wondered whether 16 clues are possible. Nikoli have a rule that none of their puzzles will have more than 32 clues.

- Over the past several years, people have collected almost 50,000 solution grids containing one or more 17-clue puzzles. These can be found on Gordon Royle's website [1]. Most of them were found by Royle, who compiled this list of 17-clue puzzles while searching for a 16-clue puzzle. There is constant discussion about the minimum number of clues problem (including a number of false proofs) in the relevant online discussion boards [4]. This is where we became aware of a completed grid found by Royle which contains twenty-nine different 17-clue puzzles (see Section 5.3.1). This was considered a likely grid to contain a 16-clue puzzle, and initially we started work on checker to solve this particular problem. Referring to the minimum number of clues problem, Royle states (January 3rd, 2011) on his

blog: “Doing the numbers suggests that something clever will be needed to solve this; even projected computer advances won’t be enough to resolve it in my lifetime ... ”

- The sudoku minimum number of clues problem has been mentioned in several journal publications [6, 7, 8, 9]. The last reference is an article entitled *The Science behind Sudoku* and written by the French Computer Science Professor J.-P. Delahaye, which appeared in the June 2006 issue of the *Scientific American*. This article quotes one of the authors (Gary McGuire) in conjunction with the sudoku minimum number of clues problem.

- In 2008 an eighteen-year-old girl submitted a proof of the nonexistence of a 16-clue sudoku puzzle as her entry to the German national science competition for high-school students (“Jugend forscht”), and she also published two papers (in German) in the journal “Junge Wissenschaft”. However, Sascha Kurz, a mathematician at the University of Bayreuth later had a project proposal for a Master’s thesis, that indicates he had found mistakes in the proof. That Master’s project was to summarize the state of the art of the minimum number of clues problem, and then to solve part of the problem. The proposal specifically mentions the above project, and cites the two respective papers, saying: “Unfortunately, the main arguments in the proof are not correct, so that the problem is still open.”

- A paper in the Notices of the American Mathematical Society [8] by Herzberg and Murty states on the first page: “For anyone trying to solve a Sudoku puzzle, several questions arise naturally. For a given puzzle, does a solution exist? If the solution exists, is it unique? [...] What is the minimum number of entries that can be specified in a single puzzle in order to ensure a unique solution?”

- A quick Internet search reveals that the sudoku minimum number of clues problem has been subject of, or at least mentioned in, quite a few talks in seminars/colloquia in mathematics and computer science department around the world. For example, a researcher in mathematics from the University of St. Andrews, Max Neunhöffer, gave a talk *Is there a Sudoku puzzle with 16 clues?* at the University of Aberdeen, outlining the very strategy we used for solving the minimum number of clues problem [10].

- Mladen Dobrichev, who appears to be a very competent programmer, has written a tool named *GridChecker* [17]. This programme basically does the same thing as our open-source version of checker, although it is considerably faster. In fact the

readme file for GridChecker mentions our original checker and even provides the URL of its homepage, saying: “The idea [of GridChecker] is based on the similar tool named checker (http://www.math.ie/checker.html).”

- In 2009, a team at the University of Graz in Austria verified, also using a computer search, that no proper sudoku puzzle can exist with fewer than 12 clues, and apparently they had also most of the computations finished that showed that in fact at least 13 clues are necessary. Their stated aim was build up to proving that no 16-clue sudoku exists, although that project appears not to be active anymore.

- A group of computer scientists at the National Chiao Tung University in Taiwan led by Professor I-Chen Wu in November 2010 published the paper *Solving the Minimum Sudoku Problem* [18]. In this article, which refers to one of the authors (Gary McGuire) and checker, they describe some of the techniques they used to speed up our original version of checker by a factor of 129. Professor Wu also gave a talk about this at the 2010 International Conference on Technologies and Applications of Artificial Intelligence. Around this time the Taiwanese research group started a distributed search over the Internet using BOINC, to search all inequivalent sudoku grids for a 16-clue puzzle. According to the project’s website, as of December 31st, 2011 they have checked 1,453,000,000 grids.

- Two research students at the University of Glamorgan in Wales, Sian Jones and Jemma Williams, are studying aspects of sudoku, see [5].

And these are the researchers we are aware of — it is of course very possible that there are further teams also working on the sudoku minimum number of clues problem, or related problems.

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. Most authors seem 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. 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 useful to us. Some researchers have tried to generalize their ideas to the case of arbitrary d, but the only such recent paper we could find presents an algorithm that has running time $\mathrm{O}(\alpha^k + n)$, where $\alpha = d - 1 + \mathrm{O}(\frac{1}{d})$ and n is the cardinality of the global set [12]. However, this algorithm is very similar to the one we used in our original version of checker from 2006. In a forthcoming paper we will therefore present the formal complexity analysis of our new

algorithm. We estimate its average-case complexity to be $\mathrm{O}(d^{k-2})$ with any instance of the hitting set problem for which the sets to be hit are of comparable density as with the sudoku minimum clues problem.

2.3 Heuristic Arguments That 16-Clue Puzzles Do Not Exist

There are two heuristic arguments at to why 16-clue puzzles should not exist, which we present here. These arguments were posted on the sudoku forum.

