Solving Sudokus Containing Multiple Solutions using DPLL: An Experimental Approach

Akshaj Verma, Freek Cool, Alon Efrati

Vrije Universiteit Amsterdam

Abstract. This paper researches the efficiency of DPLL on solving Sudokus that contain multiple solutions. These Sudokus are compared to the DPLL that solved Sudokus with only a unique solution. This is done with basic DPLL, DPLL with DLCS, and DPLL with DLIS. We can conclude that this depends on the given cells in the Sudoku. However, DPLL with DLIS has shown the best performance.

Keywords: DPLL · Sat Solving · Sudokus.

1 Introduction

In the field of Artificial Intelligence (AI) it is common practice to divide problems into defined categories, based on their run time. Problems which are easy to solve, are generally solvable in polynomial run time [1]. This class of problems, is defined as class P. However, the most difficult problems are in class NP-complete. For these problems, there does not exist an algorithm which can solve the problem in polynomial time [2]. Specifically, no problem in the class NP-complete is faster than exponential running time [3].Problems like the Traveling Salesman Problem belong to this class of NP-complete problems. Algorithms that aim to find the most optimal solution, or just even a single solution, might run for large amounts of time.

One such problem, which is categorized in the NP-complete class is satisfiability in propositional logic [4]. For small amounts of literals, this problem can be solved relatively quickly by an algorithm which fills the truth table of the logic formula, and it checks if a certain set of truth values for literals result in a satisfied logical formula. However, when the amount of literals increases, such an algorithm runs in exponential time, which is undesirable. This is because a truth table grows larger by a factor n^2, with n being the amount of literals. This makes the satisfiability problem in propositional logic an NP-complete problem.

To solve the satisfiability problem, the Davis-Putnam-Loveland-Logemann (DPLL) algorithm has been proposed [5]. This algorithm systematically assigns truth values to literals, using heuristics which aim to make the search more efficient. In this paper, the basic DPLL has been implemented [6], together with two heuristics. These three versions of the DPLL algorithm, will in this study be used in the satisfiability problem for Sudoku. Sudoku is a very well-known 1 person game. Depending on the difficulty and skill, it can be very time and energy

consuming. However, there is a specific amount of given cells that determines if a Sudoku has one or multiple solutions (INSERT VALUE).

For this study, the DPLL algorithm will be used to solve Sudoku puzzles, which will be encoded as a satisfiability problem. The objective of this paper is to examine if having more available solutions leads to more or less searching through the DPLL tree. To answer this question the amount of backtracks (CORRECT?) between standard DPLL, DPLL with the dynamic largest combined sum (DLCS) heuristic [7], and DPLL with the dynamic largest individual sum (DLIS) [7] will be compared.

2 The DPLL algorithm

In this project, first the basic DPLL algorithm has been implemented. It is important to note that the DPLL algorithm gets an logical formula as input in Clause-Normal-Form (CNF) [8]. A logical formula is in CNF, when it is a conjunction of disjunctions. Specifically, it is a conjunction of clauses, which contain disjunctions. Then it is up to the DPLL algorithm, to find a set of assignments to literals which satisfy all clauses.

The basic DPLL algorithm has already a basic heuristic to assign values to literals in a certain order, which aims to speed up the search for a solution. First, the algorithm searches for a unit clause, which is a clause containing just one literal. To satisfy this clause, that literal must be assigned the value that makes this single clause true [9]. Because these literals cannot have a different truth value, as otherwise the whole formula would result in false, the search tree is essentially cut in half. This is therefore a useful heuristic to speed up the search of the DPLL algorithm, which is implemented in our basic version of DPLL. If there does not exist a unit clause, then a literal is picked at random and set to true for the version of basic DPLL used in this study. This method of choosing literals when no unit clause exists will be changed with the heuristics discussed later. If the DPLL arrives at a contradiction or an empty clause, which can never be true, then the algorithm backtracks to the state before and assigns the previous literal chosen as false.

