Solving sudokus with a basic SAT solver

Maurice Frank
11650656
Universiteit van Amsterdam
maurice.frank@posteo.de

Melika Ayoughi
12224855
Universiteit van Amsterdam
melikaayoughi@gmail.com

March 3, 2019

1 Introduction

The *Boolean Satisfiability* or *SAT* problem is to find an assignment for all literals, where all clauses written down in conjunctive normal form evaluate to true. SAT is an NP-complete problem, thus any NP problem can be reduced to a SAT problem. Therefore, solving a SAT problem is an interesting topic. In this paper we formulate sudokus as problems in propositional logic and solve them using a SAT solver. We implement four different decision heuristics that we test on a data set of sudokus from six different difficulty levels and two different rules. We propose the following investigative questions:

- Which heuristic performs better?
- Are sudokus that are harder for humans, also computationally harder?
- How does applying more rules to sudoku affect the difficulty of finding a solution?

2 Method

We formulate the rules of sudoku in prepositional logic, precisely denoting them in their conjunctive normal form. The three constraints are: There exists at least one number in each square. There exists at most one number at each square. No number appears twice in a region (row, column or block).

Additionally we also encode additional cross rules for sudoku, that implies no number appears twice in the two diagonals (see Figure 1). The corresponding rule clauses are added in the same manner as in the third condition of normal sudokus.

2.1 DPLL

To find a solution to the Boolean satisfiability problem we implement the proven [1] Davis–Putnam–Logemann–Loveland (DPLL) [2] algorithm. The algorithm needs a decision heuristic on which literal to set to which value if direct simplification is not possible anymore. We implement four heuristics.

2.2 Next

As the most simple heuristic, Next picks the first unassigned literal in the first clause of the set of clauses.

2.3 DLIS

In the Dynamic Largest Individual Sum (DLIS) heuristic [5], the literal and polarity that satisfy the maximum number of unsatisfied clauses is chosen at each split. By making the directly most effective split possible its hypothesized that the solution is reached faster. We only take positive literals into accounts

2.4 VSIDS

The Variable State Independent Decaying Sum (VSIDS) heuristic [6] keeps a score for each signed literal. As we reach a contradiction each literal of this clause gets it score increased by b. For a split, the not yet assigned variable with the highest score is chosen. The increment grows exponentially $b^{t+1} = c \cdot b^t$. We set $c = 1$, cancelling the decaying effect. Higher scores are given to literals occurring in conflict clauses. Thus priority is given to literals that are more prone to cause a contradiction.

	Backtracks	Assigns	Splits
Backtracks			
Assignments	0.998		
Splits	0.999	0.995	
Solves	0.998	1.000	0.995

Table 1: The Pearson correlation coefficients for the different log points. As all of them are highly linearly correlated it is equivalent to use any of them as measurements of computational complexity.

2.5 Jeroslow-Wang

The Jeroslow-Wang heuristic [3] computes a score function $J(l)$ (Eq. 1) for each literal l, by looping over all clauses c which contain l. The term $|c|$ is the number of unassigned literals in the clause c. The heuristic gives higher priority to literals with high occurrences in shorter clauses. Again we only take positive literals into account.

$$J(l) = \sum_{c \in C \,:\, l \in c} 2^{-|c|} \tag{1}$$

2.6 Design Decisions

We implement the DPLL recursively, mirroring the decision tree in the call tree. On each assignment of a literal we copy the set of clauses as then we can remove false literals from clauses and satisfied clauses from the set. An empty clause represents a contradiction and an empty set of clauses implies a solution. We utilize the optimizations inherent in CPython to achieve a high-performance implementation. A clause is a dictionary with the signed literals as keys. As dictionaries in CPython are implemented as open hash-tables using only a primitive polynomial hash function checking for membership or value access is extremely fast. In the DPLL we only need to check if a signed literal is contained in a clause and remove literals. The copy of the clause set is cheap as CPython does not memory copy data if not necessary.

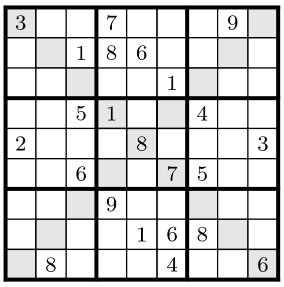


Figure 1: A sample sudoku showing the cross rules

3 Experiments

In our experiments, we measure the number of backtracks as the metric to compare different heuristics. The different runtime metrics are highly correlated, see Figure 1, thus they are all interchangeable. The number of backtracks is an intuitive metric for the performance of a recursive algorithm. For each heuristic, we run the experiment across the different difficulty levels and game rules (with/without cross rules). We note the number of clues for each sudoku. Additionally, we measure the size of the solution to illustrate the progress in the solution in time. We run the experiments on 24 Intel Xeon with 2Ghz and 90 GB of memory.