The first of these arguments shows that a 16-clue puzzle is not likely to exist. However, it also proves that a 17-clue puzzle is not likely to exist, and we know these *do* exist! Consider the total number of all possible grids, which is about $(6.7)(10^{21})$. A blank grid has this number of solutions. Suppose that each time we insert a clue in a blank grid we divide the number of possible solutions by 9. Since $(6.7)(10^{21})/9^{22} = 6.8$ and $(6.7)(10^{21})/9^{23} = 0.75$, we conclude that having 23 or more clues should give a puzzle with a unique solution, and 22 or fewer clues should give a puzzle with more than one solution. This argument assumes that all clue placements have an equal effect, which is clearly false; however, it perhaps gives an indication that puzzles with fewer than 22 clues are going to be rare. Puzzles with 16 (and 17) clues can therefore be assumed to be extremely rare.

The second argument is statistical (due to Ed Russell). People who send Royle a list of 17-clue puzzles usually do not have many new puzzles. One correspondent sent 700 puzzles, of which 33 were new. Assuming that Royle and this correspondent drew their 17-clue puzzles at random from the universe of all 17-clue puzzles, the maximum likelihood estimator for the size of the universe is about 35,000. This is an underestimate because of the (human) way we search for these puzzles, nevertheless we can assume that the 50,000 puzzles we have found must be nearly all the 17-clue puzzles that exist. On this list there are twenty-nine 17-clue puzzles with the same solution, but no more than twenty-nine have been found with the same solution. If a 16-clue sudoku exists, adding one clue to it in all possible ways would give sixty-five 17-clue puzzles with the same solution. Therefore, if we assume (by the argument given above or otherwise) that the list of known 17-clue puzzles is nearly complete, it is highly unlikely that a 16-clue puzzle exists.

On the other hand, one can argue that Royle's list of 17-clue puzzles may not be complete. To explain this, consider how the list was generated. To construct 17-clue puzzles, Royle started with a small number of 17-clue sudoku puzzles. He started to perturb them in various ways, by swapping a number for another number, or shifting an entry to another cell, all the time keeping track of any new ones that were found. This is

known in computer science as the A* search algorithm. He was hoping that by finding enough 17-clue puzzles, he would eventually stumble across a 16-clue puzzle contained in one of the 17-clue ones.

Therefore, the known 17-clue puzzles were mostly constructed from each other, so in some sense they are "close" to each other in the space of all puzzles. There could be another bunch of 17-clue puzzles that have not been found yet, lurking in some corner of puzzle space, and this bunch might contain sixty-five 17-clue puzzles with the same solution (and a 16-clue puzzle). This is highly unlikely, but theoretically possible.

3 Summary Description of Method

Our goal was to show that there are no sudoku puzzles with 16 clues, or of course to find one, had one existed. In summary, our method was as follows.

1. Make a catalogue of all 5,472,730,538 completed sudoku grids.
2. Write a program (named *checker*) that efficiently searches within a given completed sudoku grid for sudoku puzzles with 16 clues whose solution is the given grid.
3. Run through the catalogue of all completed grids and apply checker to each grid in turn.

We shall explain each of these steps in the next sections.

4 The Catalogue of All Grids

In total, there are exactly $6,670,903,752,021,072,936,960 \approx 6.7 \cdot 10^{21}$ sudoku solution grids [14]. However, it is not necessary to analyze all of them. For instance, permuting the digits of a given solution grid will obviously not change the substance of the grid as the individual digits carry no significance (any nine different symbols could be used). There are several other such *equivalence transformations* that can be performed, e.g., flipping the grid or taking its transpose, or interchanging the first and second row, etc. None of these alter the property of containing a 16-clue puzzle. This allows us to introduce an equivalence relation in the mathematical sense on the set of all sudoku solution grids, where two grids are equivalent if one may be obtained from the other by applying one or more of the equivalence transformations.

4.1 Equivalence Transformations

Definition. A *band* of rows is the set of rows 1-3, rows 4-6, or rows 7-9. A *stack* of columns is the set of columns 1-3, columns 4-6, or columns 7-9.

Definition. Call two completed sudoku grids *equivalent* if one can be obtained from the other by any sequence of the equivalence transformations below.

A solution grid contains a 16-clue puzzle if and only if all grids equivalent to it have a 16-clue puzzle. Therefore, it is enough to inspect any one representative from each equivalence class of grids.

Here are the equivalence operations.

1. permutation of the digits 1-9,
2. permutation of the rows. These come in two types:
   (a) permute the three rows within a given band,
   (b) permute the bands,
3. permutation of the columns. These come in two types:
   (a) permute the three columns within a given stack,
   (b) permute the stacks,
4. transposing the grid.

Rotations and reflections are already included in these.

A natural question to ask is: How many completed Sudoku grids are there up to equivalence? This is a natural mathematical question anyway, but it is a relevant question for us because we only need to inspect one grid from each equivalence class for a 16-clue puzzle.

4.2 Applying Burnside's Lemma

The set of all these equivalence transformations forms a group of order $9! \times 6^4 \times 6^4 \times 2$. The permutation part (i.e., when the digit permutations are omitted) is a group of order $6^4 \times 6^4 \times 2 = 3,359,232$. This group acts on the set of all completed sudoku grids. Determining the orbits and the stabilizers of this action can be done using Burnside's lemma from group theory. This was carried out by Ed Russell and Frazer Jarvis in 2006 [15], and they showed that there are exactly 5,472,730,538 solution grid equivalence classes, a result that was later also verified by others.

