On the Predictive Ability of the Simplification Hypothesis

Eva Biesot[2777979], Lucas Monne[2787725], and Chi Him Ng[2748786]

Vrije Universiteit, Amsterdam Boelelaan 1105, 1081 HV, The Netherlands

Abstract. The Simplification Hypothesis states that for any given problem better branching heuristics tend to create simpler sub-problems. Despite its success in leading to the improvement of the Jeroslow-Wang One-Sided heuristic, this hypothesis has not been explored with regards to other high-performing heuristics. Furthermore little is known over the meaning of this hypothesis to efficiency. This paper extends previous works on the Simplification Hypothesis by investigating the relationship between branching frequency and efficiency. Results adhere to initial predictions but go against unexpected odds. Ultimately, confirming that the hypothesis holds for other algorithms, while offering grounds for discussion of a possible powerful bias for solving sudokus using SAT solvers.

Keywords: Knowledge Representation · DPLL · Heuristics · SAT · Sudoku · branching · JW-TS · VSIDS.

1 Introduction

The propositional satisfiability problem (SAT) is the quintessential NP-complete problem wherein one must determine if there exists an interpretation that satisfies a given Boolean formula [1]. In the last two decades, a considerable amount of techniques and heuristics have been proposed to solve such problems [8]. Of these, backtracking algorithms such as Conflict Driven Clause Learning (CDCL) have shown some of the best results known in the literature [9]. A key feature of backtrack search SAT algorithms consists of the reasoning used to select and assign values to variables at each step: the branching heuristic. For any given search tree, the size of the tree can be reduced vastly, depending on the implementation of the branching heuristic. For example, CDCL SAT solvers are reliant on the Variable State Independent Decaying Sum (VSIDS) [10] heuristic for their performance. In an attempt to generalize the trend in performance of branching rules, [5] uses the two Jeroslow-Wang heuristics to propose the Simplification Hypothesis, stating that all else being equal, the simpler the subproblems created by a branching heuristic, the better the performance it will exhibit. If the Simplification Hypothesis is true; if heuristics which simplify a problem to greater extent are theorized to outperform those which do not, then greedy algorithms prioritizing the selection of variables which create the largest number of unit clauses should outperform any other heuristic on average. Therefore, heuristics which visit the smallest areas of a given search space, should exhibit

superior performance. In spite of this hypothesis succeeding in improving from the One-sided Jeroslow-Wang [6] algorithm to the Two-sided variant (JW-TS), there exists little literature exploring its truth for other algorithms. For instance, VSIDS, which outperforms JW-TS, is stated to only explore small communities of variables [8]. Yet, little is established with regards to the relationship between the branching frequency of the heuristic and its influence on efficiency. We define branching frequency as the area of the search space a given branching heuristic must cover before a solution is found. Hence, in an effort to extend the literature, explore the works of [5] and ultimately shed more light on the consistency of the Simplification Hypothesis, this paper attempts to answer the question: *Accordingly with the Simplification Hypothesis, to what extent does the branching frequency of a heuristic affect efficiency?* using a comparison between Two-sided Jeroslow-Wang and VSIDS.

Thus, this research tackles this question by means of the implementation of the Davis-Putnam-Logemann-Loveland algorithm (DPLL) [3, 2] and the aforementioned heuristics, comparing the area of the search space covered by each strategy and analyzing its performance on Sudoku puzzles with regards to creating smaller sub-problems.

2 Background

2.1 Davis-Putnam-Logemann-Loveland Algorithm

The DPLL algorithm has been and remains one of the most popular algorithms for solving SAT problems [11]. Given a knowledge base K and a literal l, $K|l$ is the "residual formula" obtained from K when $f(l)$ is set to 1. This formula is obtained by removing all clauses containing l, deleting all occurrences of its negation, $\bar{l}$, and removing both l and $\bar{l}$ from the list of remaining literals. Additionally, clauses with only one literal (unit clauses), and variables which only appear as either positive or negative (pure literals) are given a truth assignment f of $f(l) = 1$. These rules, applied recursively until K contains no more clauses, are the backbone of the DPLL algorithm [3, 2]. The full procedure is depicted below.

As the DPLL procedure's performance is highly dependent on the heuristic it uses, it is the ideal medium of observation for the comparison of the following two heuristics, justifying its usage.

