The importance of heuristics in SAT-solver design: exploring the differences between problem- and non-problem specific SAT-solving heuristics in a SUDOKU solving task [⋆]

No Author Given

Faculty of Science, Vrije Universiteit, Amsterdam, The Netherlands

Abstract. SAT solvers use Boolean logic to solve problems. These SAT solvers have a very wide application, and every year the performance of SAT solvers increases with the development of sophisticated algorithms. For example, the use of domain knowledge may increase the performance of SAT solvers in specific problems. In this paper, a problem specific Sudoku heuristic in combination with the DPLL algorithm is examined and compared to general splitting algorithms. The results suggest that the problem specific heuristic indeed improves the algorithm in number of backtracks for solving Sudoku's. However, this is not in proportion to the increased runtime of this heuristic. Therefore, it is concluded that the proposed problem specific heuristic does not improve the SAT solvers performance in solving Sudoku's.

Keywords: SAT-Solver · Sudoku · DPLL · DLCS · DLIS · JWOS · JWTS.

1 Introduction

1.1 General background

Satisfiability (SAT) solvers use Boolean propositional logic, statements that are True of False, to solve problems. When a correct solution is found, the SAT solver outputs the parameters that satisfy the problem. If no solution can be found, it reports that the problem is unsatisfiable [4]. The logic behind these SAT Solvers is simplistic, yet very powerful. The "Perseverance" Mars Rover, which is the autonomous physical agent that mankind put on planet Mars, is an impressive application of this technique [6]. Since direct communication with this agent is impossible due to a delayed communication of more than ten minutes, this agent needs to make decisions autonomously without interference of a human being.

Technological development enabled SAT solvers to be fast enough to be used in many different applications [4]. A conceptually easy way to solve SAT problems is the use of truth tables, which explore all possible combinations of true

[⋆] Assignment 1; Knowledge Representation '21, XM_0059 and false [5] [3]. However, this comes with the problem of exponentiation. Every added variable will make the number of combinations grow exponentially. To overcome this problem, algorithms are designed that significantly reduce the computational cost of finding the truth assignments that satisfy the problem. Research aimed at improving these algorithms is very valuable to improve the performance in SAT solving applications.

1.2 Problem specific vs. problem non-specific heuristics

SAT solvers are build to solve a generic set of literals and clauses. To a generic SAT solver, the meaning of the literals and clauses don't matter. However, within specific tasks, specific information is accessible which could help in solving these tasks more efficiently. Implementing this information within heuristics could theoretically improve the performance of the algorithm for the task at hand. However, it may decrease the performance for other problems. Heuristics using domain-specific information are from now on referred to as problem specific heuristics. Heuristics that are general and problem non-specific are referred to as problem non-specific heuristics, general or generic heuristics.

1.3 Sudoku problem

One application of SAT solvers that gained a lot of interest is the task of solving Sudokus. Sudoku is a game where the goal is to fill an $n \times n$ grid, most commonly 9×9, where a set of specific rules need to be satisfied. For a 9×9 Sudoku, every block of 3x3 need to contain every number from 1 to 9. Furthermore, every row and column need to satisfy this constraint as well. Lastly, no 3x3 block, column or row can contain the same number twice. Depending on the difficulty of the Sudoku, for n cells the correct number is already filled in. These numbers are referred to as the 'givens'.

In terms of SAT solving, depending on the representation, Sudoku puzzles often contain a lot of variables. Many of these variables are known as a result of the truth value of another variable. For example, if row one contains a one, that means that all the other cells in the first row cannot be a one. As a result, assigning a value to a cell automatically makes many other variables false due to the Sudoku rules.

