View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by St Andrews Research Repository

# Using Small MUSes to Explain How to Solve Pen and Paper Puzzles

**Joan Espasa**[1] , **Ian P. Gent**[1] , **Ruth Hoffmann**[1] , **Christopher Jefferson**[1] , **Alice M. Lynch**[1]
[1]University of St Andrews
{jea20,ian.gent,rh347,caj21,al254}@st-andrews.ac.uk

arXiv:2104.15040v1 [cs.AI] 30 Apr 2021

## Abstract

Pen and paper puzzles like Sudoku, Futoshiki and Skyscrapers are hugely popular. Solving such puzzles can be a trivial task for modern AI systems. However, most AI systems solve problems using a form of backtracking, while people try to avoid backtracking as much as possible. This means that existing AI systems do not output explanations about their reasoning that are meaningful to people. We present DEMYSTIFY, a tool which allows puzzles to be expressed in a high-level constraint programming language and uses MUSes to allow us to produce descriptions of steps in the puzzle solving. We give several improvements to the existing techniques for solving puzzles with MUSes, which allow us to solve a range of significantly more complex puzzles and give higher quality explanations. We demonstrate the effectiveness and generality of DEMYSTIFY by comparing its results to documented strategies for solving a range of pen and paper puzzles by hand, showing that our technique can find many of the same explanations.

## 1 Introduction

Puzzles like Sudoku, Futoshiki or Skyscrapers are designed to be solved on paper and continue to be incredibly popular. New variants of these puzzles are created almost weekly, and there are many websites and books dedicated to showing off new problems. The increasing popularity of the YouTube channel 'Cracking the Cryptic' shows that people enjoy seeing explanations of pen and paper puzzles. There exist specialised guides for solving many of these puzzles [Conceptis, 2002; Tectonic, 2005; Stuart, 2008; Wilson, 2006] as well as popular variants of them. These guides provide us with a reference to compare our techniques against.

Most of the paper and pen types of puzzles can be trivially solved when using a constraint solver [Simonis, 2005]. This is due to propagators which enforce various methods of consistency between subsets of the variables or constraints in the problem. However, this method has two major limitations. If we use weak propagators, we produce search trees with hundreds or thousands of nodes. If we use strong propagators we can make deductions beyond the abilities of most human players, while still often producing search trees. Whereas human players aim to solve problems with no backtrack.

There are two main reasons to look at how humans solve puzzles – to advise players on how to progress and to produce more accurate difficulty measures of puzzles. Some works [Pelánek, 2011] try to measure the difficulty of a puzzle by recording both the number and difficulty of deductions which can be applied at each point in solving.

If we want to be able to explain how a puzzle is solved, a common approach is to create custom solvers which use the same techniques as human players. For popular puzzles this is easy, as the techniques which human players use are well documented. However, this means that for each variant a custom solver has to be implemented. SudokuWiki [Stuart, 2008] is an example of such solvers for several popular Sudoku variants, showing which techniques can be applied at each stage of solving. The major limitation of these systems is the requirement for a hard-wired and ordered list of techniques. This paper provides a more general technique, based on MUSes (Minimal Unsatisfiable Subsets), which we will demonstrate on a variety of pen and paper puzzles.

When considering explanations, interpretability is defined as "descriptions that are simple enough for a person to understand, using vocabulary meaningful to the user" [Gilpin *et al.*, 2018]. We advance on the work of [Bogaerts *et al.*, 2020] in using MUSes as a basis for providing interpretable explanations of puzzles from a SAT model. However, standard MUS finding algorithms are too inefficient for larger puzzles such as Sudoku.

Our contribution is threefold. First, a novel MUS-finding algorithm optimised to find individual small MUSes. Second, improved techniques for using MUSes to generate explanations designed for pen and paper puzzles. Finally, we provide a comparison of explanations generated via MUSes to real-world tutorials and puzzle solving, showing how our techniques closely match the explanations used by real players on a variety of puzzles and guides. We also experimentally explore the efficiency of the proposed MUS finding algorithm.

## 2 Background

Most puzzles which we will discuss in this paper will be typically solved by keeping a list of the values which are being considered for each cell, called the *candidates*. Once every

cell has only one candidate remaining, the puzzle is solved. We consider puzzles which have only a single solution and are intended to be solved by humans without having to guess. We call these *pen and paper puzzles*.

*Binairo*, also known as Takuzu, is a grid where the cells are to be filled with 0s and 1s such that: each row and column has an equal number of 0s and 1s, all rows or columns are different and no more than two of either number are adjacent.

*Futoshiki* is a puzzle with an $n \times n$ grid, containing inequality constraints between the cells. The aim is to fill in the cells, with the numbers from 1 to $n$ contained in each row and column exactly once, while upholding the inequalities.