2.2 Two-Sided Jeroslow Wang

JW-TS is an improvement on the One-sided Jeroslow-Wang heuristic [5]. This strategy computes a score $J(x) + J(x')$, where:

$$J(x) = \sum 2^{-|\omega|} \tag{1}$$

With ω being the length of the clause in which the variable occurs. The one sided variant only looks at positive values, thus if the variable is a negation,

Algorithm 1 DPLL

```
function DPLL(φ)
    while there is a unit clause l in φ do
        φ ← unit-propagate(φ)
    end while
    while there is a literal l that occurs pure in φ do
        φ ← pure-literal-assign(φ)
    end while
    if φ is empty then
        return true
    end if
    if φ contains an empty clause then
        return false
    end if
    l ← choose-literal(φ)
    return DPLL(φ ∧ {l}) or DPLL(φ ∧ {l̄})
```

the algorithm will ignore this clause. JW-TS will count all clauses, disregarding whether the variable is negated or not. After the values are calculated, JW-TS will select the highest value for which the split will take place. As this heuristic is introduced alongside the Simplification Hypothesis in [5] and this paper seeks to extend it, it is sensible to use it for comparison.

2.3 VSIDS

Introduced in the CHAFF algorithm [10], VSIDS follows the following strategy:

(1) Each variable in each polarity has a counter initialized to 0
(2) When a clause is added to the knowledge base, the counter with each literal in the clause is increased
(3) The (unassigned) variable with the highest counter is chosen for assignment
(4) Ties are broken randomly
(5) Counters are divided by a constant p periodically

This particular heuristic was chosen due to its noted superiority in performance when compared to JW-TS, allowing one to determine whether the Simplification Hypothesis holds for superior algorithms of a different nature.

2.4 Hypothesis

Intuitively, since historically, VSIDS is known to perform better than JW-TS, it would be sensible that the former will explore a smaller area of the search space, consequently demonstrating a lower branching frequency. However, taking that VSIDS chooses the variables with the highest count of appearance for a given

polarity, seems to point towards a performance for VSIDS which may yield itself to be reliant on the initial unit clauses given in the sudoku. Additionally, since no clauses are added to the knowledge base over the course, it may also hamper the performance of this heuristic as its decay parameter will have little influence. On the other hand, JW-TS keeps track of counts regardless of polarity, hence it should be less reliant on the initial unit clauses provided by the sudoku. Therefore, we hypothesize that the Simplification Hypothesis does not necessarily hold, and that VSIDS will not offer lower branching frequency than JW-TS.

3 Method

3.1 Experimental Design

To determine the effect of branching frequency on a heuristic's ability to create simpler subproblems, three SAT solvers– no heuristic (basic DPLL), DPLL with VSIDS and DPLL with JW-TS –were tested on a large dataset of Sudokus. For each condition, the algorithms were run on the same test set once to five times and the branching frequency was recorded for each test-problem. The average branching frequency was used to compare between conditions. Due to the derminism of the basic DPLL condition, this particular algorithm was only run once on the test set. However, as VSIDS and JW-TS can have a degree of stochasticity for ties, these were run five times to account for any possible errors due to randomness.

3.2 Design Choices

DPLL The implementation of the basic DPLL algorithm used for this experiment is centered around efficient running time, in terms of the search space visited such that the DPLL algorithm's branching is optimized by default. This justifies the usage of recursion as it facilitates the maintenance of a counter for branching frequency. Note that the usage of recursion comes at the cost of space-complexity due to the increasing number of states which must be held in the program's stack. However, time/space complexity is not relevant to this experiment. The basic algorithm keeps track of its remaining variables in a stack, where items are stored in descending numerical order. This is an emergent effect of creating this stack in parallel with the first call of the unit propagation function. Branching for the basic DPLL condition is implemented such that the last variable of said stack is popped, and assigned a value prioritizing 0 (false), followed by 1 (true). As there are always significantly more false assignments than true ones, it is sensible to bias the DPLL solver with prioritizing values of 0. Hence, this choice of variable assignment optimizes the algorithm's branching, ensuring that the data obtained offers the best comparison against the two other heuristics.

VSIDS A MiniSAT with a decay factor of 0.95 used in [8] demonstrated the best performance amongst other implementations presented within the experiment. Therefore, the decay factor used in this VSIDS also stands at 0.95. Additionally, following [10] ties between variables are chosen to be broken randomly.

JW-TS Two-sided Jeroslow Wang gives exponentially higher weights to shorter clauses. Since the basic DPLL keeps track of the remaining variables after every recursion, the algorithm will look at the number and length of clauses if one variable gets assigned a value. Additionally, following [6] ties between variables are chosen to be broken randomly.