To summarize, the basic DPLL algorithm implemented in this study makes use of the unit clause heuristic when assigning values in a certain order to literals. When a unit clause does not exist, a random literal will be chosen to be assigned true. When a backtracking action occurs to that literal, it will be assigned false. As noted, this way of choosing a literal randomly will be changed in the heuristics, described next in section 3.

3 Implemented heuristics

3.1 DLCS

The first heuristic implemented in this study, to replace random selection of literals, is the DLCS heuristic [7]. For this heuristic, the amount of occurrences

of a literal in either negated or non-negated form are counted. Formally, CP(v) is number of occurrences of literal v, and CN(v) is the number of occurrences of $\neg v$. Then for the DLCS heuristic, v is picked with the largest $CP(v) + CN(v)$. However, the literal which is chosen according to this rule, is not just assigned true first, and if there is backtracked to this literal the it gets false assigned. In this heuristic, it is also counted in which form literal v occurs most. This is either negated, or non-negated form. If the negated form occurs more, thus $CN(v) > CP(v)$, then literal v is made false. If the other way around is true, then literal v is made true. If it happens that there is backtracked to a literal, which got its truth value assigned according to this heuristic, then the opposite truth value is assigned to this literal.

To summarize, the DPLL adapted with this heuristic will first search for unit clauses, and assign truth values to these literals. If no unit clause exists, it will execute this heuristic, count the occurrence of literals, and assign the appropriate truth value to that literal according to the rule stated above.

3.2 DLIS

The second heuristic which is implemented in this study, is the DLIS heuristic. This heuristic is in some aspects relatively the same as the DLCS algorithm, in that it will also count the number of occurrences of a literal. In contrast to the DLCS heuristic, DLIS considers v and $\neg v$ as two different elements which we count separately [7]. We thus pick a literal v with the largest $CP(v)$ or $CN(v)$. Then trivially, the literal chosen gets the value assigned according to the form in which it occurred the most. When backtracked to this literal, like the DLCS heuristic, this literal gets the opposite truth value assigned.

To summarize, the DPLL will first assign truth values to unit clauses, and when these do not exist anymore it will use the DLIS heuristic to choose a literal, instead of chosing it randomly. We now have defined a basic DPLL, a DPLL with DLCS, and a DPLL with DLIS which we will compare and use to answer the research question if having more available solutions lead to more or less searching through the DPLL tree.

4 Hypothesis

A Sudoku with a minimum of 17 cells given can contain only one possible solution. If there were less cells given, this means that there are multiple ways to solve the Sudoku. For a human being [10], a Sudoku with less given cells (thus more possible solutions) tend to be harder than Sudoku's with more given cells. For DPLL, this may not be the case. Thus we want to test if Sudoku's that are known to be more difficult for humans if it also takes for DPLL longer to solve. This returns three experiments: Sudoku's with multiple possible solutions: 4, 9, and 15 given cells. And for comparison also Sudoku's with 18 given cells, and thus a unique solution. We hypothesize that DPLL is able to find a solution for Sudoku's with multiple solution less efficient than a Sudoku with a unique

solution. This hypothesis comes from the fact that for humans it is also more difficult to solve these Sudoku's. Furthermore, we hypothesize that DPLL needs to assign and 'try' more values if there are more possible solutions. This is because more clauses would also restrict the search space for DPLL. If there are less clauses (thus more possible solutions) DPLL would likely to backtrace more and be less efficient in solving the Sudoku.

5 Experimental design

5.1 Overview

The experiment follows a specific setup. The following table represents this setup.

Sudoku (100 puzzles)	Algorithm	Runs
4 pre-filled cells	DPLL, DPLL + DLIS, DPLL + DLCS	2 x
9 pre-filled cells	DPLL, DPLL + DLIS, DPLL + DLCS	2 x
15 pre-filled cells	DPLL, DPLL + DLIS, DPLL + DLCS	2 x
18 pre-filled cells	DPLL, DPLL + DLIS, DPLL + DLCS	2 x

To evaluate the algorithm and the runs, the amount of backtracking is calculated. The least amount of backtracks will be seen as the optimal run. The most amount of backtracks will be seen as the most inefficient run.