4.3 Enumerating Representatives

For our project it is not enough to know the number of equivalence classes. It is necessary to enumerate a set of representatives of the equivalence classes, and store these in a file. Glenn Fowler of AT&T Labs wrote a programme that enumerates all the inequivalent completed grids. Uncompressed, the catalogue of grids would require approximately 418 GB of storage space. Fowler also wrote a data compression algorithm to store the catalogue of grids in under 6 GB.[1] Fowler has kindly shared his executables and we have used them in order to generate the compressed catalogue. We were thus able to store the catalogue on a single DVD.

[1] Note the amazing compression rate—each grid takes on average only a little more than one byte.

5 Checker

5.1 Overall Strategy of Checker

The obvious algorithm for exhaustively searching a given sudoku grid for a 16-clue puzzle simply tests all subsets of size 16 of the given grid for a unique completion. Although such a *brute force* algorithm is effective in that the sudoku minimum number of clues problem could be solved that way, the actual computing power required would be far too great. In fact, even searching one grid only would take a long time since

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

Fortunately, with a little theory the number of possibilities to check can be reduced dramatically. Very briefly, it is possible to identify regions (subsets) in a solution grid, called *unavoidable sets*, that always have at least one clue from any proper puzzle contained in that grid. Therefore, when picking the first clue of a trial (candidate) 16-clue puzzle, one does not need to try out all 81 clues in the given grid; rather, one finds one smallest unavoidable (sub)set and then tries each element in this unavoidable set as the first clue of the puzzle being constructed. Similarly for all further clues.

So the overall strategy we use in our programme checker may be summarized as follows:

1. Find a sufficiently powerful collection of unavoidable sets for the given grid;
2. Enumerate all hitting sets of size 16 for this collection, i.e., enumerate all sets having 16 clues that intersect all the unavoidable sets found in step 1;
3. Check if any of the hitting sets found is a valid 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.

We explain each of these points in more detail now.

5.2 Unavoidable Sets For an Individual Grid

The idea of an unavoidable set originated on the sudoku forums, and is easily explained by the following example.

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

The reader can see that if 5 and 9 are interchanged *among the four red numbers only in rows 1 and 2, columns 1 and 5*, then a different valid completed sudoku grid is obtained. Therefore, in any sudoku puzzle with this grid as the only possible answer, one of the four red numbers *must* be a clue. Because, a puzzle not containing any of these four numbers as a clue would have at least two solutions and therefore would not be a valid puzzle. We say that the set of these four numbers is unavoidable—we cannot avoid having a clue from these four. This motivates the following definition.

Definition. Let G be a sudoku solution grid.[2] A subset X of G is called an *unavoidable set* if $G \backslash X$ (the complement of X) has multiple completions.

So if a set of clues does not intersect every unavoidable set, then it cannot be used as a set of clues for a sudoku puzzle because there will be multiple completions. Equivalently, any set of clues for a valid puzzle must use at least one clue from every unavoidable set. In fact, the converse is true as well:

Lemma 1. *Suppose that $X \subseteq G$ 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, contradiction. □

An unavoidable set is said to be *minimal* if no proper subset is itself unavoidable. Usually when we say unavoidable set we mean minimal unavoidable set.

---

[2]Formally, a sudoku solution grid (completed sudoku) is a function $\{0, \ldots, 80\} \to \{1, \ldots, 9\}$, 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 $\{0, \ldots, 80\} \times \{1, \ldots, 9\}$.

5.2.1 Finding minimal unavoidable sets

In the original version of checker, unavoidable sets in a given 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 4.1. On the forums, Ed Russell had investigated unavoidable sets and compiled a list of blueprints. We added all blueprints of size twelve or less from Russell's list to checker (525 blueprints in total).[3] When actually finding unavoidable sets, checker would simply compare each blueprint against all grids in the same equivalence class of the given grid, modulo the digit permutations. That is, checker would generate $3,359,232$ grids, as explained in Section 4, and for every grid generated checker would try each blueprint for a match. With a typical grid, this yields about 360 unavoidable sets in total.

Using the algorithm just described, finding unavoidable sets in a grid takes approximately half a minute. Five years ago this was not a major bottleneck 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 managed to efficiently enumerate the candidate 16-clue puzzles, in order to make this project feasible, obviously we also had to come up with a better algorithm for finding the unavoidable sets. While we kept our original strategy, again a bit of theory helps to significantly reduce the number of possibilities to check.

Lemma 2. *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 $G \backslash U$, then G and H differ exactly in the cells contained in U. In particular, every digit appearing in U occurs at least twice.*[4]

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. $\square$

Corollary 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 $G \backslash U$, then H may be obtained from G*

[3] Later we wrote a small tool called *unavpat*, which enumerated all blueprints of a given size. We used unavpat to prove that Russell's list (from 2005) already contained every possible blueprint of size up to 11.