*Kakuro* is a crossword style puzzle, with numbers instead of words. Each cell will contain a digit from 1 to 9, such that the cells add up to the hint given for that sub-row (or sub-column). Each sum consists of unique digits.

*Skyscrapers* is an $n \times n$ grid based puzzle where the numbers 1 to $n$ represent skyscrapers of different heights: 1 being the lowest, $n$ being the highest. The grid represents a top-down view of the skyscrapers. The rules are that in each row or column each number must occur exactly once and the numbers around the edges indicate how many buildings are visible on that row or column looking from that edge. Therefore, higher skyscrapers will hide smaller skyscrapers, looking from one edge, but the smaller could be visible from the opposite edge.

*Sudoku* is a logic puzzle with the goal of filling a $9 \times 9$ grid of cells, where each cell contains a number between 1 and 9, so that each column, row, and defined $3 \times 3$ box (usually marked with a bold outline) contain each of the numbers 1 to 9 exactly once. The grid contains some initial values and has a single solution. The popularity of Sudoku has led to many variants. These variants keep an $n \times n$ grid where each row and column contains the numbers from 1 to $n$, but vary the shape of the boxes, or add extra constraints.

*Starbattle* is a puzzle requiring each row, column and region to contain the same number of stars. The number of stars can range from 1 to n, each puzzle indicates how many stars are required. Stars cannot be placed in adjacent cells (orthogonally or diagonally).

*Tents and Trees* has a grid containing some cells filled with a tree, and an indication of the number of tents required in each row and column. The goal is to fill in the remaining cells with either a tent or grass, such that each tree has an orthogonally adjacent tent, the number of tents reflect the expected number for each row and column, and no two tents are touching (orthogonally and diagonally).

*Thermometer* is a grid filled with thermometers of different lengths. The goal is to partially fill each thermometer with mercury from the "bottom" (a cell with a rounded end). Thermometers do not have to be filled to the top. The hints on the outside edges of the grid indicate how many mercury filled cells there are in the corresponding column or row.

### 2.1 Minimal Unsatisfiable Sets

**Definition 1** (Unsatisfiable Set/Core)**.** *An* unsatisfiable set *of an unsatisfiable constraint problem is any unsatisfiable subset of the set of constraints of the problem.*

Traditionally, unsatisfiable sets are defined on the clauses of a conjunctive normal formula. In this paper, we extend this definition to general constraint problems. A given problem can have many unsatisfiable sets of different sizes. The hypothesis of our work is that unsatisfiable sets closely align with how human players solve puzzles. Unsatisfiable sets have many uses, such as on interactive applications [Junker, 2001], repairing knowledge bases [Mazure *et al.*, 1998] or model checking [McMillan, 2003]. See [Silva, 2010; Cimatti *et al.*, 2011] for extensive surveys.

Identifying *minimum* unsatisfiable sets is a $\sum_2$-complete problem [Gupta, 2002], there are some attempts [Lynce and Silva, 2004; Zhang *et al.*, 2006], FORQES [Ignatiev *et al.*, 2015] at addressing this. On the other hand, the decision problem of a *minimal* (i.e. irreducible) unsatisfiable set can be formulated as the difference between two NP problems, which lies in the DP-complete [Papadimitriou and Wolfe, 1988] category. Different approaches have been studied for computing *Minimal Unsatisfiable Sets* (*MUS*) such as insertion-based [de Siqueira N. and Puget, 1988], deletion-based [Chinneck and Dravnieks, 1991], dichotomic [Hemery *et al.*, 2006] or by progression [Marques-Silva *et al.*, 2013]. As most techniques that require unsatisfiable sets do not strictly require them to be minimum, most specialised tools such as Muser2 [Belov and Marques-Silva, 2012], HaifaMUC [Nadel *et al.*, 2013], MCSMUS [Bacchus and Katsirelos, 2015], TarmoMUS [Wieringa and Heljanko, 2013] or some modern SAT solvers try to find minimal or even small enough unsatisfiable sets, striking a balance between size and practicality.

## 3 Model Augmentation

The rules for many well-known pen and paper puzzles can be expressed as constraints. For example, a Sudoku is built from *AllDifferent* constraints, one on each row, column and $3 \times 3$ box. Kakuro rules add sums, Futoshiki need inequalities and the Tents and Trees rules count elements. Other commonly found constraints can be easily modelled using first-order logic with equality.

To be able to give explanations for each reasoning step, all constraints used to model the rules of puzzles are half-reified. They take the form $x \rightarrow c$, where $c$ is the constraint and $x$ is a Boolean variable that controls if the constraint is active. Each constraint is then associated with a string, describing in natural language terms what the constraint is expressing.

