Towards Generic Explanations for Pen and Paper Puzzles with MUSes*

Joan Espasa[0000−0002−9021−3047], Ian P. Gent[0000−0002−5604−7006], Ruth Hoffmann[0000−0002−1011−5894], Christopher Jefferson[0000−0003−2979−5989], Matthew J. McIlree, and Alice M. Lynch[0000−0001−8393−8333]

University of St Andrews
{jea20,ian.gent,rh347,caj21,mjm42,al254}@st-andrews.ac.uk

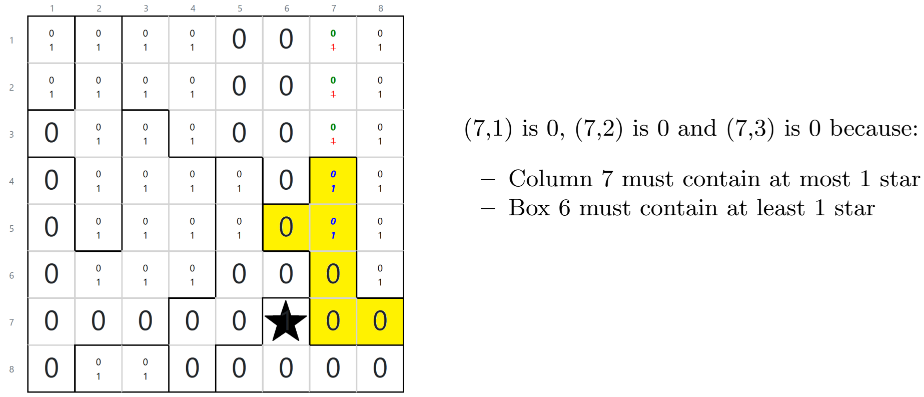
Abstract. Pen and paper puzzles like Sudoku, Futoshiki and Star Battle are hugely popular. Solving such puzzles can be a trivial task for modern AI systems. However, most AI systems solve problems using a form of backtracking, while people try to avoid backtracking as much as possible. This means that 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 Star Battle 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 [14, 13]. Such guides provide a reference to compare our techniques against.

Most paper and pen puzzles can be trivially solved when using a constraint solver [12]. This is due to propagators which enforce consistency between subsets of the variables or constraints in the problem. Propagators 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.

* This research was supported by the Royal Society URF\R\180015 .



	1	2	3	4	5	6	7	8
1	0 1	0 1	0 1	0 1	0	0	0 ~~1~~	0 1
2	0 1	0 1	0 1	0 1	0	0	0 ~~1~~	0 1
3	0	0 1	0 1	0 1	0	0	0 ~~1~~	0 1
4	0	0 1	0 1	0 1	0 1	0	0 1	0 1
5	0	0 1	0 1	0 1	0 1	0	0 1	0 1
6	0	0 1	0 1	0 1	0	0	0	0 1
7	0	0	0	0	0	★	0	0
8	0	0 1	0 1	0	0	0	0	0

(7,1) is 0, (7,2) is 0 and (7,3) is 0 because:

- Column 7 must contain at most 1 star
- Box 6 must contain at least 1 star

Fig. 1. Example of an explanation produced by Demystify for StarBattle

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. A common approach to explain how puzzles are solved 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. SudokuWiki [13] provides solvers for several Sudoku variants, showing which techniques can be applied at each stage of solving. Some works [9] 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. The major limitation of these systems is the need for an existing list of techniques. This paper provides a more general technique, based on Minimal Unsatisfiable Subsets (MUSes), which we demonstrate on a variety of puzzles. An example of our system's output is given in Figure 1.

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.

2 Background

The puzzles we discuss in this paper are typically solved by keeping a list of the values which are being considered for each cell of a grid, called the *candidates*. Once every cell has only one candidate remaining, the puzzle is solved. We consider puzzles with a single solution, which and are intended to be solved by humans without guessing. We call these *pen and paper puzzles*. We consider *Bi-*

```
given grid: int
letting griddim be int(1..grid)
given starcount: int

$#VAR stars
find stars: matrix indexed by [griddim, griddim] of bool

$#CON rowup "at least {p[’starcount’]} star(s) in row ({a[0]})"
$#CON rowdown "at most {p[’starcount’]} star(s) in row ({a[0]})"
find rowup: matrix indexed by [griddim] of bool
find rowdown: matrix indexed by [griddim] of bool

forAll i: griddim. rowup[i] -> (sum(stars[i,..]) >= starcount),
forAll i: griddim. rowdown[i] -> (sum(stars[i,..]) <= starcount)
```

Fig. 2. Fragment of the Demystify model for Star Battle, showing the constraints that each row must contain `starcount` stars

nairo, *Futoshiki*, *Kakuro*, *Starbattle*, *Tents and Trees*, *Thermometer*, *Skyscrapers* and *Sudoku* (rules and examples of all these puzzles can be seen on [14]).