[4] We say that a digit d, $1 \leq d \leq 9$, appears in U if there exists $c \in \{0, \dots, 80\}$ such that $(c, d) \in U$. Similarly we say that a cell c is contained in U if $(c, d) \in U$ for some d.

by a derangement[5] *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. □

The basic idea how to make the actual search for minimal unavoidable sets in a given grid faster is to replace the blueprints from Russell's list by appropriate members in the same equivalence class that are chosen such that only a fraction of grids equivalent to the given one need to be checked for a match.

More concretely, call a blueprint an $m \times n$ blueprint if it hits m bands and n stacks of the 9×9 matrix, and 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 each blueprints hits at most as many bands as it hits stacks, i.e, for an $m \times n$ blueprint we may always assume that $m \leq n$.

Suppose that $m = 1$, i.e., suppose that a blueprint hits only one band. By swapping bands if necessary, we may then assume that only the top band is hit. Note that $n \geq 2$, since by Lemma 2, a 1×1 blueprint cannot exist as any digit in a minimal unavoidable set appears at least twice. 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:

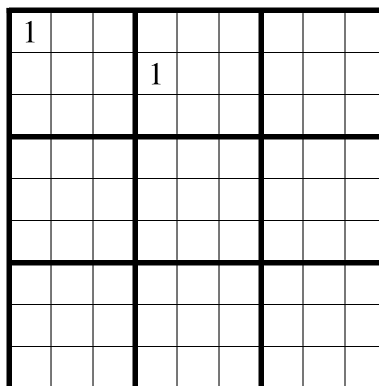

Figure 1: Two clues in any $1 \times n$ blueprint

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

[5]Recall that a *derangement* is a permutation with no fixed points, i.e., an element $\sigma \in S_n$ such that $\sigma(k) \neq k, \forall\, k = 1, \ldots, n$.

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, instead of 1,296 row permutations with the original algorithm.[6] For the columns, 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. Now 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 actually 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. The reader will have noticed that this will, with 50% probability, 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:

1		2	3	4				
4		3	1	5				
		5		2				

1		2	3		4			
4		3	1		5			
		5			2			

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 earlier, see Figure 1. 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 $3,359,232$ (equivalent) grids, in actual fact we only need to generate 324 grids.

We find 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. As with

[6] A consequence of this inefficiency of the original algorithm was that most unavoidables were found many times, especially so all 1×2 unavoidables.

the middle stack, we do not 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:

1		2				3	4	
4		3	1	2				
			3	4		2	1	

1		2				3		4
4		3	1	2				
			3	4		2		1

1		2				3	4	
4		3	1		2			
			3		4	2	1	

1		2				3		4
4		3	1		2			
			3		4	2		1

In order to tackle 2×2, 2×3 and 3×3 blueprints, we need the following result.

Proposition 4. *Every blueprint is equivalent to one containing the same digit twice in the same band.*

Proof. Later. □

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

following three types, again in order to reduce the number of possibilities to check (note that the first and the third type overlap):

1								
		2	1					
2								

1								
		2	1					
	2							

1								
2			1					
	2							

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

5.2.2 Higher-degree unavoidable sets

There are unavoidable sets that require more than one clue, which we call *higher-degree* unavoidable sets. Let us illustrate this with an example.

1			4			7		
4			7			1		
7			1			4		

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.

The above is an example of a degree 2 unavoidable set. If we say that an unavoidable set as defined earlier is an unavoidable set of degree 1, then we can recursively define the notion of an unavoidable set of degree k for $k > 1$.

Definition. A nonempty subset $U \subseteq G$ 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 $k - 1$.[7]

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 above example is a $(9, 2)$ unavoidable set that is the union of nine $(6, 1)$ unavoidable sets. Of course, one can easily construct higher-degree unavoidable sets, e.g., the union of any two disjoint minimal unavoidable sets is trivially an unavoidable set of degree 2. More generally, we make the following definition.

Definition. A minimal unavoidable set U of degree k is said to be *nontrivial* if there does not exist a minimal unavoidable set U_1 of degree k_1 and a minimal unavoidable set U_2 of degree k_2 and disjoint from U_1 such that $U = U_1 \cup U_2$; otherwise, we say that U is *trivial*.

So the above $(9, 2)$ unavoidable set is nontrivial. In fact, it is one of only two types of nontrivial $(9, 2)$ unavoidable sets, the other being this one:

1			2			3		
2			1			4		
3			4			1		

[7] An equivalent (nonrecursive) definition would be to say that U is unavoidable of degree k if for all combinations of distinct $c_1, \ldots, c_{k-1} \in U$, the set $U \backslash \{c_1, \ldots, c_{k-1}\}$ is an unavoidable set in the earlier sense. So in this project we have proved that any sudoku solution grid is unavoidable of degree 17.

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 $(11, 2)$ unavoidable sets, which however we did not use in this project.[8] Naturally we have the following result.

Proposition 5. *Let $U \subseteq G$ be an (m, k) unavoidable set. Then we need to add at least k elements from U to $G \backslash U$ to obtain a sudoku puzzle with a unique completion. Moreover, if $V \subseteq G$ is an (m', k') unavoidable set such that $U \cap V = \emptyset$, then $U \cup V$ is an $(m + m', k + k')$ unavoidable set.*

Proof. Both claims follow directly from the definition and by using induction on k respectively $k + k'$. □

Repeated application of the second part of this proposition gives the following useful fact.