When providing an explanation where $c$ is involved, we use the associated string to be able to give a meaningful explanation to the user, and highlight in a user interface all literals (candidate assignments) involved in $c$.

**Example 1.** *Given the constraint $c_1 : v_2 + v_3 \geq 5$, it is half-reified as $x_1 \rightarrow v_2 + v_3 \geq 5$. Boolean variable $x_1$ will be used to determine if the constraint $c_1$ is active. The string "the second cell plus the third cell must be greater or equal than five" is associated with the constraint, so it can be later*

*used to give meaningful explanations.*

We use SavileRow [Nightingale *et al.*, 2015] to automatically translate high-level models of puzzles into SAT. We add annotations to SavileRow's input to mark the purpose of the variables of the problem. Variables can be labeled as either *Problem*, *Constraint* or *Auxiliary*. *Problem* variables represent the answer to the puzzle, usually expressed as a grid. *Constraint* variables must be Boolean and are used for activating the constraints of the puzzle. Each *Constraint* variable is also labelled with a string which describes it. Finally, *Auxiliary* variables are only used to allow easier modelling of the puzzle. SavileRow was extended to output which SAT variable is used to represent the possible assignments to each labelled variable.

## 4 MUSes for Explaining Puzzles

DEMYSTIFY[1] generates explanations very similarly to [Bogaerts *et al.*, 2020], which applies the techniques to logic grid puzzles. Below is a schematic description showing our general procedure of explaining decisions when solving a puzzle.

1. Translate the description of the puzzle rules to a CNF formula $P$ (we use SavileRow [Nightingale *et al.*, 2015] for this step). This translation produces, amongst other things, a set of Boolean variables $L$: one for each value which can be assigned to each *problem* variable. There is also the set $X$ of Booleans which represent the *constraints* variables of the puzzle. As explained in Section 3, each $x \in X$ is associated with a constraint of the puzzle $c$, where $x \rightarrow c$. Therefore a MUS of the puzzle can be represented as a subset of $X$ which, when all assigned TRUE, makes $P$ unsatisfiable.
2. For each $l \in L$ take its value $a$ in the unique solution, find MUSes for the problem $P \wedge (l \neq a)$.
3. Pick a variable $l \in L$ which has the best MUS and display this to the user. Our criteria for picking the best MUS uses the following ordering: First, choose the MUS with the fewest constraints. Break ties by choosing the MUS whose constraints refer to the fewest literals. Finally, choose the MUS which can be used to discard the most literals.
4. Assign any literals which can be deduced from the best MUS and iterate from step 2 until all variables are assigned.

DEMYSTIFY uses Glucose [Audemard and Simon, 2018] as the underlying SAT engine. Learned clauses are kept between calls to the solver, which helps to speeds up the time taken to find many unsatisfiable sets [Audemard *et al.*, 2013].

There are several choices we make when presenting MUSes to the user, which reduce information overload and allow us to solve the puzzles in fewer steps. These settings can all be configured, depending on the preferences of the user, or the particular puzzle being solved.

Firstly, MUSes of size 1 are grouped together, as there can be many such MUSes and they are very simple to understand.

[1]https://github.com/stacs-cp/demystify

Secondly, for each MUS we generate a list of the literals contained in it. This is generally very fast, as the MUS is already small and we only need to check literals contained in at least one of the constraints in the MUS. A single MUS can often be used to deduce many literals.

MUSes are displayed to the user by listing the English descriptions of the constraints. The literals contained in the constraints of the MUS are highlighted in the user interface in one colour, while the literals removed by the MUS are highlighted in another colour. The literals contained in each constraint are highlighted by moving the mouse over the constraint. Providing a higher quality interface, which can help users better interpret MUSes, is left to future work.

## 5 MUS Algorithms

As previously discussed, there are many existing algorithms for finding MUSes. We found existing state-of-the-art techniques for finding all MUSes either did not finish on our instances or could only produce MUSes where each constraint is a single SAT clause.

Furthermore, we do not wish to find the smallest MUS for a single problem but to find the globally smallest MUS for a *set* of problems – one for each remaining unassigned problem variable. At each step of solving many of these problems will have no small MUSes, so we search all problems for MUSes below a parameter $MaxSize$, which is increased only if no MUSes are found.

Our algorithms require a SAT solver that provides a `FindUnsatCore` function. This function should accept a SAT problem and list of Boolean variables, and return FAIL if there is a solution where all those variables are TRUE, or a subset of $X$ which, if all assigned TRUE, lead to an unsolvable problem. This set is not necessarily minimal. Most modern SAT solvers provide this functionality.

