Influence of initial Sudoku variables on the computational performance of a DPLL implementation using several different heuristics

No Author Given

Vrije Universiteit, De Boelelaan 1105, Amsterdam

Abstract. Sudoku puzzles can be defined as a boolean satisfiability problem (SAT), which can then be represented as a propositional logic formula in conjunctive normal form. This simplifies the algorithmic solving of SAT problems. Davis–Putnam–Logemann–Loveland (DPLL) is one algorithm that has this capability, making use of some simplifications before assigning a variable and backtracking whenever the assignment makes the problem unsatisfiable. In this paper, we suggest an implementation of DPLL that makes use of five different heuristics to choose which variables to assign. We then investigate whether an increase in the number of initial variables in a Sudoku - which simplifies the problem for human solvers - also decreases the number of evaluations and backtracks of our algorithm. Our findings suggest that this is not the case, problems that are easier for human solvers require more computational effort from our algorithm. Further research could investigate whether the cause of our observed findings is specific to this implementation or a peculiarity of DPLL itself.

Keywords: SAT, boolean satisfiability, DPLL, Sudoku, Python, conjunctive normal form, DIMACS

1 Introduction

1.1 Basic DPLL

The Davis–Putnam–Logemann–Loveland (DPLL) algorithm [2][3] is a method of solving boolean satisfiability problems (SATs) that are represented in Conjunctive Normal Form. SAT problems are part of the NP-Complete set of problems, which means that any other problem in NP-Complete can be reduced to it. Hence it would be a great breakthrough if someone were to come up with an algorithm that could solve an SAT efficiently, i.e. in polynomial time. As of now, no such algorithm exists and thus the search continues. The best-case performance of DPLL is $O(1)$, but this is only in a trivial situation. The worst-case performance, however, gets as high as $O(2^n)$.

DPLL uses several simplifications and backtracking to solve an SAT problem or prove that it is unsatisfiable. The DPLL algorithm without heuristics is presented as pseudocode in 1

Algorithm 1 DPLL algorithm

input: *clauses* is the list of clauses
output: boolean value, *true* if *clauses* is satisfiable, else $false$
procedure DPLL(*clauses*)
    if $clauses = \varnothing$ **then return** *true*
    end if
    if *clauses* contains an empty clause **then return** $false$
    end if
    if *clauses* contains $p \vee \neg p$ **then return** DPLL($clauses/\{p \vee \neg p\}$)
    end if
    if *clauses* has unit clause $\{l\}$ **then return** DPLL($clauses\{l = true\}$)
    end if
    if *clauses* has pure literal l **then return** DPLL($clauses\{l = true\}$)
    end if
    if DPLL($clauses\{l = true\}$) is satisfiable **then return** *true*
    else return DPLL($clauses\{l = false\}$)
    end if
end procedure

This is a recursive function that takes a list of clauses in CNF form and returns a boolean, signifying the satsifiability of the input SAT problem.

First, it checks whether the list of clauses is empty, if that is the case it will return true immediately. Second, it verifies whether the list of clauses contains an empty clause and returns false if that is the case, since it would make the problem unsatisfiable.

Then it checks for tautologies, where a clause has both the positive and the negated version of a variable and removes these clauses before calling itself recursively. Note that this only has to be done once, since assigning or removing variables will not create new tautologies.

Next, it checks for unit clauses, which can immediately be assigned to true since that's the only way that the CNF would be satisfied.

The next step is often omitted since it is quite computationally intensive. In this step, it iterates over every single literal and then every clause and finds whether it only occurs with one sign (positive or negative). If this is the case, it can assign this clause to true (in case of only positive occurrences) or otherwise to false, before calling itself again.

Lastly, it starts to split by assigning either *true* or $false$ to one of the variables (using a heuristic, which will be discussed later). If the function returns false, it tries again but then assigning the opposite value.

This last step is an excellent point of improvement for the algorithm, since choosing the right variable can greatly simplify the problem and therefore increase the performance of the algorithm. Several heuristics have been implemented here, which are discussed in the following section.

1.2 Heuristics

MOM's heuristic MOM stands for Maximum Occurrence of Minimum size. The idea of this heuristic is to find literals that occur most often in the smallest clauses. The formula for this calculation is presented below.

$$[f(x) + f(x')] \cdot 2^k + f(x) \cdot f(x') \tag{1}$$

Where $f(x)$ and $f(x')$ are the number of occurrences in the smallest, non-satisfied clauses of x and negated x respectively. k represents a tuning parameter that can be altered to optimise the heuristic.