3.3 Test Set

For this experiment, a large group of test cases with varying complexity was necessary to ensure an adequate coverage of the aptitude of each heuristic. Additionally, an algorithm to parse and convert the test cases into a suitable problem representation was required.

The sudoku problems were obtained across a number of unnamed databases. Test cases were divided in subsets with varying size ranging from 4x4 to 9x9. Furthermore, there was a division with subsets depending on the complexity, with complexity increasing across test cases at a logarithmic rate. Each subset was provided in a single file, with each line in the file containing all the rows of a single sudoku. A total of 1999 sudokus were used for testing.

DIMACS format was opted for as the appropriate problem representation used for the test cases. The DIMACS conjunctive normal form (CNF) format is a textual representation of a formula in CNF. DIMACS CNF was chosen as satisfiability problems as it represents literals in a simplified manner, allowing algorithms to solve problems represented as such with more ease. Conversion to DIMACS is done while following the rules for sudoku. In short, the rules are translated as follows: for an n x n sudoku, all cells must contain a value between 1 and n, while all possible values for every cell, row, column and box have been verified. Simultaneously, for every value for each row, a value can not occur twice. The same rule applies when verifying each value for a column and a grid box. An empty box will be given a value of 0.

3.4 Evaluation

In the comparison between conditions, the branching frequency is used as a measure of the search space visited by the branching heuristics, thereby yielding a measure of a heuristics' ability to create simpler sub-problems. Branching frequency is recorded by keeping count of the total amount of times a given heuristic splits for a given problem. Furthermore, the time for solving a problem, in seconds, was also recorded. Running time can not be used as a measure of absolute efficiency due to differences in CPU and hardware across computers. However, it is safe to use it as a measure of relative efficiency and running

time is often used in time-analyses of the DPLL procedure [4, **?**]. Since the pre-established research question aims to explore efficiency through a comparison of JW-TS and VSIDS' efficiency, running time is used as a measure to compare efficiency between conditions.

4 Results

As explained before, the experiment compared the branching frequency of the three heuristics. The average branching frequencies for each condition are presented in table 1. The mean branching frequency and its standard deviation for the heuristic over all sudokus are shown. Furthermore the lowest and highest amount of branching for a single sudoku is also presented. All sudokus used in experimenting were solvable. Average running times for all conditions are shown in table 2.

The mean frequency of branching was the lowest for the basic DPLL, followed by JW-TS and VSIDS. The highest number of branching while looking for a solution for a single sudoku, was the highest for JW-TS. Despite this difference, the standard deviation of JW-TS was lower when compared to VSIDS. Similarly, the basic DPLL condition exhibits the fastest relative running time, with JW-TS coming second.

Table 1. Results for branching frequency for each version of the SAT solver (n = 1999)

Heuristic	Mean Branching Frequency	Std. Deviation	Minimum	Maximum
DPLL Basic	201	869	0	26928
JW-TS	1021	4323	0	98193
VSIDS	2589	5219	0	41656

Table 2. Average running time for the three heuristics (n = 1999)

Heuristic	Average Time	Std. deviation	Minimum	Maximum
DPLL Basic	2.545	9.112	0.002	276.886
JW-TS	12.76	52.457	0.002	1277.755
VSIDS	31.126	64.227	0.002	463.017

Closer inspection of the results leads to the observation that majority of the 4x4 Sudoku were solved while only using unit-propagation. Since branching did not take place in these instances, another table (3) was created to present the results where branching occured. The mean was naturally higher. However, the results between the heuristics did not change in relation to each other. The corresponding running times are shown in table 4.

Table 3. Results for branching frequency for each version of the SAT solver, where branching occurred (n = 998)

Heuristic	Mean Branching Frequency	Std. Deviation	Minimum	Maximum
DPLL Basic	400	1197	1	26928
JW-TS	1021	5944	1	98193
VSIDS	5182	6410	1	41656

Table 4. Average running time for the three heuristics (n = 998)

Heuristic	Average Time	Std. deviation	Minimum	Maximum
DPLL Basic	2.545	12.381	0.523	276.886
JW-TS	25.542	72.007	0.386	1277.755
VSIDS	62.280	79.476	0.559	463.017

The results were tested to determine if the difference in mean measurements for heuristics were statistically significant. Since the data was not normally distributed and the same instances with different outcomes were compared, the assumption of normality was not met. Hence, a non-parametric test, the Wilcoxon Rank sum Test, was used. The basic DPLL had significant less branching when compared to JW-TS (Z = -5.84, p = <.05) and VSIDS (Z = -7.30, p = <.05). Furthermore, a significant difference can be seen between JW-TS and VSIDS (Z = -5.41, p = <.05). For the results where branching did occur, the difference between DPLL and JW-TS (Z = -8.60, p = <.05) and VSIDS (Z = -9.37, p = <.05) were statistically significant. Lastly, the same could be observed for JW-TS and VSIDS (Z = -9.12, p = <.05).