We first give `BasicMUS`, a variant of the deletion-based algorithm of [Dershowitz *et al.*, 2006] in Algorithm 1. This algorithm accepts a problem $P$ and a set of variables $X$ (representing the constraints) and tests removing each element of $X$ in turn, checking the result is still unsolvable. It uses `FindUnsatCore` to reduce the unsolvable subset of $X$ at each step. The only new feature is keeping track of the known required members of the MUS and stopping once $MaxSize$ members have been found and checking if they form a MUS, if not we need more values so we can return FAIL.

Our experiments demonstrate a limitation of `BasicMUS`, which is the lack of variety in the MUSes it returns. The `ManyChop` algorithm, given in Algorithm 2, is intended to increase MUS variety. This algorithm uses `BasicMUS` after performing some initial reduction of $X$. This algorithm works by removing random subsets of $X$ and checking if the result is still unsatisfiable. `ManyChop` chooses a fixed-size proportion of $X$ to remove and keeps trying to remove that many elements of $X$ and checking for unsatisfiability. The idea behind the algorithm is that given a set $X$, if we remove some proportion $p$ of the elements of $X$, the probability any particular value is left behind is $1 - p$. Therefore, the chance that any fixed collection of $n$ elements remains behind is approximately $(1-p)^n$. This approximation is accurate enough

**Algorithm 1** Basic MUS finding algorithm

```
1:  procedure BasicMUS(P, X, MaxSize)
2:      X = FindUnsatCore(Shuffle(X))
3:      MusSize = 0                    ▷ Values known to be in MUS
4:      ToConsider = ShuffledCopy(X)
5:      for c ∈ ToConsider do
6:          if c ∈ X then
7:              core = FindUnsatCore(P, X − c)
8:              if core == FAIL then
9:                  MusSize += 1       ▷ c must be in the core
10:                 if MusSize == MaxSize then
11:                     X = X[1..MaxSize]
12:                     if FindUnsatCore(P,X) == FAIL then
13:                         return FAIL
14:                     else
15:                         return X
16:             else X = core
17:     return X
```

for the values of $|X|, p$ and $n$ used in our experiments.

In our experiments, we choose $p$ such that there is a probability of at least $\frac{1}{10}$ of finding a MUS of size $MaxSize$ and then run the loop 20 times. We leave tuning the constants of this algorithm to future work.

This algorithm is much more likely to remove large MUSes than small MUSes. If we are for example looking for a specific MUS $M$ of size 5, then if we remove $\frac{1}{4}$ of the elements in $X$ there is a $23.7\%$ chance that $M$ will not be removed. Alternatively, there is only a $0.32\%$ chance a MUS of size 15 will not be removed. This suggests we should find smaller MUSes more often (when they exist) and informally we observe this.

**Algorithm 2** ManyChop Algorithm

```
1:  procedure ManyChop(P, X, MaxSize)
2:      X = Shuffle(X)
3:      step = min({n ∈ ℕ|(1 − 1/2^n)^MaxSize ≥ 1/10})
4:      frac = 1 − 1/2^step
5:      for i ∈ [1..20] do
6:          check = Shuffle(X)[1..|X| ∗ frac]
7:          if Solve(check) == FALSE then
8:              return BasicMUS(check, MaxSize)
9:      return FAIL
```

The algorithms discussed so far take a limit for the size of MUS to find. We can combine them with Algorithm 3 to find a globally smallest MUS. This accepts a SAT problem $P$, list of Boolean activators for the constraints $X$, problem literals $L$ (which represent the values these variables can take in the solution) and the number of times $n$ to search for each size of MUS (for our experiments we set $n = 100$).

We search using iterative deepening, trying larger and larger sizes of MUS. In our experiments the loops on Line 7 are executed in parallel, distributing the calls to the MUS algorithm over all available CPUs. We wait until Line 13 to check if we have found a small enough MUS, rather than return as soon as a MUS of size $s$ is found, as we may find many MUSes of the same size.

**Algorithm 3** Finding a globally smallest MUS

```
1:  procedure FindGlobalMUS(P, X, L, n, musAlg)
2:      SmallMUSd = FindSize1MUS(P, X, L)
3:      if SmallMUSd ≠ FAIL then return SmallMUSd
4:      MUSd = dict()                  ▷ Init as an empty dictionary
5:      small = ∞
6:      for s in [1..|X|] do
7:          for r ∈ [1..n] and l ∈ L do
8:              core = musAlg(P + {¬l}, X, s)
9:              if core ≠ FAIL then
10:                 if (l ∉ MUSd) or
                        (|MUSd[l]| > |core|) then
11:                     MUSd[l] = core
12:                     small = min(small, |core|)
13:         if small ≤ s then return MUSDict
```

## 6 Experiments

We consider two different methodologies to show that MUS generation via `FindGlobalMUS` in DEMYSTIFY[2] lines up with how players solve puzzles. Firstly, we compare against a selection of published tutorials. Secondly, we look at solving an entire puzzle where the player discusses their reasoning at each step. All experiments were run on a 6 Core 3.7GHz Intel i5-9600K with 16GB of RAM running Ubuntu 20.04 and Python 3.8.5.