Jeroslow-Wang This heuristic assigns weights on clauses ω based on their length and then sums up the values from the following formula.

$$J(l) = \sum 2^{-|\omega|} \tag{2}$$

Where l is the literal and ω represents the length of each clause that literal is in. This equation sums up all these clause weights, with larger clauses (with a higher ω) receiving smaller weights. The chosen literal is the one that maximises this function, thus it is biased towards literals that are present in many small clauses. There are two versions of this heuristic:

- One-sided: In this version simply the literal with the highest value of $J(l)$ is selected.
- Two-sided: Here the value of both $J(l)$ and the negated $J(l')$ of each literal is computed and then the maximum sum of these two functions is computed, which corresponds to the variable to be assigned. If $J(l) \geq J(l')$, the variable is set to $true$, otherwise, it is set to $false$

Minimum occurences heuristic (MOH) gives highest priority to literals that have been seen the least amount of times during the entire SAT experiment. The literals involved are the ones that have not been assigned yet. The formula is as follows:

$$c(l)_{i+1} = \begin{cases} c(l)_i + 1, & \text{if } l \epsilon \omega_{i+1} \\ c(l)_i, & \text{otherwise} \end{cases} \tag{3}$$

where l is the literal to observe (typically every remaining literal in the current DPLL process), ω_{i+1} the clauses that are currently left and $c(l)_i$ is the previous result of the function (previous stage of the DPLL process). The goal is to minimize $c(l)_{i+1}$.

Length priority heuristic (LPH) prioritizes the literals that are in the shortest (remaining) clauses. The formula is as follows:

$$l(l) = \frac{1}{|\omega|} \tag{4}$$

Here, ω is the length of a given clause that contains l. The literal that maximizes this function, i.e. the literal which is in the shortest clause, is prioritized. When multiple literals have an equal priority, one of those is chosen at random.

1.3 Sudokus

A Sudoku is a game of numbers, where a square grid is presented with none or some numbers already given and the user (or an algorithm) then needs to find out which set of numbers would fill all the empty spaces while complying with the preset rules. There are various sizes of Sudokus possible, for example 4x4, 9x9 and 16x16. The most common one is 9x9, which consists of 81 squares, 9 on each side, where there can be a number from 1 to 9, with each number occurring exactly 9 times.[6]

The rules of a Sudoku are as follows:

- Each row should contain each number exactly once.
- Each column should contain each number exactly once.
- Each sub-square (in the case of a 9x9 Sudoku, this would be 3x3) should also contain each number exactly once.

1.4 Research

In this paper we analyse how the DPLL algorithm scales in terms of the given variables in a 9x9 Sudoku, with several different heuristics implemented. For a human user, the difficulty of solving a particular Sudoku is negatively correlated with the number of variables given, since each additional given variable brings the user closer to a solution and reduces the number of possible solutions. It would then seem logical for a Sudoku solver to scale in the same way, i.e. increasing in performance with a larger number of initial variables.

Therefore we expect the DPLL algorithm, regardless of the implemented heuristic, to require fewer evaluations to find the solution of a 9x9 Sudoku when there are more variables given. Additionally, we expect the algorithm to require fewer backtracks with more initial variables since it decreases the number of possible solutions. This would be because having fewer possible solutions means that the algorithm would not need to explore as much of the variable assignment tree.

RQ: *How is the performance of the DPLL algorithm using several different heuristics affected by a changing number of given variables in solving a 9x9 Sudoku?*

H1: *Increasing the number of given variables results in a decrease of evaluations needed for DPLL to solve a 9x9 Sudoku, regardless of implemented heuristics.*

H2: *Increasing the number of given variables results in a decrease in the number of backtracks executed by the DPLL algorithm in solving a 9x9 Sudoku, regardless of implemented heuristics.*

2 Experiment

2.1 Data preprocessing

Sudokus can be serialized in different ways. A commonly used format is a so-called "dotted"-format, where the dots represent the empty squares that need to be filled in, and the numbers represent the given starting numbers. As Sudokus are always squared, the square root determines the dimensionality. This means that for a 9x9 Sudoku, a new row begins after 9 characters. Below is an example of a serialized dotted-format Sudoku.

$$.94...13.............76..2.8..1.....32.........2...6.....5.4.......8..7..63.4..8 \tag{5}$$

As dotted-format is rather unhandy, for SAT-solvers especially, all used Sudokus are being converted to DIMACS-format beforehand. This is a standardized format for SAT-solvers. In DIMACS, all individual lines represent disjunctions, whereas the lines together are conjunctions. This is also called Conjunctive Normal Form (CNF). Each variable in DIMACS can either be positive or negative, which determines whether it is non-negated or negated, respectively [4].