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. 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 or model checking; see [11] for an extensive a survey. Identifying *minimum* unsatisfiable sets is a $\sum_2$-complete problem [5], there are some attempts including FORQES [7] at addressing this. On the other hand finding *Minimal Unsatisfiable Sets* (*MUS*) is easier [4]. MUSes cannot be shrunk by removing members, but may not be minimum (there may exist smaller MUSes). We concentrate on these in this paper, as they can be found in reasonable time.

3 Model Augmentation

The rules for many well-known pen and paper puzzles can be expressed using constraints. For example a Sudoku is built from *AllDifferent* constraints, Kakuro rules have sums and Futoshiki has inequalities.

To be able to give explanations for each reasoning step, all constraints used to model the rules of puzzles are half-reified. For each constraint c, SavileRow outputs $x \rightarrow c$, where c is the constraint and x is a Boolean variable that controls if the constraint is active. Each constraint is also associated with a string, describing in natural language terms what the constraint is expressing. We use SavileRow [8] to automatically translate high-level models of puzzles into SAT, with annotations to mark which variables represent the *Problem* (that the player completes) and which activates the *constraints*. Part of the specification for the puzzle Star Battle is given in Figure 2. The `$#VAR` annotation marks the variables the use must complete, and the `$#CON` annotation gives variables representing constraints. The language used to specify the English description of each constraint is contained in the Demystify documentation.

In problems such as Sudoku it is common for players to remove possible values for a cell one at a time until only one remains, commonly referred to as *candidate elimination*. In other puzzles such as Skyscrapers, Kakuro or Futoshiki it is common to only fill in a cell once the player knows its value. To support these two methods of playing, Demystify can either generate MUSes for both positive and negative assignments to *Problem* variables (allowing candidate elimination) or only for positive assignments to *Problem* variables.

4 MUSes for Explaining Puzzles

Demystify[1] generates explanations very similarly to [2], 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 (using SavileRow [8]). This translation produces a set L of variables L representing each value which can be assigned to each *problem* variable and a set X of variables which activate the half-reified constraints of the puzzle.
2. $\forall l \in L$ take the value a of l in the solution, find MUSes for $P \wedge (l \neq a)$.
3. Pick $l \in L$ which has the "best" MUS and display this to the user. Our criteria for picking the best MUS is: 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.

There are several improvements we make when presenting MUSes to the user, which reduce information overload and allow us to solve the puzzles in fewer steps. Firstly, MUSes of size 1 are grouped together, as there can be many such MUSes and they are very simple to understand. Secondly, for each MUS we find *all* literals which can be deduced using the MUS. 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. This lets us deduce several literals in a single step. MUSes are displayed to the user by listing the English descriptions of the constraints.

5 MUS Algorithms

As previously discussed, there are many existing MUS finding algorithms. We found existing state-of-the-art techniques for finding smallest MUSes either did

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

Algorithm 1 Basic MUS finding algorithm

```
procedure BasicMUS($P, X, MaxSize$)
    $X$ = FindUnsatCore(**Shuffle**($X$));   ToConsider = **ShuffledCopy**($X$)
    MusSize = 0                                          ▷ Values known to be in MUS
    for $c \in$ ToConsider do
        if $c \in X$ then
            core = **FindUnsatCore**($P$, $X - c$)
            if core == Fail then
                MusSize += 1                                     ▷ $c$ must be in the core
                if MusSize == $MaxSize$ then
                    $X = X[1..MaxSize]$
                    if FindUnsatCore($P$,$X$) == Fail then return Fail
                    else return $X$
            else $X$ = core
    return $X$
```

Algorithm 2 ManyChop Algorithm

```
procedure ManyChop($P, X, MaxSize$)
    $step = \min(\{n \in \mathbb{N} | (1 - \frac{1}{2^n})^{MaxSize} \geq \frac{1}{10}\})$;   frac $= 1 - \frac{1}{2^{step}}$
    for $i \in [1..20]$ do
        check = **Shuffle**($X$)$[1..|X| * \text{frac}]$
        if **Solve**(check) == False then return **BasicMUS**(check, $MaxSize$)
    return Fail
```

not finish in reasonable time or could only find MUSes when 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* literal, where often most of these problems will have no small MUSes.

Demystify uses Glucose [1] as the underlying SAT engine via the PySAT library [6]. Our algorithms use the `FindUnsatCore` function of Glucose. This function takes a SAT problem and list of variables X. It returns Fail if there is a solution where all members of X are True, or a subset of X such that P is unsolvable if all members of X are assigned true.