5.2 Files

Each Sudoku experiment are *.cnf* encoded in their own file. The experiment directory consists of a textfile with the ruleset, and four files with *.cnf*, which encode the Sudoku's. Each of the files with the Sudoku's contain puzzles with a different amount of cells given. As noted, these are four, nine, fifteen, and eighteen given cells.

Each *.cnf* file contains 100 Sudoku's where the given value in the filename corresponds to the given cells given in the Sudoku. The Sudoku's are given in DIMACS format. DIMACS is a cnf format where the literals are represented as integers, and the negative integers are the negated literals. For DPLL to be able to solve the Sudoku, it also needs to have the Sudoku rules in DIMACS format. This experiment only focusses on 9x9 Sudoku's, so these rules also needs to be available in the experiment directory.

5.3 Running experiment

During the experiment, three different DPLL algorithms have been runned. Basic DPLL, DPLL with DLCS, and DPLL with DLIS. Each DPLL will try to solve all 400 Sudoku's and return a solution. After a solution to a given Sudoku has been found, the DPLL returns the solution of all the true values. We do this because the true values return all 81 values of the cells. Moreover, also the amount of backtracks is calculated. These backtracks can be used to evaluate and compare the algorithms with each other.

6 Experimental results

In this study, two measurements were recorded. For each Sudoku, the time it took to solve it in seconds is recorded, and most importantly to answer the research question stated in this paper, the amount of backtracks. In the experiment, as described in the experimental setup, the amount of backtracks and time it took to solve a Sudoku were measured when in the Sudoku four, nine, fifteen, and eighteen numbers were given already. Below, the resulting run time and number of backtracks are plotted. The number of backtracks have also been visualized

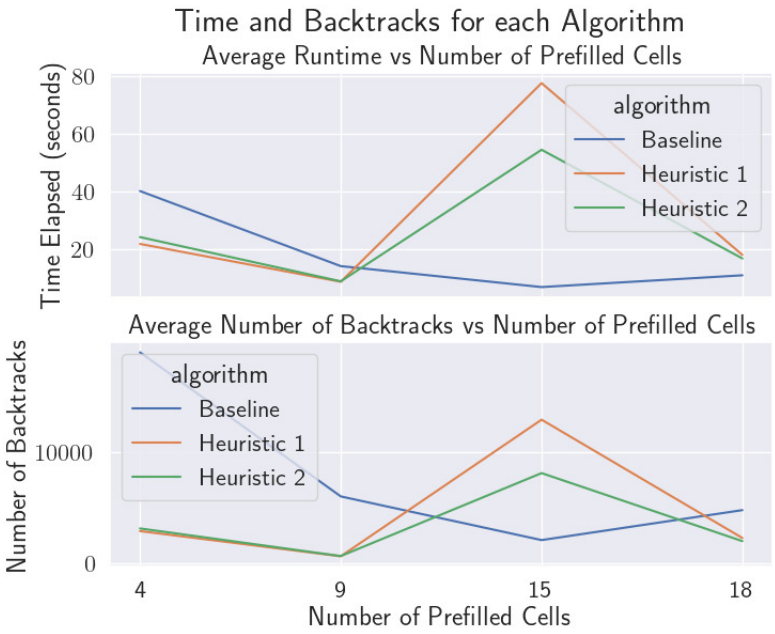

Fig. 1. Plot of the average run time, and number of backtracks

in a bar plot, which can be seen in figure 2.

To test the statistical significance of our results, we employ an unpaired two-sample t-test. We computed the following test statistic to check whether the number of backtracks differed significantly across the 3 DPLL methods.

$$z = \frac{(\bar{x}_i - \bar{x}_j)}{\sqrt{\frac{s_i^2}{n_i} + \frac{s_j^2}{n_j}}} \tag{1}$$

Table 1 contains the results of the t-test. Our null hypothesis is that there is no statistically significant difference between the number of backtracks between 2 groups of DPLL algorithms. Because our p-value is less than 0.05 for 'Baseline