The constraints of a Sudoku can also be presented in DIMACS-format. Because DIMACS is a serialization of CNF, both the constraints and the Sudoku instance to be solved can be merged, thus reducing to a single SAT-problem.

2.2 DPLL implementation

Specific implementation choices drastically impact the performance of DPLL. Although processing time is not a variable this research aims to focus on (since it depends on the hardware), it improves the reproducibility of the experiment. Possibilities of optimization are mainly located around the tracking of clauses and literals.

As stated earlier, the DPLL algorithm solves conjunctions of disjunctions. This can be represented as a two-dimensional array, f.e. $\{\{1 \vee 2 \vee 3\} \wedge \{\neg 1 \vee 2 \vee \neg 3\}\}$. Let s_1 be the number of clauses and s_2 the sum of the length of all clauses. When searching all clauses for a literal l, the lookup time of a two-dimensional array will then yield a worst complexity of:

$$\mathcal{O}(\sum_{i=1}^{n} |w_i|) \tag{6}$$

where w_i is the length of a specific clause i and n is the number of clauses. In other words, the bigger the problem gets, i.e. the more clauses to be solved, the higher the complexity becomes, and the slower a solution is found.

One solution to reduce the complexity is by implementing indices. An example is a hash table. Hash tables have at worst a complexity of $\mathcal{O}(n)$. However, their average complexity is $\mathcal{O}(1)$ (Constant Running Time)[5][1], and only is worse with a poorly implemented hashing algorithm.

To benefit from hash tables, they have been implemented in two ways. Firstly, instead of an array, clauses are being tracked in a hash table. Every (unique) clause gets an id assigned (c_idx_i), which is used as the key in the hash table. A number of n unique clauses will lead to an n sized hash table (See listing 1).

After generating a hash table for all clauses, a hash table for all literals is made (See listing 2). This way, a reference to all clauses the literal is present in, is tracked. This makes it possible to directly point to a specific clause, instead of searching all clauses. With hash tables, the complexity to find all clauses containing -3 now yields $\mathcal{O}(1+1)$, whereas the two-dimensional array yielded $\mathcal{O}(3+3)$. This, of course, differs drastically on an increasing number of literals and clauses.

Listing 1 Example clauses hash table of $\{\{1 \vee 2 \vee 3\} \wedge \{\neg 1 \vee 2 \vee \neg 3\}\}$

```
{
    "1_2_3": [1, 2, 3]
    "-1_2_-3": [-1, 2, -3]
}
```

Listing 2 Example literals hash table of $\{\{1 \vee 2 \vee 3\} \wedge \{\neg 1 \vee \neg 3\}\}$

```
{
    "1": ["1_2_3"],
    "2": ["1_2_3", "-1_2_-3"],
    "3": ["1_2_3"],
    "-1": ["-1_2_-3"],
    "-3": ["-1_2_-3"]
}
```

2.3 Data collection

To be able to analyze the hypothesis and make conclusions, an experiment has been set up. In order to make fair comparisons, 1000 different Sudokus have been selected, with a different number of initial variables, ranging from 21 to

29. The Sudokus have been randomly selected from a big Sudoku pool and were therefore normally distributed. The variables that have been tracked, for each Sudoku combined with each heuristic, are the heuristic itself, the Sudoku-id (a unique id for a Sudoku), the number of initial variables, the number of DPLL evaluations and the number of DPLL backtracks.

3 Results

3.1 Statistics

Figure 1 shows the number of evaluations and backtracks for different numbers of given variables, sorted by the heuristic. Pearson's Correlation Coefficient was used to determine whether the number of initially given variables correlated with either the number of evaluations or the number of backtracks. For both conditions, a moderately positive correlation was found, $r(6064)=0.027$, $p=0.035$. For the length_priority_max and mom heuristics, a strong positive correlation was found in both conditions, see Table 1.