### 6.1 Experimental Design

We wrote each of our puzzles in the high-level input language of SavileRow [Nightingale *et al.*, 2015]. We try to match the original English description of the puzzles. In this subsection we discuss some modelling challenges which arose.

In problems such as Sudoku and variants, it is common for players to remove possible values for a cell one at a time, until only one remains. This is commonly referred to as candidate elimination. In problems such as Skyscrapers, Kakuro or Futoshiki the game interfaces are commonly designed to let players only fill in a cell once they know its value. To support these two methods of playing, DEMYSTIFY can either generate MUSes for all candidates for all cells (allowing candidate elimination) or only MUSes for "positive" literals (only allowing cells to be assigned their final value).

In problems which do not allow candidate elimination we imposed *AllDifferent* constraints as a single constraint. For problems which allow candidate elimination we decomposed the *AllDifferent* constraints into smaller pieces, requiring that each pair of variables take different values and each value occurs exactly once. This is because without candidates the deductions possible from a single *AllDifferent* are quite simple, while with candidate elimination the tutorial will decompose **AllDifferent** constraints into smaller simpler pieces.

Several puzzles (including Tents and Trees, Thermometers and Starbattle) require that there is some fixed given number of objects in rows, columns or regions. We split these equality

[2]https://github.com/stacs-cp/demystify

constraints into $\geq$ and $\leq$ constraints, as often only one part was required, and these made the resulting MUSes easier to understand.

In the Tents and Trees puzzle, there must be a bijection between tents and trees. In every puzzle we looked at it is easy to see which tent is attached to which tree, but expressing this as a constraint is difficult. Instead, we assign each tree a unique number between 1 and $n$, then fill in cells with a number between 0 and $n$, where 0 represents empty and $i > 0$ represents that this is the tent for tree $i$. We then require each non-zero number in the grid occurs exactly once.

There are some puzzles we do not consider in this paper, as we found we were unable to represent them in our input language in such a way that the resulting MUSes lined up with tutorials. The two main problems we found were problems with implicit arithmetic reasoning (such as Killer Sudoku) and problems where the player is required to draw a single line or cycle which is connected. These both produce bad models for MUSes for the same reason – they are large constraints which are considered a single constraint by the player, but which allow very complicated deductions. In the future, we will investigate ways of splitting these constraints into smaller pieces such that the resulting pieces produce human-understandable MUSes.

## 6.2 Tutorials

To show that MUS generation lines up with how players solve puzzles, we compared our techniques to the tutorials for ten different puzzles, seeing in each case if the MUS highlighted the same constraints as those given by the tutorial.

For each step of each tutorial, we use `ManyChop` to get the smallest MUS for one of the deductions produced by that tutorial step. We do not use the globally smallest MUS, as in many cases there were smaller MUSes in different parts of the puzzle, unrelated to the logical rule the tutorial step was demonstrating.

In some cases, a MUS may only deduce one, or a subset, of the deductions described in a single tutorial step, as many tutorial steps describe a general idea and then apply it in many places. We define a successful match by the MUS when it correctly captures the reasoning for the single deduction we chose. Where tutorials show several connected steps we consider each step individually, rather than running DEMYSTIFY to solve the whole puzzle.

There were two common issues we found with tutorials. In some cases the tutorial example had multiple answers, DEMYSTIFY still works in this case, but will only deduce values which take the same values in all solutions. A small number of tutorial steps had no solutions, in this case, our algorithms do not work and we remove those instances.