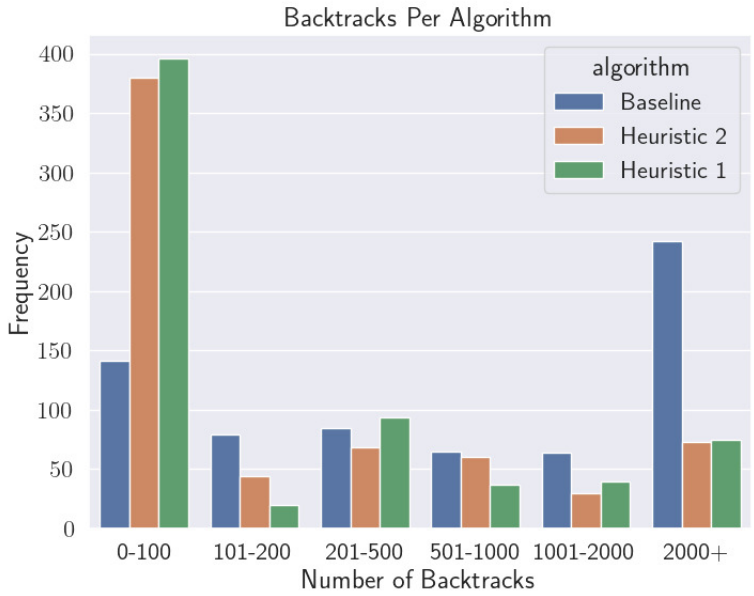

Fig. 2. Bar plot of the number of backtracks per algorithm

vs Heuristic-2', we are going to reject our null hypothesis for this. Which means there is a statistical difference between these two samples.

7 Conclusion

In this paper, the research question was whether more available solutions would lead to more or less searching through the DPLL tree. To answer this question, graphs were gathered, and t-tests were done to get insights if there is a significant difference. From table 1, it is clear that only basic DPLL with 4 solutions given and DPLL with DLCS solving with 9 solution given are significantly quicker than sudokus containing a unique solution. All the other experiments do not show any significant difference. This means that we did not receive enough evidence to assume that DPLL that solves sudokus with multiple solutions have a more difficult time compared to DPLL solving sudokus with an unique solution.

Moreover, from figure 1 and 2, it becomes apparent that the DPLLs using the implemented heuristics perform relatively the same. As can be seen from figure 1, a number of 15 given cells in a Sudoku puzzle resulted in averagely the most number of backtracks, and time spend to solve the given Sudoku's with this amount of given cells. Therefore intuitively, fifteen given cells seemed to be the most difficult category. However, do note that the baseline DPLL did better on this category than it did with all other categories of given cells, and only

Table 1. Results of statistical t-tests. Unique and Multi solution columns refer to the number of pre-filled values [10].

Algorithm	Unique Solution	Multi Solution	T-Value	P-Value
Baseline	18	4	-2.474	**0.013**
Baseline	18	9	-0.277	0.781
Baseline	18	15	0.690	0.490
Heuristic 1	18	4	-0.500	0.616
Heuristic 1	18	9	2.005	**0.046**
Heuristic 1	18	15	-1.616	0.107
Heuristic 2	18	4	-0.908	0.364
Heuristic 2	18	9	1.715	0.0876
Heuristic 2	18	15	-1.315	0.189

the heuristic DPLLs did worse. The explanation of this can be that the baseline DPLL is stochastic, and can get to a result faster due to good random selections of literals. From the deterministic characteristics of the DPLLs which use the implemented heuristics, we can see that the algorithms have it more difficult with 15 cells given, than with less than 15. However, with 18 cells given, the amount of backtracks of the two algorithms using heuristics are far less than with 15 cells given. The amount of backtracks are approximately the same again as with four and nine cells given. Therefore, the DPLLs which use heuristics seem to have more difficulties with more cells given in a Sudoku, while the stochastic basic DPLL was faster with more cells given, while it had more backtracks with less given cells than the algorithms which use the heuristics implemented in this study.