When looking at the whole dataset the difference between DPLL and VSIDS were significantly different (Z = -9.28, p = <.05) and also for JW-TS (Z = -8.45, p = <.05). The difference between JW-TS and VSIDS was also significant (Z = 2.99, p = <.05). When only looking at the instances where branching occured, the difference between DPLL and JW-TS was significant (Z = -5.67, p = <.05) for VSIDS, this was (Z = -9.84, p = <.05). The difference between JW-TS and VSIDS was significant (Z = -7.96, p = <.05).

5 Discussion

Results found that the simple DPLL algorithm with no heuristics exhibited lower average branching frequency relative to the Two-Sided Jeroslow-Wang and VSIDS heuristics. VSIDS branches more than JW-TS. The results also show that DPLL needs the least time to solve the sudokus, followed by JW-TS and then VSIDS. Thus, rankings for both average efficiency (running time) and branching frequency are in the increasing order of DPLL basic, JW-TS, and VSIDS. These results support the original hypothesis but yield mixed interpretations with regards to the Simplification Hypothesis while suggesting a positive relationship between running time and branching frequency.

As hypothesized, VSIDS did not offer lower branching frequency than JW-TS. This is likely due to the fact that VSIDS excels in environments where the decay parameter is able to accomplish its functionality. That is, VSIDS functions best in conjunction with algorithms or problems where new clauses are added to the KB and the decay parameter is therefore utilized. On the other hand, in sudoku problems no new clauses are added over time. Additionally, VSIDS is used as a standalone branching heuristic, without any other algorithm which adds clauses such as the commonly used CDCL [8].

In addition to VSIDS underperforming with regards to branching frequency and time, it is worth noting that in this context DPLL basic outperforms both JW-TS and VSIDS. This goes against expectations as the heuristics should lead to more efficient solving according to [5]. After all, JW-TS and VSIDS are expected to create simpler subproblems than an algorithm with no heuristic as both attempt to choose variables which create more unit clauses [10, **?**]. Two explanations are relevant. First, with regards to efficiency, DPLL basic likely outperforms VSIDS and JW-TS as both heuristics require running through each variable in each clause, adding a substantial amount of time to the variable assignment (splitting) step. This is comparable to the usage of the pure literal rule, which increases running time exponentially [4]. Last, with regards to branching frequency, DPLL basic outperforms the two heuristics. The reason behind this lies in the nature of sudokus and the representation of rules in CNF. For any given NxN sudoku represented in CNF, knowledge bases contain the same rules, and initial known positions are represented as unit clauses. However, as all sudoku rules are written in terms of the possible variables for any given position, the cumulative sums of each possible variable appearing in such rules are identical. In a given algorithm run, unit clauses are first removed and a variable is chosen/assigned last. Hence, by the time variables are chosen, only sudoku rules are left. Though unit propagation will create some inequalities with regards to the distribution of variables present in clauses, such clauses quickly become unit clauses. This implies that for a large portion of the search, there is a mostly equal distribution of variables across the knowledge base. As JW-TS and VSIDS choose variables based on the most frequently occurring variable across all clauses of the knowledge base, this leads to numerous ties, which are broken randomly in this implementation. With this in mind, as DPLL basic chooses variables in ascending order based on a reversed sorted list of unassigned variables, this grants it a bias. That is, for any given sudoku, DPLL basic will always begin assigning values, prioritizing false, to all items on the first row and items on the first column. Solving a puzzle column by column or row by row proves itself a powerful bias as it ensures that the algorithm fills and ascertains a given set of 1-9 positions before moving on to another [7]. In contrast, the generally random assignment provided by JW-TS and VSIDS offers little competition to such a bias. Hence, our implementation for DPLL basic likely branches less often due to an unintentionally introduced bias. Moving forward, one simple way to verify this would consist of rerunning the experiment using random variable assignment for the basic implementation.