We have taken instances from eight different online guides. For Sudoku, X-Sudoku and Jigsaw Sudoku we used [Stuart, 2008]. The other two major sources for instances of techniques, for various puzzles, are [Conceptis, 2002] and [Tectonic, 2005][3]. Some tutorials present named techniques with one or more example puzzles; in other cases, the explanations are spread over a step-by-step solving guide. Table 1 shows the total number of instances we extracted for each puzzle type, and how many times we matched the same required constraints as the tutorial.

| Puzzle | #techniques | matched | |
|---|---|---|---|
| | | # | % |
| Binairo | 13 | 13 | 100% |
| Futoshiki | 15 | 13 | 87% |
| Jigsaw Sudoku | 3 | 3 | 100% |
| Kakuro | 16 | 16 | 100% |
| Skyscrapers | 14 | 12 | 85% |
| Starbattle | 24 | 21 | 88% |
| Sudoku { Basic/Tough | 29 | 20 | 69% |
| Sudoku { Diabolical † | 29† | 12 | 41% |
| Tents and Trees | 9 | 9 | 100% |
| Thermometers | 7 | 6 | 86% |
| X-Sudoku | 3 | 3 | 100% |

Table 1: Summary of the number of instances in guides, and how many DEMYSTIFY matched. †We exclude 'Unique Rectangle' techniques, which make use of the requirement that Sudokus have a unique answer. As we use MUSes to check if a problem is unsolvable and not if it has a unique solution, our technique does not apply.

For Binairo, Jigsaw Sudoku, Kakuro, Skyscrapers, Tents and Trees and X-Sudoku we matched all tutorial steps (Table 1). On average for all puzzles, apart from classic Sudoku, we match 85%. In some cases where DEMYSTIFY produced a different MUS to the tutorial we believe it could be argued the MUS found by DEMYSTIFY was simpler, but we strictly compare to the reasoning presented rather than apply our judgement as to which reasoning was simpler.

Our results on the classic Sudoku puzzle are not as impressive as for the other puzzles. There are several reasons for this. One is that we often find constraints which represent a different Sudoku technique to the one in the tutorial. For example, instead of the "Naked Triples" or "Hidden Triple" techniques we find "Pointing Pairs": the latter is sometimes considered as an easier technique, e.g. by Sudoku Dragon's strategy guide [Senn, 2020]. A second reason is that Sudoku is exceptionally well-studied and many rules have been invented. Some of these 'Diabolical' [Stuart, 2008] techniques are required exceptionally rarely and many involve very large MUSes (up to 56 constraints), much larger than the MUSes in any of the other problems we looked at. We only accept these when we matched exactly and in many cases we found similar (and often smaller) but not identical reasoning. We separate the "Diabolical" techniques in Table 1, where we see significantly better performance on the 'Basic' and 'Tough' techniques.

Overall, we believe Table 1 gives strong evidence for the validity of using MUSes for solving unseen puzzles. With no significant tuning (other than deciding how to represent *AllDifferent* constraints) we have reproduced a significant number of the techniques from a varied set of puzzles.

## 6.3 Miracle Sudoku

One notable variant of Sudoku is the *Miracle Sudoku*, designed by Mitchell Lee. In the Miracle Sudoku the standard Sudoku rules apply, and cells separated by a king's

[3] A full list of tutorials is provided in the supplementary materials

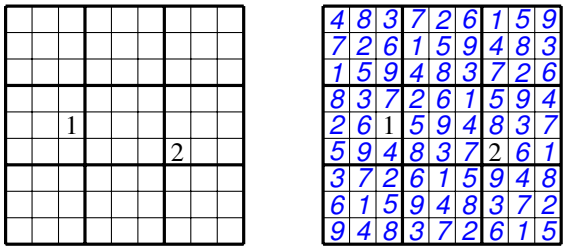


Figure 1: The Miracle Sudoku by Mitchell Lee.

move or knight's move in chess must have different values, and orthogonally adjacent cells cannot contain consecutive numbers. A video showing an expert solving this puzzle achieved over one million views in under three months [Anthony and Goodliffe, 2020] and resulted in mainstream press attention [Usborne, 2020]. The puzzle, and final solution, are shown in Fig. 1. To show our technique can solve entire puzzles, we compared DEMYSTIFY against the solution given in [Anthony and Goodliffe, 2020].

We generated a full solution for the Miracle Sudoku. The explanation contains steps which involved MUSes of size 1 and a smaller number of more complex steps. There were 49 steps where a non-unit MUS was used. These MUSes were size 3 or 4 and fell into one of the following categories.

- A classic Sudoku technique such as pointing pairs.
- A generalisation of this technique to Miracle Sudoku, for example where the only remaining possibilities for a 4 in a box are within a King's or Knight's move of another cell, that other cell cannot be 4.
- What Anthony called "dominoes": if we know one of two adjacent cells must be 4 then neither can be either 3 or 5, by the consecutive numbers rule.
- A similar but slightly more complex case of "triominoes". If we know that a 4 must occur in one of two non-adjacent cells, and both are adjacent to a third cell, the third cell cannot be 3 or 5.