From figure 2, it can be seen that the DPLLs which use heuristics have a lot of runs with less than 100 backtracks. Therefore, these were able to solve more Sudoku's in less backtracks than the baseline DPLL. Also from this figure it becomes visible that the baseline heuristic has more runs for Sudoku's with 2000+ backtracks. Thus, the baseline is more likely to have an extreme number of backtracks for a given Sudoku, while an DPLL with the heuristics implemented in this study is more likely to solve a Sudoku with less backtracks than the baseline DPLL. This seems to be due to the fact that the DPLLs using heuristics use less backtracks with a minimal amount of cells given.

8 Limitations & Further Research

This paper has found some very interesting insights into how DPLL solves Sudoku's that can have multiple solutions. However, during the experiment also some other questions and insights sparked up. The first one being that some specific Sudokus took an extremely large time for DPLL to solve. When looking at the backtracks, these Sudokus could have more than 850,000 backtracks before a solution was found. This also shows that the structure and which cells are already given has quite an impact on the DPLL. It was out of this scope of this

paper to analyze the effect of the position of the given cells has on the DPLL, but this is something that is interesting to investigate in further research. Secondly, DPLL has been implemented with two specific heuristics. However, there are different heuristics that can be implemented that could possibly behave differently in this experiment. Finally, the results showed that Sudokus with 15 given cells were the most difficult to solve for the DPLL with heuristics. While for basic DPLL it did not have issues with solving these Sudokus efficiently. It was out of the scope of this paper to find out why this is the case, but it is interesting to delve into this fact in future research.

During the experiment, there were also some obstacles found that needs some clarification. When running the experiment multiple times, there are some difference in the amount of backtracks for the same algorithm and Sudokus. This is because the DPLL implementation also has some random factor. This is also the reason why the experiment runs multiple times. If DPLL would be implemented deterministically, then instead of running multiple times we could have ran it on more Sudoku samples. Unfortunately, due too time constraints it was not possible to implement it this way.

References

1. Eiben, A., & Smith, J. (2015). Introduction to Evolutionary Computing. Natural Computing Series, 58-61. doi: 10.1007/978-3-662-44874-8
2. Garey, M. R., Johnson, D. S., & Stockmeyer, L. (1974). Some simplified NP-complete problems. Proceedings of the Sixth Annual ACM Symposium on Theory of Computing - STOC '74, 47. https://doi.org/10.1145/800119.803884
3. Ullman, J. D. (1975). NP-complete scheduling problems. Journal of Computer and System Sciences, 10(3), 384–385. https://doi.org/10.1016/s0022-0000(75)80008-0
4. Tovey, C. A. (1984). A simplified NP-complete satisfiability problem. Discrete Applied Mathematics, 8(1), 85–89. https://doi.org/10.1016/0166-218x(84)90081-7
5. Ahmed, T. (1970, January 1). An implementation of the DPLL algorithm. Spectrum. Retrieved November 21, 2022, from https://spectrum.library.concordia.ca/id/eprint/976566/
6. Nieuwenhuis, R., Oliveras, A., & Tinelli, C. (2005). Abstract DPLL and abstract DPLL modulo theories. Logic for Programming, Artificial Intelligence, and Reasoning, 38–40. https://doi.org/10.1007/978-3-540-32275-7_3
7. Sang, T., Beame, P., & Kautz, H. (2005). Heuristics for fast exact model counting. Theory and Applications of Satisfiability Testing, 229. https://doi.org/10.1007/11499107_17
8. Socher, R. (1991). Optimizing the clausal normal form transformation. Journal of Automated Reasoning, 7(3), 325–329. https://doi.org/10.1007/bf00249017
9. Van Gelder, A. (2005). Pool resolution and its relation to regular resolution and DPLL with clause learning. Logic for Programming, Artificial Intelligence, and Reasoning, 583. https://doi.org/10.1007/11591191_40
10. arXiv, E. T. from the. (2020, April 2). Mathematicians solve minimum Sudoku problem. MIT Technology Review. Retrieved November 23, 2022, from https://www.technologyreview.com/2012/01/06/188520/mathematicians-solve-minimum-Sudoku-problem/