Corollary 6. *Suppose that $U_1, \ldots, U_t$ are degree k unavoidable sets of a sudoku solution grid G that are pairwise disjoint. Then $U_1 \cup \cdots \cup U_t$ is an unavoidable set of degree $t \cdot k$.*

Example

Using higher-degree unavoidable sets we give an example of a sudoku 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

Lemma 7. *Any sudoku puzzle whose solution is this grid has at least 18 clues.*

[8] Note how any minimal $(11, 2)$ unavoidable set is necessarily nontrivial as there are no minimal $(m, 1)$ unavoidable sets for $m = 1, 2, 3, 5, 7$.

Proof. Upon inspection, the grid is the union of nine pairwise disjoint (9,2) unavoidable sets. By the previous corollary this grid is therefore an (81,18) unavoidable set and thus requires at least 18 clues for a unique solution. □

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.

5.3 Enumerating the Hitting Sets For an Individual Grid

Now that we have shown how to quickly find a sizable collection of unavoidable sets for a given grid, it is time that we finally explained how to efficiently enumerate the hitting sets of size 16 for such a family of unavoidable sets. For this section, we fix a completed sudoku grid and work within it.

Definition. Given a collection of subsets of clues (the unavoidable sets), a *hitting set* for this collection is a set of clues that intersects every one of the subsets.

Recall that the hitting set problem is known to be computationally hard—in fact it is one of R. Karp's twenty-one classic NP-complete problems [13]. Thus it was likely that a standard backtracking algorithm would not nearly be fast enough to search all grids in the catalogue, which indeed turned out to be the case. We should also mention that a common way to approach the hitting set problem is actually a greedy algorithm; however, as the result such an algorithm produces is only approximate it is of no use in solving the sudoku minimum number of clues problem.

5.3.1 Algorithm of the original checker

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 this algorithm can be described in one sentence—take one unavoidable set that has not been hit and branch in all possible ways; repeat recursively until 16 clues have been picked in this way.

That is, at each step, we first find one (smallest) unavoidable set that does not contain any of the clues picked so far, and then try each element of this unavoidable set as the next clue. We keep doing this until 16 clues have been chosen. If our collection of unavoidable sets is exhausted before we get to the 16th clue, then we simply add the remaining clues needed in all possible ways.

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 $\mathcal{F} = \{U_1, \ldots, U_m\}$ where $U_i \subseteq \{0, \ldots, 80\}$ and $\#U_i \leq \#U_{i+1}$. Then for each 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 $S \subseteq \{0, \ldots, 80\}$ the function $\mathcal{X}_S : \{0, \ldots, 80\} \to \{0, 1\}$ 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 ith 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:

```
proc InitHittingVectors()
  for c from 0 to 80
    hitvec[c] := (X_{U[1]}(c),...,X_{U[m]}(c));
  end for
end proc
```

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, $\mathrm{statevec[0]}, \ldots, \mathrm{statevec[16]}$. 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 kth step, $0 \leq k \leq 15$, then we simply set

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

What this means is that we first save the clue c to the array hitset and then perform componentwise (bitwise) boolean OR on the binary vectors $\mathrm{statevec[k-1]}$ and $\mathrm{hitvec[c]}$ and store the result in $\mathrm{statevec[k]}$. So $\mathrm{statevec[k]}$ contains a 1 in the ith slot if and only if either $\mathrm{statevec[k-1]}$ or $\mathrm{hitvec[c]}$ has a 1 in the ith 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 either $k = 16$ or $\mathrm{statevec[k]} = (1, \ldots, 1)$ for some $k < 16$. In the latter case, i.e., if at some stage we find that all the unavoidable sets in our collection have been hit, we add $16 - k$ more clues to the hitting set *in all possible ways* and then check the resulting 16-clue puzzles for a unique solution, by running them through a sudoku solver procedure, see Section 5.4.

At the beginning of this subsection, we said that there was one small improvement (over the obvious hitting set algorithm) that we already used in the original checker and which we will describe now. 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 very 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 $\{0, 3\}$.

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). So the pseudocode for the procedure that initially sets up the required binary vectors is as follows:

```
proc Initialize()
  // first set up vectors needed for the dead clues
  for each U in F // F = family of unavoidable sets
    for each c in U
      U.deadclues[c] := (0,...,0);
      for each d in U
```

```
        if d <= c then
          U.deadclues[c].SetBit(d);
        end if
      end for
    end for
  end for
  deadvec[0]  := (0,...,0);
  // now set up hitting vectors, see above
  InitHittingVectors();
  statevec[0] := (0,...,0);
end proc
```

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

```
proc AddClues(k)
  if statevec[k] = (1,...,1) then
    // all unavoidable sets are hit, add 16-k clues
    // in all possible ways and then check each
    // puzzle for a unique completion
    GeneratePuzzles(k,hitset,deadvec[k]);
    return;
  else if k=16 then
    // already drawn 16 clues, but some unavoidable
    // sets are still unhit, so nothing to do
    return;
  end if
  // 'F' is the family (array) of unavoidable sets
  // pick the first unavoidable set that is unhit
  U := F[statevec[k].GetIndexOfLowestZeroSlot()];
  for each c in U
    // first verify that this clue is still alive
    if deadvec[k].GetBit(c) = 0 then
      // add clue to hitting set
      hitset[k+1]   := c;
      // update the state vector (which keeps track
      // of which unavoidable sets are already hit)
      statevec[k+1] := statevec[k] OR hitvec[c];
      // exclude all smaller clues from the search
```

```
      deadvec[k+1]  := deadvec[k] OR U.deadclues[c];
      // recursively add the remaining clues
      AddClues(k+1);
    end if
  end for
end proc
```