1.4 Research aim

The aim of this research is to explore the significance of using problem specific heuristics instead of general heuristics when tackling a specific problem, in this case, solving Sudokus. Specifically, the research focuses on the performance of different splitting heuristics that were added to the standard DPLL algorithm (see section 2.1). For these experiments, a new splitting heuristic is designed that takes into account Sudoku-specific domain knowledge and attempts to use this to find a smart variable to split on. In this context, a smart variable is defined as 'the cell with the least possible numbers that can be assigned at that given moment'. This heuristic will be compared to several well known, generic splitting heuristics. Descriptions of the DPLL algorithm and the different splitting heuristics that were used can be found in section 2. Therefore, the research question is as follows:

RQ: *How does adding a Sudoku-specific splitting heuristic influence the performance of the DPLL algorithm compared to adding well-known, generic splitting heuristics?*

Since the Sudoku-specific splitting heuristic takes information into account that the other heuristics do not have access to, it may be able to make smarter decisions based on this information and therefore require less backtracking. Specifically, if a cell has fewer numbers that could possibly be assigned to it at at that specific point in the process, a variable for that cell is more likely to be correct compared to a cell that could take more possible numbers. This means that the algorithm makes less mistakes which may also lead to finding the solution faster, since less backtracking is required. However, the heuristic requires the algorithm to keep track of more information for each cell and regularly has to update it, which likely leads to a higher runtime. It would be interesting to see whether the heuristic does indeed make smarter decisions and whether this weighs up to the additional computational cost. Taking this into account, the following hypotheses are formulated:

$\boldsymbol{H_0}$ **:** *Using a Sudoku-specific splitting heuristic increases the number of backtracks the DPLL algorithm needs, but it decreases the runtime of the algorithm compared to general splitting heuristics.*

$\boldsymbol{H_1}$ **:** *Using a Sudoku-specific splitting heuristic decreases the number of backtracks the DPLL algorithm needs, but it increases the runtime of the algorithm compared to general splitting heuristics.*

2 Method

In this section, the DPLL algorithm and heuristics used in the experiments are discussed. The basic DPLL algorithm is implemented as a baseline. Several splitting heuristics such as DLCS, DLIS and Jeroslow-Wang were added to this baseline as well as a sudoku-specific splitting heuristic, developed by the authors of this paper.

2.1 DPLL algorithm

DPLL(Davis-Putnam-Logemann-Loveland) is a complete and backtracking-based tree search algorithm designed to solve the satisfiable problem of propositional logic formulae in conjunctive normal form[2] and it's one of the most powerful SAT solvers. This algorithm uses a branching procedure that shrinks the search space by falsifying clauses[1] [4]. The DPLL algorithm simplifies the basic backtracking algorithm by using the unit rule and pure literal elimination.

- **Unit-rule**. When a clause only contains a single literal, this literal can be set to true due to the property of CNF formulas that each clause has to be true in order to find a solution. All clauses containing this literal are then removed, since these are now by definition true. Lastly, the negation of this literal is removed from all other clauses since it is false. This narrows the searching space, thus accelerating the speed of the algorithm.
- **Pure literal elimination**. If a propositional variable occurs with only one polarity in all clauses, it is called a pure literal. This literal can be set to true and all clauses that contain this literal can be omitted, as they will also evaluate to true. However, implementing this rule requires additional searching to find if a variable is a pure literal, which increases the runtime of the algorithm. In the representation for Sudokus used in this paper, pure literals do not occur often and many clauses only have two literals. For this reason, pure literal elimination has not been implemented in these experiments.

The DPLL algorithm can be summarized in the following steps:

1. If a satisfiable solution is found or the result is found to be unsatisfiable, the procedure stops.
2. Unit propagation: if one or more unit clauses are present, assign the literal to be true and simplify the clauses as described above.
3. Splitting and backtracking: once there are no unit clauses or pure literals left, one literal is selected to split on and the literal will be assigned either true or false, depending on the implementation. If this split leads to an empty clause there is an inconsistency in the Sudoku, so the algorithm returns to a previous state and assigns the alternative truth value to the literal. This last step is called backtracking.

2.2 General splitting heuristics

The DPLL algorithm implements random splitting and chronological backtracking if no other heuristics are present. Adding heuristics to the DPLL algorithm can either improve or reduce the performance of the algorithm, depending on the specific heuristic as well as the problem that the algorithm is used on. An example of a heuristic that can be implemented within the DPLL algorithm is the CDCL algorithm, a backtracking algorithm which improves the performance greatly.