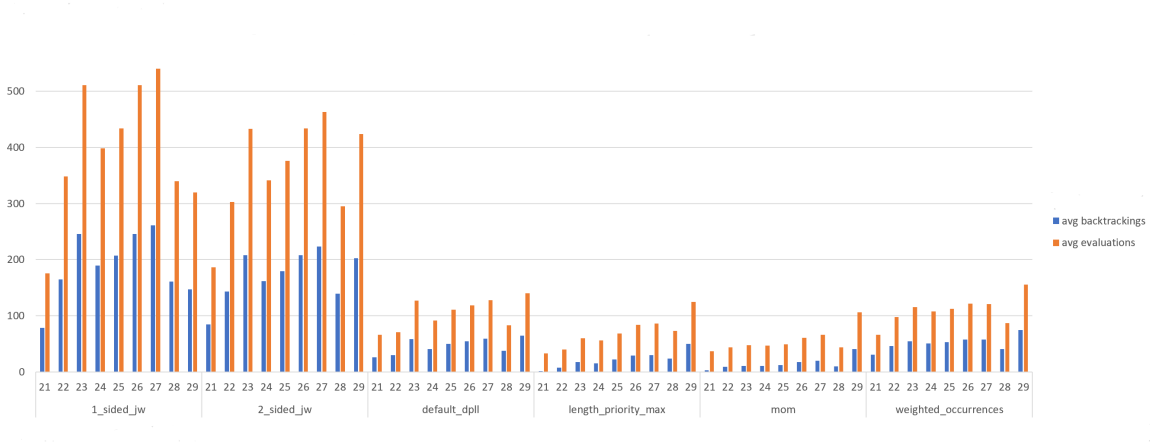

Fig. 1: Results

3.2 Animations

In the appendix, one can find animations of the DPLL algorithm solving the same Sudoku using each of the implemented heuristics.

Table 1: Statistics by heuristic

Heuristic	Hypothesis 1		Hypothesis 2	
	r-value	p-value	r-value	p-value
1_sided_jw	0.032	0.312	0.032	0.313
2_sided_jw	0.034	0.278	0.034	0.277
default_dpll	0.043	0.167	0.045	0.157
length_priority_max	0.141	7.10e-6	0.141	6.31e-6
mom	0.110	4.38e-4	0.115	2.56e-4
weighted_occurrences	0.026	0.417	0.025	0.421

4 Conclusion

Our findings suggest that both the presented hypotheses should be rejected, since a moderately positive correlation is found, instead of the expected negative correlation. Hence, an increase of given variables causes an increase in both the number of evaluations and the number of backtracks. It thus appears that Sudokus which are easier for a human to solve, require more computations from our algorithm. When investigating the nature of given variables in a Sudoku puzzle more closely, it becomes more clear what could be the cause behind our unexpected results. An increase of given variables brings a solver closer to the solution, by decreasing the number of possible correct solutions until eventually there exists only one. For a human solver, this simplifies the problem, since it would aim towards a specific solution related to the problem context of a Sudoku and backtrack when the problem becomes unsatisfiable. For an algorithm that scans the solution space indiscriminately by assigning variables based on heuristics not specific to Sudokus, however, this could cause an increase in computational effort.

5 Discussion

This paper illuminates several possible paths that future research may go into. One possibility would be to investigate the differences between the methods of a human user and each of the heuristics described in this paper and then compare these results. Perhaps such an investigation would result in discovering a novel heuristic based on the human approach to solving Sudokus. Another possible way that research into this topic could go is making comparisons between different Sudoku sizes, such as 4x4, 9x9 and 16x16 and then analysing how scalable these algorithms really are. Yet another possibility would be to apply the same algorithm to other SAT problems and then observe whether each of the heuristics performs differently for different kinds of problems, thus showing some sort of bias towards a certain problem.

References

1. Amarif, M. & Alashoury, I. (2019, February). The Implicit Path Cost Optimization in Dijkstra Algorithm using Hash Map Data Structure. In *International Conference on Computer Sciences Information Technology COSIT2019 (CS IT)* (pp. 33-44). https://doi.org/10.5121/csit.2019.90204
2. Davis, M., Logemann, G. & D. Loveland. (1962). A machine program for theorem proving. *Comm. ACM* 5(7), 394–397.
3. Davis, M. & H.Putnam. (1960). A computing procedure for quantification theory. *Journal of the ACM (JACM)* 7(3), 201-215.
4. DIMACS, C. (1993). Satisfiability: Suggested format. *DIMACS Challenge. DIMACS*, 32. Available online at ftp://dimacs. rutgers.edu/pub/challenge/satisfiability/doc.
5. Mohr, A. (2014). Quantum computing in complexity theory and theory of computation. *Carbondale, IL*, 1.
6. Simonis, H. (2005, October). Sudoku as a constraint problem. In: CP Workshop on modeling and reformulating Constraint Satisfaction Problems (Vol. 12, pp. 13-27).Citeseer.

Appendix

Animations of DPLL heuristics

- Standard DPLL: https://imgur.com/a/Awjpp2P
- MOM heuristic: https://imgur.com/IwmEXp5
- One-sided Jeroslow-Wang: https://imgur.com/v2RhtKq
- Two-sided Jeroslow-Wang: https://imgur.com/uVBRv8a
- Length priority heuristic: https://imgur.com/pZUS9WC
- Minimum occurrences heuristic: https://imgur.com/xMHRDvS