Comparing these step-by-step with the YouTube video, we find that all the above techniques were used except the last. It is particularly striking that reasoning steps specialised to this variant, such as the use of dominoes, were discovered during solving both by Anthony and by our program. Anthony never used the triomino technique above: in some cases, he used slightly more complicated reasoning steps in terms of the number of cells involved but ones which did not necessitate the discovery of the triomino reasoning step. Both Anthony's and our explanation proceed similarly.

While this section only reports on a single type of Sudoku variant, it is promising that our techniques could find broadly similar explanations to a human expert.

### 6.4 Performance Comparison

While performance is not a primary concern in this paper, we performed one small experiment to compare the performance of our algorithms. This experiment demonstrates the need for an SAT solver which has the functionality of `FindUnsatCore`, and also shows the wider variety of MUSes that `ManyChop` finds.

| Technique | # | Time | Choices | MUSes |
|---|---|---|---|---|
| **Basic** | 6 | 92 | 80 | 125 |
| **-Limit** | 6 | 140 | 82 | 136 |
| **-Core** | 4 | - | 101 | 178 |
| **-Limit, -Core** | 3 | - | 70 | 129 |
| **ManyChop** | 6 | 49 | 124 | 312 |
| **-Core** | 6 | 13086 | 124 | 323 |

Table 2: Solving 6 LA Times Sudokus to completion. Solvers run with no FindUnsatCores (Core) and no MUS size limit (Limit) where appropriate. Times in CPU mins, solvers given 9600 CPU mins to solve each Sudoku. # - Total solved, Choices - total number of candidates with smallest MUS size, MUSes - total number of distinct MUSes found for all these candidates.

The algorithms solved six Sudokus from August 20th 2020 to August 30th 2020 from the LA Times which require at least one MUS of size greater than 1. To ensure the algorithms were forced to consider the same steps, we first solved each Sudoku with the `BasicMUS` technique, and then use the same sequence of choices for all the other algorithms.

Our results are presented in Table 2. We ran our algorithms with `FindUnsatCore` always returning its input when the problem is unsolvable instead of a subset (`-Core`), and with no limit on the size of the MUS to be found (`-Limit`). We measured, each time the algorithms had to find a MUS of size greater than 1, both the number of candidates with MUS of the smallest size found by any algorithm and the total number of distinct smallest size MUSes found for all candidates.

We observe the `-Core` variants are orders of magnitudes slower, so a good implementation of `FindUnsatCores` is vital for MUS finding. The `Basic` algorithm, while fast, produces the fewest different smallest MUSes. The `ManyChop` algorithm performs faster and also produces a much greater number of smallest MUSes.

## 7 Conclusion and Future Work

In this paper, we have presented a new algorithm for efficiently finding small MUSes. We demonstrate its usefulness and generality by producing descriptions of steps for many pen and paper puzzles. We also demonstrate that MUSes align very closely with pre-existing research on how human players decide how to solve these puzzles. This work, along with earlier work on Logic Grid Puzzles [Bogaerts *et al.*, 2020], provides strong evidence that MUSes are a powerful, natural, and generic method of explaining how to solve puzzles in a human-like way.

We believe the FORQES [Ignatiev *et al.*, 2015] approach is one that closely aligns to our needs. However, it works on problems where constraints are only represented as individual SAT clauses, while our puzzle models describe constraints as many SAT clauses. As part of future work we want to produce an extension of the FORQES approach for incrementally solving puzzles specified by high-level constraints.

For future work, we want to also explain exactly how the constraints in a MUS can be used to deduce the next step

of the puzzle. This needs a step beyond the current work to involve significant work in Human Computer Interaction as well as a possible collaboration with psychologists.

## References


[Anthony and Goodliffe, 2020] Simon Anthony and Mark Goodliffe. The miracle sudoku, 2020.

[Audemard and Simon, 2018] Gilles Audemard and Laurent Simon. On the glucose SAT solver. *Int. J. Artif. Intell. Tools*, 27(1):1–25, 2018.

[Audemard *et al.*, 2013] Gilles Audemard, Jean-Marie Lagniez, and Laurent Simon. Improving glucose for incremental SAT solving with assumptions: Application to MUS extraction. In *SAT*, 2013.

[Bacchus and Katsirelos, 2015] Fahiem Bacchus and George Katsirelos. Using minimal correction sets to more efficiently compute minimal unsatisfiable sets. In *CAV*, 2015.

[Belov and Marques-Silva, 2012] Anton Belov and João Marques-Silva. Muser2: An efficient MUS extractor. *J. Satisf. Boolean Model. Comput.*, 8(3/4):123–128, 2012.

[Bogaerts *et al.*, 2020] Bart Bogaerts, Emilio Gamba, Jens Claes, and Tias Guns. Step-wise explanations of constraint satisfaction problems. In *ECAI*, 2020.