This research focuses on the impact of adding different splitting heuristics to DPLL. Other areas are unchanged, so backtracking still happens in the standard chronological order. In total, six different algorithms are considered for this experiment. The first one is standard DPLL with no further heuristics, meaning

[1] Literals are statements that can be either true or false and clauses are sets of literals in a clausal sentence [5].

splitting is done randomly. One is DPLL plus our own Sudoku-specific splitting algorithm, which is discussed in the next session. Lastly, four generic splitting algorithms were added to the standard DPLL algorithm.

These include DLCS (Dynamic Largest Combined Sum) and DLIS (Dynamic Largest Individual Sum) which both count the occurrences of each literal that occurs in the clauses. DLCS picks the most frequent literal, including its negation, while DLIS picks the literal with most occurrences where negated literals are counted separately. Both methods aim to simplify the largest number of clauses.

Both versions of the Jeroslow-Wang heuristics were also implemented; JW_os (one sided) and JW_ts (two sided). The basic idea of the Jeroslow-Wang heuristic[1] is to select the splitting variable by calculating its weighting. For a given literal, compute:

$$J(I) = \sum 2^{-|w|} \tag{1}$$

which is the sum over two to the power of the length of the clauses. For the One sided Jeroslow-Wang (JW_OS) method, the literal with the highest value of J is selected. Two-sided Jeroslow-Wang(JW_TS) takes into account whether a literal occurs positively or negatively. If J for positive literal is larger, then this heuristic picks the positive phase. Otherwise, it picks the negative phase. In an intuitive sense, weights in smaller clauses are higher. If a literal with the maximum weight is chosen, this means that a literal from a small clause is selected which is most likely to be satisfiable and yield a unit clause.

2.3 Sudoku-based splitting heuristic

Lastly, a splitting heuristic was designed that considers domain-based knowledge of Sudokus to find effective cells on which to split. The base assumption is that a cell that has a smaller number of values that can correctly be entered in that cell, is more likely to have a corresponding literal that will not turn out to be incorrect. This would lead to less backtracking. This heuristic was designed with the DPLL algorithm in mind, and is meant to be added on top of this algorithm, replacing the randomized splitting.

The heuristic works as follows. First, a dictionary is used which holds for each cell the number of possible values that, at any point during the algorithm, could still correctly be entered in that cell. During initialization for a 9×9 Sudoku, this value is 0 for cells that contain a given and 9 for cells that do not have a given. The dictionary is updated each time a literal is set to true or false. If set to true, the value for the corresponding cell is set to 0. If set to false, the value of the cell is reduced by 1. Each time the dictionary is updated, the algorithm checks if there is a new 'best cell'. The best cell is a cell that currently holds the lowest value in the dictionary, that is, the cell that has the least possible values that could still correctly be tried in that cell. When a new 'best cell' is established, a random literal from the current clauses corresponding to that cell is selected. Next time the algorithm needs to split, it will do so on this literal.

2.4 Experimental Design

The different algorithms are tested on a test set of 1011 Sudoku's. These Sudokus are the standard size of 9×9 and vary in difficulty with no specific qualities. One thousand Sudoku's should be adequate to provide enough data to draw the first conclusions on the performance of the proposed new Sudoku-specific splitting heuristic.

To compare the different heuristics and answer the research question, two metrics were decided to be relevant: runtime and number of backtracks. The runtime is a good general measure for the performance and the number of backtracks will help determine if the splitting heuristics are making smart decisions. In theory, choosing smart literals to split on will lead to less mistakes, resulting in less backtracking. On the other hand, inefficient splitting leads to more mistakes which requires more backtracking to find the solution. The runtime depends on the amount of backtracks the algorithm has to do, but also on the computation required for the algorithm.