In terms of improvements, heuristics could have been optimized for sudoku-solving for better results. This would have yielded a fairer comparison given the possible bias conveyed by the basic implementation. For instance, JW-TS could be improved by not just using it to produce a variable to assign, but also to determine the value to assign. This would be done similarly do the method used for Dynamic Largest Individual Sums (DLIS) [12], by assigning a value false if the variable occurs more negated than non-negated and true if it occurs more non-negated than negated. Doing so would ensure a higher probability of assigning the correct value to any given variable [13] VSIDS could mostly be improved by adding some kind of clause learning heuristic, such as CDCL. This would then allow the decay parameter to function as intended, giving a higher weight to the learned clauses.

6 Conclusion

In conclusion, branching frequency seems to hold a direct relationship with running time and the Simplification Hypothesis seems to remain true for other heuristics than the Jeroslow-Wang strategies. A possible bias found serendipitously has demonstrated superior efficiency and performance (according to the Simplification Hypothesis) in solving sudokus, compared to renowned heuristics such as JW-TS and VSIDS. Further testing using a DPLL implementation which chooses its variables randomly is necessary to establish whether the bias is responsible for DPLL ranking above the aforementioned heuristics with regards to branching frequency. Additional testing using improved versions of JW-TS and VSIDS could be utilized to explore the extent of the relationship between branching frequency and efficiency. Alternatively, testing with highly difficult sudokus, where this bias may not be as effective may also prove an experiment worth pursuing.

References

1. Cook, S.A.: The complexity of theorem-proving procedures. In: Proceedings of the Third Annual ACM Symposium on Theory of Computing. p. 151–158. STOC '71, Association for Computing Machinery, New York, NY, USA (1971). https://doi.org/10.1145/800157.805047, https://doi.org/10.1145/800157.805047
2. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM **5**(7), 394–397 (jul 1962). https://doi.org/10.1145/368273.368557, https://doi.org/10.1145/368273.368557
3. Davis, M.D., Putnam, H.: A computing procedure for quantification theory. J. ACM **7**, 201–215 (1960)
4. Goldberg, A., Purdom, P., Brown, C.: Average time analysis of simplified davis-putnam procedures. Information Processing Letters **15**(2), 72–75 (1982). https://doi.org/https://doi.org/10.1016/0020-0190(82)90110-7, https://www.sciencedirect.com/science/article/pii/0020019082901107
5. Hooker, J., Vinay, V.: Branching rules for satisfiability, vol. 15, pp. 426–437 (01 2006). https://doi.org/10.1007/3-540-58715-2$_1$43

6. Jeroslow, R.G., Wang, J.: Solving propositional satisfiability problems. Annals of Mathematics and Artificial Intelligence **1**, 167–187 (2005)
7. Lee, N.L., Goodwin, G.P., Johnson-Laird, P.N.: The psychological puzzle of sudoku. Thinking & Reasoning **14**(4), 342–364 (2008)
8. Liang, J.H., Ganesh, V., Zulkoski, E., Zaman, A., Czarnecki, K.: Understanding vsids branching heuristics in conflict-driven clause-learning sat solvers (2015). https://doi.org/10.48550/ARXIV.1506.08905, https://arxiv.org/abs/1506.08905
9. Marques-Silva, J., Sakallah, K.: Grasp: a search algorithm for propositional satisfiability. IEEE Transactions on Computers **48**(5), 506–521 (1999). https://doi.org/10.1109/12.769433
10. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient sat solver. In: Proceedings of the 38th Annual Design Automation Conference. p. 530–535. DAC '01, Association for Computing Machinery, New York, NY, USA (2001). https://doi.org/10.1145/378239.379017, https://doi.org/10.1145/378239.379017
11. Ouyang, M.: How good are branching rules in dpll? Discrete Applied Mathematics **89**(1), 281–286 (1998). https://doi.org/https://doi.org/10.1016/S0166-218X(98)00045-6, https://www.sciencedirect.com/science/article/pii/S0166218X98000456
12. Sang, T., Bacchus, F., Beame, P., Kautz, H.A., Pitassi, T.: Combining component caching and clause learning for effective model counting. In: International Conference on Theory and Applications of Satisfiability Testing (2004)
13. Sang, T., Beame, P., Kautz, H.: Heuristics for fast exact model counting. In: Bacchus, F., Walsh, T. (eds.) Theory and Applications of Satisfiability Testing. pp. 226–240. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)