[Chinneck and Dravnieks, 1991] John W. Chinneck and Erik W. Dravnieks. Locating minimal infeasible constraint sets in linear programs. *INFORMS J. Comput.*, 3(2):157–168, 1991.

[Cimatti *et al.*, 2011] Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. Computing small unsatisfiable cores in satisfiability modulo theories. *J. Artif. Intell. Res.*, 40:701–728, 2011.

[Conceptis, 2002] Conceptis. ConceptisPuzzles.com, 2002.

[de Siqueira N. and Puget, 1988] J. L. de Siqueira N. and Jean-Francois Puget. Explanation-based generalisation of failures. In *ECAI*, 1988.

[Dershowitz *et al.*, 2006] Nachum Dershowitz, Ziyad Hanna, and Alexander Nadel. A scalable algorithm for minimal unsatisfiable core extraction. In *SAT*, 2006.

[Gilpin *et al.*, 2018] Leilani H. Gilpin, David Bau, Ben Z. Yuan, Ayesha Bajwa, Michael Specter, and Lalana Kagal. Explaining explanations: An overview of interpretability of machine learning. In *DSAA*, 2018.

[Gupta, 2002] Anubhav Gupta. *Learning abstractions for model checking*. PhD thesis, Carnegie Mellon University, 2002.

[Hemery *et al.*, 2006] Fred Hemery, Christophe Lecoutre, Lakhdar Sais, and Frédéric Boussemart. Extracting mucs from constraint networks. In *ECAI*, 2006.

[Ignatiev *et al.*, 2015] Alexey Ignatiev, Alessandro Previti, Mark H. Liffiton, and João Marques-Silva. Smallest MUS Extraction with Minimal Hitting Set Dualization. In *CP*, 2015.

[Junker, 2001] Ulrich Junker. Quickxplain: Conflict detection for arbitrary constraint propagation algorithms. In *Workshop on Modelling and Solving problems with constraints (IJCAI)*, 2001.

[Lynce and Silva, 2004] Inês Lynce and João P. Marques Silva. On computing minimum unsatisfiable cores. In *SAT*, 2004.

[Marques-Silva *et al.*, 2013] João Marques-Silva, Mikolás Janota, and Anton Belov. Minimal sets over monotone predicates in boolean formulae. In *CAV*, 2013.

[Mazure *et al.*, 1998] Bertrand Mazure, Lakhdar Sais, and Éric Grégoire. Boosting complete techniques thanks to local search methods. *Ann. Math. Artif. Intell.*, 22(3-4):319–331, 1998.

[McMillan, 2003] Kenneth L. McMillan. Interpolation and sat-based model checking. In *CAV*, 2003.

[Nadel *et al.*, 2013] Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. Efficient MUS extraction with resolution. In *FMCAD*, 2013.

[Nightingale *et al.*, 2015] Peter Nightingale, Patrick Spracklen, and Ian Miguel. Automatically Improving SAT Encoding of Constraint Problems Through Common Subexpression Elimination in Savile Row. In *CP*, 2015.

[Papadimitriou and Wolfe, 1988] Christos H. Papadimitriou and David Wolfe. The complexity of facets resolved. *J. Comput. Syst. Sci.*, 37(1):2–13, 1988.

[Pelánek, 2011] Radek Pelánek. Difficulty Rating of Sudoku Puzzles by a Computational Model. *FLAIRS*, 2011.

[Senn, 2020] Mark Senn. *Sudoku Dragon - Strategy Guide*, 2020.

[Silva, 2010] João P. Marques Silva. Minimal Unsatisfiability: Models, Algorithms and Applications (Invited Paper). In *ISMVL*, 2010.

[Simonis, 2005] Helmut Simonis. Sudoku as a constraint problem. In *CP Workshop on modeling and reformulating Constraint Satisfaction Problems*, 2005.

[Stuart, 2008] Andrew Stuart. SudokuWiki.org, 2008.

[Tectonic, 2005] Tectonic. TectonicPuzzel.eu, 2005.

[Usborne, 2020] Simon Usborne. Puzzled man solving 'miracle' sudoku becomes youtube sensation. *The Guardian*, 2020.

[Wieringa and Heljanko, 2013] Siert Wieringa and Keijo Heljanko. Asynchronous multi-core incremental SAT solving. In *TACAS*, 2013.

[Wilson, 2006] Robin J. Wilson. *How to solve sudoku : a step-by-step guide*. Sterling Pub, 2006.

[Zhang *et al.*, 2006] Jianmin Zhang, Sikun Li, and ShengYu Shen. Extracting minimum unsatisfiable cores with a greedy genetic algorithm. In *AI*, 2006.