3.1 Data set

To generate a controlled and stable dataset we employ Simon Tatham's Puzzle Collection [7]. The collection includes a sudoku game (*there:* Solo) with six levels of difficulty (*block, simple, intersect, set, extreme, recursive*), arbitrary board size and different game rules. We rewrite the game code to use it as a sudoku generator. Besides the normal rules of sudoku we further use the advanced cross-rules in which digits are only allowed to appear in each diagonal once (see Figure 1). For each difficulty level and the two rule sets we generate 300 sudokus making in total a collection of 3600 unique games.

3.1.1 Difficulty levels

The generator from [7] knows six levels of difficulty based on six modes of (sudoku-domain specific) reasoning. A sudoku with difficulty d can be solved by applying rules $[1,\dots,d]$ but not with only the rules $[1,\dots,d-1]$. The modes of reasoning in order of increasing difficulty are the following [7]:

1. **Positional elimination** (*block*): A number must go in a square because all the other empty squares in a given region (row/column/block) are ruled out.

2. **Numeric elimination** (*simple*): A square must have a number in because all the other numbers that could go in it are ruled out.

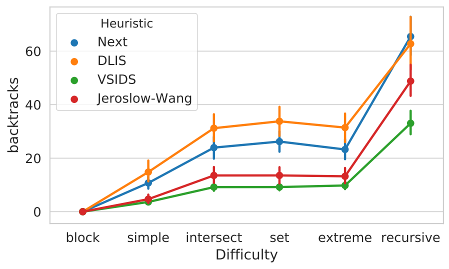
3. **Intersectional analysis** (*intersect*): Given two overlapping regions, if the possible locations for a particular number in one of the regions can be narrowed down to the overlap, then that number must be ruled out everywhere but the overlap in the other region.

4. **Set elimination** (*set*): If there is a set of empty squares within a region with only as many possible numbers as the size of the set, then those numbers can be ruled out everywhere else in the region.

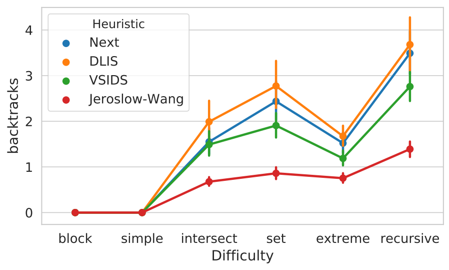
5. **Forcing chains** (*extreme*): A forcing chain is a path of pairwise-exclusive squares (i.e. each pair of adjacent squares in the path are in the same row, column or block) where: (a) Each square on the path has precisely two possible numbers. (b) Each adjacent pair share at least one possible number. (c) Each square in the middle of the path shares its numbers with at least one of its neighbours. These rules imply that knowing one number in one end of the path, forces all the rest of the numbers along the path. If we know that the two end squares are both in line with some third square and the third square currently has number x as a possibility, we can deduce that at least one of the two ends of the forcing chain has number x; therefore, the mutually adjacent third square does not.

6. **Recursion** (*recursive*): Build a search tree and try guessing values for a particular square recursively.

4 Results and Analysis



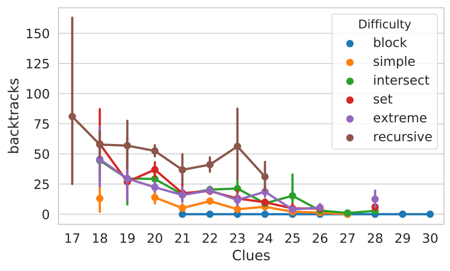
(a) With cross rules



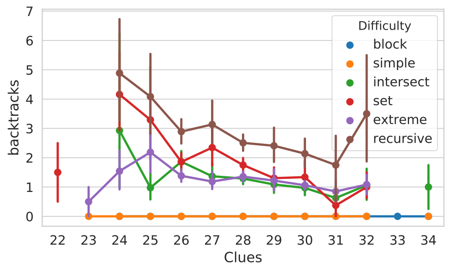
(b) Without cross rules

Figure 2: Mean number of needed backtracks at different difficulty levels and for the two different set of rules