We take a moment to convince ourselves that the above algorithm does not miss any hitting sets. So 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 by checker 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 U contained in P. 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 16th clue in this way, our hitting set will equal P, and since P hits all unavoidable sets, in particular, all unavoidable sets contained in the family $\mathcal{F}$ that checker uses, no further unavoidable sets are available, so that checker will test the set P for a unique completion, 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 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

for all 17-clue puzzles. To this day, this particular grid holds the world record as the grid having the largest known number of 17-clue puzzles (29, all found by Gordon Royle). Back in 2005, this grid was considered a likely candidate to have a 16-clue puzzle, but using checker we proved that none existed in it. 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 the grid in less than a week for all 17-clue puzzles. The result was that Gordon Royle had already found all of them, i.e., it was now known that there are *exactly* 29 different 17-clue puzzles contained in the grid.

5.3.2 Algorithm of the New checker

As far as the algorithm is concerned, there are really three differences between the original checker and the version we used for searching all grids in the catalogue.

The first one is that we added (trivial) 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 16th clue as we know that at least two more clues are required for a hitting set.

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.

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 optimal. 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.

The first of the above changes 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. We will now discuss all three improvements in detail. We begin with the third change, for which we need the

following definition.

Definition. A collection of pairwise disjoint unavoidable sets in a sudoku solution grid is called a *clique*. If there is no bigger clique (having a greater number of unavoidable sets), then we further say that the clique is *maximal*.[9]

In the original checker, the procedure that enumerated all the hitting sets would actually first find a maximal clique among the unavoidable sets supplied to it; say m is the size of this clique. It would then take the cartesian product of the unavoidable sets the clique consisted of. Every element (m-tuple) of this cartesian product would be considered individually, and a further $16 - m$ clues were added by drawing more clues from other unavoidable sets, as described in the previous subsection. The first thing to notice is that using a clique in this way is not ideal, because a maximal clique will often involve unavoidable sets of size 10 or bigger, whereas even after drawing as many as 15 clues, the smallest unavoidable set not yet hit in most cases has just six or maybe eight elements. Therefore it is better to just choose the next unavoidable set for drawing clues from to be one of smallest size among those not yet hit. So from our array of minimal unavoidable sets, we pick the one of lowest index that does not contain any of the clues drawn so far. Since this array is ordered by size, this will automatically yield a set of smallest size. However, as pointed out already, this is still 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. For this reason, in the new checker, when selecting the first ten clues we always use an unavoidable set of minimum effective size.[10] (Here, the effective size of an unavoidable set is the number of clues it has that are not yet dead.) 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 set contains (i.e., in the

[9] This terminology comes from graph theory—in the original checker, a maximal 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. A standard clique algorithm was then used to find a maximal clique in this graph. 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.

[10] 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 5.5 "Tradeoffs".

latter vector we set all slots corresponding to dead clues to zero). We finally obtain the Hamming weight of the resulting vector, which is equal to the effective size. As we do this for all unavoidable sets, we remember the index of the first set that had the minimum effective size.

Next we will explain the most important of the three improvements we made to the algorithm. It is about how the use of trivial higher-degree unavoidable sets enables us to (considerably) prune the search tree. Directly from the definition, if, after drawing k clues, there is an unavoidable set of degree $17 - k$ 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, by Corollary 6, the union of the sets in a clique of size m is an unavoidable set of degree m. Therefore, right before we begin the search, we obtain a sizeable 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. By what we just said, after 12 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, respectively 15 clues. This may seem like an obvious way to prune the search tree with the hitting set problem, however, back in late 2008 when we first realized that the above would allow us to dramatically speed up checker[11], this was not yet described anywhere in the literature, even though, like the other two improvements we made, as well as the dead clue vector, it is not at all specific to sudoku but applies in an equal manner to the general hitting set problem. Actually the first public mention of this idea, to our knowledge, was in a posting of July 23rd, 2010 to the sudoku programmers' forum by Mladen Dobrichev, who had just released his open-source tool *GridChecker*:

> "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, what

[11] By mid-2009, we had a version of checker searching for 15-clue puzzles up and running that used higher-degree unavoidables just as described above (references available on request).

he does not seem to have fully realized is the power of this idea—although GridChecker does use trivial higher-degree unavoidable sets, with the exception of the degree 2 unavoidable sets, even in the very latest release of GridChecker apparently only a relatively limited collection of higher-degree unavoidable sets is being used, namely those coming from the members of a maximal clique.[12] 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 $\mathrm{chunkProcessor} :: \mathrm{iterateClue}$, 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 $\mathrm{chunkProcessor} :: \mathrm{iterateClueBM}$.)