`BasicMUS`, a variant of the deletion-based algorithm of [4], is given in Algorithm 1. `BasicMUS` accepts a problem P and a set of variables X (representing the constraints) and tries removing each element of X in turn, checking the result is still unsolvable. It uses `FindUnsatCore` to reduce X at each step. The only new feature is stopping once $MaxSize$ members have been found and checking if they form a MUS, if not we need more values and return Fail.

One major limitation of `BasicMUS` is the lack of variety in the MUSes it returns, as `FindUnsatCore` often returns the same unsat core. We mitigate this with the `ManyChop` algorithm (Algorithm 2), which starts by removing a random subset of X. `ManyChop` chooses a fixed-size proportion of X to remove and keeps trying to remove that many elements of X and checking if the problem is still unsolvable, before using `BasicMUS` to find a MUS. The intuition behind `ManyChop` is that, given a set X, if we remove some proportion p of the elements of X, the chance that any fixed collection of n elements remains behind is approximately $(1-p)^n$. In our experiments, we choose p such that there is at least a probability of $\frac{1}{10}$ of finding a MUS of size $MaxSize$, then search 20 times.

We finally find the globally smallest MUS by searching over the problem variables (which represent the values these variables can take in the solution) in parallel using iterative deepening looking for larger and larger sizes of MUS.

6 Experiments

We compare Demystify against a selection of published tutorials to show how it lines up with human players. We wrote each of our puzzles in Essence' [8]. Below we discuss some modelling challenges which arose during this process.

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 a fixed number of objects in rows, columns or regions. We split these equality constraints into $\geq$ and $\leq$ constraints, as this made the resulting MUSes easier to understand. In the Tents and Trees puzzle, there is a bijection between tents and trees. To express this bijection 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 require each non-zero number in the grid occurs exactly once.

6.1 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

Puzzle		#techs	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%
	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 MUSes cannot make this deduction.

Demystify to solve the whole puzzle. There were two common issues we found with tutorials. In some cases the tutorial example had multiple answers, in this case Demystify can only deduce variables which take the same value in all solutions. Some tutorials had no solutions – we remove those instances.

We have taken instances from different online guides. For Sudoku, X-Sudoku and Jigsaw Sudoku we used [13]. The other two major sources for instances of techniques, for various puzzles, are [3] and [14]. 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 tutorials. 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 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 [10]. A second reason is that Sudoku is exceptionally well-studied and many rules have been invented. Some of these 'Diabolical' [13] techniques are required exceptionally rarely and many involve very large MUSes (up to 56 constraints), much larger than 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.

7 Conclusion and Future Work

We have presented a new algorithm to efficiently find 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 [2], 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 [7] 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

1. Audemard, G., Simon, L.: On the glucose SAT solver. IJAIT **27**(1), 1–25 (2018)
2. Bogaerts, B., Gamba, E., Claes, J., Guns, T.: Step-wise explanations of constraint satisfaction problems. In: ECAI (2020)
3. Conceptis: ConceptisPuzzles.com (2002), http://www.conceptispuzzles.com
4. Dershowitz, N., Hanna, Z., Nadel, A.: A scalable algorithm for minimal unsatisfiable core extraction. In: SAT (2006)
5. Gupta, A.: Learning abstractions for model checking. Ph.D. thesis, Carnegie Mellon University (2002)
6. Ignatiev, A., Morgado, A., Marques-Silva, J.: PySAT: A Python toolkit for prototyping with SAT oracles. In: SAT (2018)
7. Ignatiev, A., Previti, A., Liffiton, M.H., Marques-Silva, J.: Smallest MUS Extraction with Minimal Hitting Set Dualization. In: CP (2015)
8. Nightingale, P., Spracklen, P., Miguel, I.: Automatically Improving SAT Encoding of Constraint Problems Through Common Subexpression Elimination in Savile Row. In: CP (2015)
9. Pelánek, R.: Difficulty Rating of Sudoku Puzzles by a Computational Model. FLAIRS (2011)
10. Senn, M.: Sudoku Dragon - Strategy Guide (2020), https://www.sudokudragon.com/sudokustrategy.htm
11. Silva, J.P.M.: Minimal Unsatisfiability: Models, Algorithms and Applications. In: ISMVL (2010)
12. Simonis, H.: Sudoku as a constraint problem. In: CP Workshop on modeling and reformulating Constraint Satisfaction Problems (2005)
13. Stuart, A.: SudokuWiki.org (2008), http://www.sudokuwiki.org/
14. Tectonic: TectonicPuzzel.eu (2005), http://www.tectonicpuzzel.eu