Results of the mean number of backtracks for the different rules and heuristics are given in Figure 2. DLIS requires the highest amount of backtracks, over all difficulty levels, followed by Next. VSIDS outperforms Jeroslow-Wang on the cross-sudokus while for the non-cross the reverse holds. Other works [4] suggested VSIDS to considerably outperform Jeroslow-Wang. We hypothesize this difference in the fact that we use a simplified version of VSIDS as well as sudokus being comparably simple problems. The strength of the original VSIDS lays in its multiplicative decay behaviour in Conflict Driven Clause Learning (CDCL). As we do not use such a solver and sudokus seldomly present a deep decision tree, its use here is not comparable. The simple Next heuristic outperforming DLIS can also be explained by the specifics of this sudoku experiment. In Next we pick the next literal found in our clauses. Because of the ordering of our clauses



(a) With cross rules



(b) Without cross rules

Figure 3: Mean number of needed backtracks for sudokus with different number of clues given. Note the extremely high variance at the border cases.

we will pick squares sweeping from top-left to bottom-right and split on their corresponding literals. This being a good heuristic in computationally solving a sudoku performs well on our data but will not generalize outside of this domain.

We assumed that the levels of difficulty provided in the puzzle collection roughly represent human conception of difficulty. Thus, our experiments can indicate if human perceived complexity relates to computational complexity. Figure 2 shows that the required number of backtracks generally increases with higher difficulty level with *extreme* posing an exception. The forcing chains of the *extreme* level are rather difficult for humans to use but not so for our solver. As the chains are just combinations of logical rules which is already the operation of the SAT solver. Assigning literals in a forcing chains will have a high pruning factor; therefore, we hypothesize that having chains makes it easier for the SAT solver. The *recursive* level is the hardest as in the generation it was ensured that it is not possible to solve this sudoku without guessing values at some point. In return, this calls for splitting during the solving.

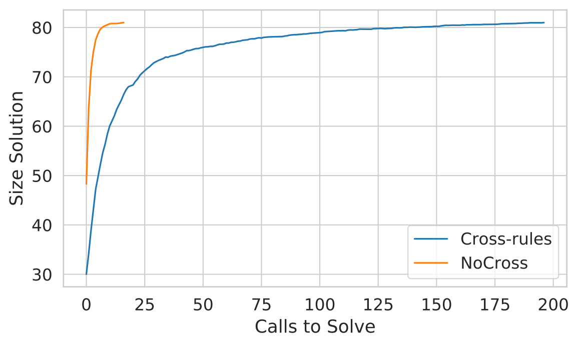


Figure 4: Although solving with cross rule takes roughly 10 times longer, the solver finds most of the solution in a short time.

We hypothesized that additional cross rules would make the sudoku easier to solve, as it is for humans. The results suggest that cross rules increase the number of backtracks significantly. Mainly because cross rules are all constraining rules that forbid existence of the same number in other squares. As a result, by forcing more constraints on the values, we face dead ends more and need to backtrack to the original guesses more.

Using our result data we also analyze the relationship between computational complexity and the number of given clues (see Figure 3. We see a general trend to more number of backtracks with less clues. The high variability of the results and the small size of the datasets do not allow for strong generalization though.

Lastly we analyze the size of the solution over the runtime of the solver as shown in Figure 4. The solver finds 85% of the solution in the first 10% of its runtime.

5 Conclusion

In this paper we analyzed the performance of different heuristics for the DPLL algorithm in solving sudokus. We showed that heuristics that fit the domain of sudokus well are performing better than in generalized SAT solver experiments. Further we investigated if human perceived difficulty of sudokus would translate into corresponding change of computational complexity in solving them. Using the difficulty rules from a sudoku generator we found such a relationship. In contrast and more notable we showed that adding the rules of the cross sudoku significantly decreases the performance of all solvers even though these rules tend to make the sudoku easier to solve for humans.

References

[1] Tomáš Balyo, Marijn J. H. Heule, and Matti Järvisalo. Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions.

[2] Martin Davis, George Logemann, and Donald Loveland. A Machine Program for Theorem-proving. 5(7):394–397.

[3] Robert G. Jeroslow and Jinchang Wang. Solving Propositional Satisfiability Problems. 1(1-4):167–187.

[4] Jia Hui Liang, Vijay Ganesh, Ed Zulkoski, Atulan Zaman, and Krzysztof Czarnecki. Understanding VSIDS Branching Heuristics in Conflict-Driven Clause-Learning SAT Solvers.

[5] Joao P. Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. 48(5):506–521.

[6] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Annual Design Automation Conference*, pages 530–535. ACM.

[7] Simon Tatham. Simon Tatham's Portable Puzzle Collection.

A Supplementary figures

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

Figure 5: The sample sudoku *solved*. This was difficulty advanced.