Earlier we said that the idea of using trivial higher-degree unavoidable sets was not described in any research article at the time we started work on the new checker (in 2008). This changed in November 2010, when I-Chen Wu et. al. published the paper [18]. In this work, in Lemma 2 it is proven that the search for an n clue puzzle in a sudoku solution grid can be stopped after selecting m clues provided that there are at least $n - m + 1$ unavoidable sets that are not yet hit ($n - m + 1$ "active" unavoidable sets, in the language of that paper). The authors further describe how they used this result so as to speed up our original checker by a factor of 129 and hence achieve a running time of 13.9 seconds per sudoku solution grid on average, a remarkable improvement. As far as the problem of finding a clique of a certain size of active unavoidable sets is concerned, they point out that the maximal clique problem is itself NP-complete (like the hitting set problem), and that they therefore use a greedy algorithm for trying to construct cliques of the required size.

However, this is not the most efficient way, and it is likely the main reason why our own, new checker is about twice as fast as the checker written by Wu and colleagues. For, constructing new cliques over and over again, even just small ones, means duplicating effort. On the other hand, in our new checker we compute a large number of cliques of all sizes less than or equal to five *just once* at the beginning of the search, and keep track of which ones are hit (become inactive) as we add more clues. The consequence is that, e.g., after drawing twelve clues, we merely have to do a boolean OR of two binary vectors (of length 1,536) and then check if the resulting vector has a 1 in every slot in order to find out if there is an clique of size five.[13] All that can be done quite efficiently using SIMD programming, whereas constructing a clique of size 5 from scratch

[12]So GridChecker also uses a maximal clique, however, it does so in a much more clever way than our original checker.

[13]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.

is certainly more work and in particular involves much 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 11 clues are picked), but on the other hand, a greedy algorithm will often miss cliques of the desired size even though they exist. On balance, it seems that ours is the more efficient approach.

Now is a good time to summarize exactly what we actually 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 $\mathrm{U}[1], \ldots, \mathrm{U}[m]$ in our initial family of minimal unavoidable sets. We add the following statements to the procedure InitHittingVectors, see p. 24.

```
 var   count := 0;
 const START := 27;
 for i from START to m
   for j from START-1 to i-1
     if U[i] intersects U[j] then
       continue;
     end if
     for k from START-2 to j-1
       if U[k] intersects either U[i] or U[j] then
         continue;
       end if
       for l from START-3 to k-1
         if U[l] intersects none of U[i],U[j],U[k] then
           // found a clique of size 4
           CLQ[count] := union(U[i],U[j],U[k],U[l]);
           inc(count);
           if count = 32768 then
             goto SetUpHittingVectors;
           end if
         end if
       end for
     end for
   end for
 end for
SetUpHittingVectors:
 for c from 0 to 80
```

```
  quadhitvec[c] := (X_{CLQ[0]}(c),...,X_{CLQ[count-1]}(c));
end for
```

Here CLQ is an array of sets, $\mathrm{quadhitvec}$ is an array of binary vectors of length 32,768, and $\mathcal{X}_{\mathrm{CLQ[i]}} : \{0, \ldots, 80\} \to \{0, 1\}$ is again the indicator function for the set $\mathrm{CLQ[i]}$ with respect to $\{0, \ldots, 80\}$. Finally, START is a constant used 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 $\mathrm{U}[1]$, for example, since the first unavoidable set in our array will always be used for drawing the first clue from, so will in particular be hit by the time we check if there is an unhit clique of size 4.

The final improvement we made to the 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 in checker in the innermost loops. 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*. In a future version of this document we will describe exactly how this works. For now we refer the reader to the procedure $\mathrm{ConsolidateHitvec}$ in the file $\mathrm{checker.cpp}$ of the checker source code.

5.4 Using A Sudoku Solver

We require a sudoku solver for checker, because we run all hitting sets that survive through the solver to see if they have a unique completion. In fact, we only need to test if a hitting set has more than one completion, so once two completions are found, we discard that hitting set and move to the next one. We do not need the exact number of completions.

In the original version of checker we used Guenter Stertenbrink's public domain solver. For this computation, however, we switched to the open-source sudoku solver written by Brian Turner, which is available online. This solver can check around 50,000 16-clue puzzles per second for a unique completion, and at the time we had to make a decision which solver to use with the new checker, Turner's solver was the fastest available, to our knowledge.

5.5 Tradeoffs

During the development of checker, there were several design choices that involved tradeoffs. That is, there were often several alternatives that each had their advantages, so we

had to find out the 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. For example, we also tried adding unavoidable sets of size 13 to checker, but it made almost no difference to the running time one way or another.
- 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.

There are other trade-offs, which we will describe in a later version of the article.

5.6 Some remarks on the implementation of the new checker

During much of this chapter we described how we made checker faster by improving the algorithm. Of course, once one has decided on an algorithm it is also possible to improve the running time by making changes to the *implementation*. The main change to the implementation (compared to the original checker) 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.

5.7 Testing 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 known sudoku grids containing any 16-clue puzzles). So to be able to do at least some amount of testing, we produced a version of checker that searches for 17-clue puzzles. 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.

Since there is also one known pseudo 16-clue puzzle, i.e., a puzzle with 16 clues having *two* solutions, as a test we also modified checker such that it would find puzzles with two solutions. We ran it on the aforementioned grid and the pseudo 16-puzzle was found.