Some standard descriptive statistics are used to describe the resulting data for both the backtracks and runtime. This includes the mean, median, standard deviation, max and min. For the runtime, the range is also provided. For the backtracking this was left out since the range is equivalent to the max for each algorithm. One special case is also considered: when a Sudoku is solved with zero backtracks, it means the algorithm did not assign any incorrect truth values that later had to be changed. This is considered a valuable metric for this research, since the Sudoku-specific heuristic is hypothesized to make smarter splitting decisions which would also lead to a larger number of Sudoku's solved without backtracks.

3 Results

This section covers some basic statistics of the experiments that were run. Table 1 contains the average, median, max and min for the number of backtracks of each heuristics as well as the number of Sudokus that were solved without backtracking. In each column, the best value is underlined.

Table 1. Descriptive statistics for backtracks

Heuristic	Mean	Median	SD	Max	Min	# of 0s
Random	4.75	2	8.66	83	0	345
DLCS	113.34	33	238.13	2475	0	233
DLIS	116.20	34	252.19	3041	0	232
JW_os	63.32	11	181.83	3552	0	247
JW_ts	102.50	33	203.80	2177	0	235
Sudoku	3.09	1	6.8625	75	0	437

Table 2 contains the average, median, max, min and range for the runtime of each algorithm. All runtimes are shown in *seconds*. Again, in each column, the best value is underlined.

Table 2. Descriptive statistics for runtime

Heuristic	Mean	Median	SD	Max	Min	Range
Random	0.5396	0.5160	0.0816	1.4060	0.4840	0.9220
DLCS	1.7194	0.8120	2.7165	29.0160	0.4690	28.5470
DLIS	1.7307	0.8120	2.8461	36.7030	0.4690	36.2340
JW_OS	1.5876	0.6560	3.5103	69.8440	0.4690	69.3750
JW_TS	2.3522	1.0310	3.9299	37.9840	0.4840	37.5000
Sudoku	49.3057	49.2190	0.62311	56.8590	48.2340	8.6250

The normality of the data was assessed using a Kolmogoriv-Smirnov and Sharipo-Wilk test. Both tests reach statistical significance, $p < 0.01$, on all variables suggesting that the difference between the data and the normal distribution reaches statistical significance. Therefore normality is not assumed and non parametric tests are used, being the non parametric version of the dependent ANOVA, Friedman's Test. The Friedman's Test suggests that the six different heuristics are statistically significantly different in both runtime, $\chi^2(5) = 3263.83$, $p < 0.01$, and backtracks, $\chi^2(5) = 2569.63$, $p < 0.01$. Post-hoc tests using a Wilcoxon Signed Ranks Test with an α of 0.01 (Bonferonni correction, for multiple (5) testing, $\alpha = .05/5 = 0.01$), as presented in Table 3, demonstrated that the difference in Median sampled ranks, in terms of runtime, between the Sudoku (2.0) and the Random (2.2), $N = 1011$, $p < 0.01$, DLCS (4.4), $N = 1011$, $p < 0.01$, DLIS (4.5), $N = 1011$, $p < 0.01$, JW_OS (3.6), $N = 1011$, $p < 0.01$, and JW_TS (4.4), $N = 1011$, $p < 0.01$, heuristic is reaching statistical significance. As presented in Table 2 the direction of the difference is towards more runtime for the Sudoku heuristic compared to the other five heuristics. However, the Median sampled ranks, in terms of number of backtracks needed to find the solution of the Sudoku, suggest a contrary effect, see Table 1. Using the Sudoku (6.0) heuristic, less backtracks are needed to find a solution compared to using the random (1.8), $N = 1011$, $p < 0.01$, DLCS (3.2), $N = 1011$, $p < 0.01$, DLIS (3.1), $N = 1011$, $p < 0.01$, JW_OS (2.8), $N = 1011$, $p < 0.01$, or JW_TS (4.3), $N = 1011$, $p < 0.01$, heuristic.

Table 3. Wilcoxon Signed Ranks Test [2-tailed], adjusted $\alpha = .01$

	runtime	runtime	backtracks	backtracks
Comparison	Z	p-value	Z	p-value
Sudoku vs. Random	27.54	< .01	8.26	< .01
Sudoku vs. DLCS	27.54	< .01	24.26	< .01
Sudoku vs. DLIS	27.54	< .01	24.21	< .01
Sudoku vs. JW_OS	27.54	< .01	23.10	< .01
Sudoku vs. JW_TS	27.54	< .01	24.14	< .01

4 Discussion

The effect of the Sudoku-specific heuristic is positive in terms of number of backtracks, but also negative in terms of runtime. Therefore, it should be argued whether the positive effects outweigh the negative effects. Less backtracking is needed when using the Sudoku heuristic, which suggests that the heuristic makes smarter splitting decisions. However, since this only a single aspect of the algorithm it may give a distorted picture of the overall performance. The relative run time is a more comprehensive measure of the general computational costs of an algorithm. The results suggest that in term of runtime, the Sudoku algorithm performs much worse than the other heuristics. With a Median time of 49.12 s per run to find a solution, the Sudoku heuristic is slower by a factor 47, compared to the second slowest heuristic. Compared to the fastest heuristic this difference is even higher, by a factor 95. Although the Sudoku heuristic outperforms the other in terms of backtracking, the positive effect seem to be negligible in relation to the negative effect of the extra computation time that is needed.

A plausible explanation for the increased runtime of the algorithm might be the need to constantly update the amount of numbers that are still possible within each cell as well as keeping the current 'best cell' up to date. That means that for every literal that is assigned a truth value, many values need to be updated and considered. The other algorithms don't keep track of this many variables and are therefore much more simplistic and therefore faster. To the knowledge of the authors, this is the first study that compared problem specific heuristics with problem non-specific heuristics. Since this study showed some improvements in one of the aspects of performance, being the number of backtracks, but the overall results is negative, more research is needed. Research could focus on improving problem specific heuristics by using smart data structures that don't require as much computational cost.

5 Conclusion

This study aimed to explore the significance of using problem-specific heuristics instead of general heuristics when tackling a specific problem. This study used Sudokus as the problem at hand. It was hypothesized that using a Sudoku-specific splitting heuristic improves the performance of the DPLL algorithm by requiring less backtracking but also increases runtime, compared to general splitting heuristics. The results suggest that the null hypothesis, $\boldsymbol{H_0}$, should be rejected since the Sudoku heuristic performs better in terms of backtracks, yet it increases the duration to find a solution. Thereby evidence is provided for the hypothesis, $\boldsymbol{H_1}$. However, the negative effects, expressed in an increased runtime, far outweigh the positive effects. Therefore it is concluded that problem specific heuristics decrease the performance of SAT solver when solving Sudokus. Moreover, more research is needed to test more comprehensive problem specific heuristics.

References

1. Chen, J.: Phase selection heuristics for satisfiability solvers. arXiv preprint arXiv:1106.1372 (2011)
2. Davis, M., Putnam, H.: A computing procedure for quantification theory. Journal of the ACM (JACM) **7**(3), 201–215 (1960)
3. Gehrke, M., Walker, C.L., Walker, E.A.: Normal forms and truth tables for fuzzy logics. Fuzzy Sets and Systems **138**(1), 25–51 (2003)
4. Gomes, C.P., Kautz, H., Sabharwal, A., Selman, B., van Harmelen, F., Lifschitz, V., Porter, B.: Handbook of knowledge representation. Foundations of Artificial Intelligence **3**, 89–134 (2008)
5. Lifschitz, V., Morgenstern, L., Plaisted, D.: Knowledge representation and classical logic. Foundations of Artificial Intelligence **3**, 3–88 (2008)
6. Sheini, H.M., Peintner, B., Sakallah, K.A., Pollack, M.E.: On solving soft temporal constraints using sat techniques. In: International Conference on Principles and Practice of Constraint Programming. pp. 607–621. Springer (2005)