As far as white-box testing goes, we have applied state-of-the-art software-engineering practices. In particular, using the debugger we stepped through every single line of the code, carefully verifying that each instruction does exactly what we thought it would do. For further debugging, we also used the tool *valgrind*, as well as its companion *cachegrind*, the latter mainly for optimizing memory accesses.

Moreover, to be on the safe side, 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.

Similarly, 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. However, this situation never actually happened with any grid, i.e., the answers the solver provided were always correct. (Should that ever not have been the case, then the grid in question would have been logged.)

6 Running Through All Grids: Further Details about Code Performance and the Computation

With checker in hand, the actual search involved running through the catalogue of all grids, and executing checker on each grid. Because of the number of grids (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.

6.1 Platform

As mentioned already, during our PRACE prototype project in late 2009 [3], we were able to try an early version of the new checker on four different platforms (AMD Istanbul, IBM Blue Gene/P, IBM Power 6, Intel Nehalem), and found out that Nehalem is the best for us. Therefore we totally optimized checker for Nehalem, to the extent of rewriting much of the code in assembly language (originally checker was written completely in C++). We did this in such a way so as to maximize simultaneous use of the various execution units of a Nehalem core (instruction-level parallelism). We further heavily used SSE/SIMD 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. Finally, we were of course able to take advantage of the SMT mode of a Nehalem CPU (hyper-threading), which was enabled on the cluster *Stokes* at ICHEC that we used for this computation.

Note that the different tasks (grids) to be considered with this project were completely independent of each other, i.e., our computation was *embarrassingly parallel*. So the parallelism — utilizing hundreds or even thousands of processor cores simultaneously — of this project was almost trivial. We only had one type of run, which scaled linearly with the number of processors. Moreover, we were always entirely flexible on the maximum duration of a run and on the number of cores allowed per run. The only limitation was that there had to be one MPI process that managed the run (the *master*) and which was thereafter unavailable for the actual computation. However, there was nonetheless no significant loss of efficiency since the SMT mode compensated very well for this.

6.2 Load Balancing and Task Farming

One nontrivial aspect of this computation was the load-balancing[14], as there was considerable variability in the running time of checker, depending on the grid — approximately

[14] Load-balancing refers to the distribution, or *task farming*, of the jobs by a master node to the slave nodes.

20% of all solution grids consumed roughly half of the total running time, and the remaining 80% of grids the other half. Furthermore, for optimal efficiency, we naturally wanted all slave nodes to finish around the same time.

During our PRACE prototype run we used task farming software developed by Gilles Civario to run an early version of checker on 64 racks (262,144 CPUs) of JUGENE simultaneously. This worked very well, so the load-balancing problem problem was solved with this software. Moreover, as the time required for analyzing one sudoku solution grid is typically of several orders of magnitude shorter than the duration of one job, the usage of the CPU cores was near-optimal, i.e., almost no processor time was lost at the end of a job. Therefore the actual speedup gained by increasing the number of nodes was practically equal to the theoretical (ideal) speedup.

6.3 The Actual Computation

The entire computation took about 7.1 million core hours on the Stokes machine. Stokes is an SGI Altix ICE 8200EX cluster with 320 compute nodes. Each node has two Intel (Westmere) Xeon E5650 hex-core processors and 24GB of RAM. We divided the computation up into several hundred jobs. We started running jobs in January 2011, and we finished in December 2011.

7 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.

References

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

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

[3] http://www.prace-project.eu/news/six-projects-granted-access-to-the-prace-prototype-systems-4-5-million-core-hours

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

[5] http://model.research.glam.ac.uk/projects/sudoku/

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

[7] Laura Taalman, *Taking Sudoku Seriously*, Math Horizons, Volume 15, September 2007, pp. 5–9, http://www.math.jmu.edu/~taal/sudoku_mathhorizons.pdf

[8] Agnes M. Herzberg and M. Ram Murty, *Sudoku Squares and Chromatic Polynomials*, Notices of the AMS, Volume 54, Issue 6, pp. 708–717, June/July 2007, http://www.ams.org/notices/200706/tx070600708p.pdf

[9] Jean-Paul Delahaye, *The Science behind Sudoku*, Scientific American, June 2006, pp. 80–87, http://www.cs.utexas.edu/~kuipers/readings/Sudoku-sciam-06.pdf

[10] Max Neunhöffer, *Is there a Sudoku puzzle with 16 clues?*, Lecture at the University of Aberdeen, March 2010, http://www-groups.mcs.st-and.ac.uk/~neunhoef/Publications/pdf/sudoku16_aberdeen.pdf

[11] Faisal N. Abu-Khzam, *Kernelization Algorithms for d-Hitting Set Problems*, LNCS 4619, pp. 434-445, 2007.

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

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

[14] Bertram Felgenhauer and Frazer Jarvis, *Mathematics of Sudoku I*, Mathematical Spectrum, Volume 39, Number 1, pp. 15–22, September 2006, http://www.afjarvis.staff.shef.ac.uk/sudoku/felgenhauer_jarvis_spec1.pdf

[15] Ed Russell and Frazer Jarvis, *Mathematics of Sudoku II*, Mathematical Spectrum, Volume 39, Number 2, pp. 54–58, January 2007, http://www.afjarvis.staff.shef.ac.uk/sudoku/

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

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

[18] 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.