Investigating the dynamics of branching heuristics for the Davis Putnam, Loveland and Logemann algorithm for solving Sudokus

Elias Kassapis and Konstantin Todorov

March 7, 2019

I. INTRODUCTION

Propositional satisfiability (SAT) is the problem of determining whether a configuration of the values of the variables of a given propositional logic formula exists that can satisfy it. It is a well-known NP-complete problem with a multitude of theoretical and practical applications in many fields of Computer Science and Engineering. As a result, research on practical SAT solvers has become very popular, leading to the development of several solver algorithms. Many SAT solvers use variable splitting in order to search for a satisfying assignment, and most of these are based on the Davis Putnam, Loveland and Logemann algorithm (DP algorithm) [Zuim et al.2008].

The DP algorithm is a backtracking-based search algorithm used for many NP-complete problems, and consists of two main stages. The first stage is pruning the search space based on simplification rules and falsified clauses, and the second stage is the variable splitting which guides the exploration of the search space in the process of deciding on the satisfiability of the given propositional formula. A critical aspect of this algorithm is how the variables to undergo splitting are selected at each decision step of the algorithm[Zuim et al.2008]. Various extensions to the basic DP algorithm using branching heuristics have been proposed. These have highlighted the importance of clause analysis and locality during the search space pruning stage [Li and Anbulagan2000].

The simplest branching heuristic, which can be considered as the default DP branching heuristic, samples the next variable for splitting randomly from the set of unassigned variables. This heuristic is known as RAND. However, most branching heuristics utilize dynamic feedback collected of the search progress that rein the direction of the search exploration, giving rise to adaptive branching [Freeman1995]. One of the most commonly used branching heuristics is the Maximum Occurrences on clauses of Minimum size heuristic (MOM's) [Marques-Silva1999a]. This involves maximizing a function of the current variable and clause state of the problem. Several MOM's heuristic variants have been proposed which implement different versions of the heuristic function [Freeman1995].

In this paper, our aim was to investigate the effect of branching heuristic extensions of the basic DP procedure on the search dynamics of the algorithm. We propose a new MOM's variant which we coin as the TK1 heuristic. It is technically a generalization of MOM's heuristic which examines variables from all clauses instead of only the clauses of minimum size, but also incorporates extra information in the heuristic function to be optimized, with the intention on improving the quality of the feedback collected, therefore improve our branching strategy. We examined and analyzed the performance of MOM's heuristic and TK1, and compared them using the basic DP algorithm with the RAND heuristic as control. For this we used Sudoku puzzles, which are essentially a constraint satisfaction problem that require a combination of logical reasoning, combinatorics, and trial and error. Therefore the systematic backtracking search procedure used to explore the potentially exponentially sized space of variable assignment, and constrain propagation property of DP make the Sudoku problem a suitable substrate on which we can analyze the performance of such algorithms [Lynce and Ouaknine2006].

We hypothesized that TK1 will outperform MOM's heuristic because it better constrains the search. Our hypothesis is based on the premise that these two heuristics lie on the same spectrum of branching heuristics that by analyzing the current state of the search select the most constrained literal to branch on, hence assigning a value to that literal will massively constrain the search, yielding better results.

To test our hypothesis we first encode the Sudoku puzzles in a propositional formula in a DIMACS format for normal clausal form. Subsequently we present the formula to our SAT solver implementing the different DP variants to find a satisfying assignment, therefore solving the game. We evaluated performance by computing the number of splits of the variables selected for branching, and number of splits that did not result to backtracks. We found that both MOM's heuristic and the TK1 DP variants outperformed DP with RAND, however, the two had comparable performances.

II. BACKGROUND

MODELS

This section describes the architecture details of the basic DP algorithm with the RAND heuristic, MOM's heuristic, and our proposed branching heuristic, TK1.

Basic DP algorithm:

The pseudo-code of our DP SAT solver algorithm is illustrated below in Algorithm 1.

```
DP(Σ):
    if Σ = ∅ then
        return satisfied;
    end
    if validate(Σ) == invalid then
        return unsatisfied;
    else if validate(Σ) == conflict then
        backtrack();
        exit;
    end
    simplify(Σ);
    if validate(Σ) == invalid then
        return unsatisfied;
    else if validate(Σ) == conflict then
        backtrack();
        exit;
    end
    literal = decide(Σ);
    split(literal);
end;
```

Algorithm 1: DP pseudo-code. ($\mathbf{\Sigma}$ is the set of clauses in clausal normal form (CNF))

The function $validate()$ asserts if the set of clauses $\mathbf{\Sigma}$ is a valid CNF propositional formula by checking for empty clauses, conflicting unit clauses. It returns $invalid$ if we cannot backtrack, and returns $conflict$ if we can. $simplify()$ goes through the clauses in the current set $\mathbf{\Sigma}$, removing all clauses which are already satisfied, i.e. one of their elements is True; assigns a True value to literals in unit clauses (as these clauses can only be satisfied in this way), and checks the literals within clauses, removing the ones that are assigned a False value. Subsequently, $deduce()$ checks for conflicts by searching for unit clauses that are inconsistent (eg. $(P) \wedge (\neg \mathrm{P})$). In case of a conflict, the $backtrack()$ function is used to abort the current iteration, and return to the decision node to assign a different value. The $assign\_values()$ function is used to assign values to the selected literals. It does that based on the position of the literal. If it's negated, the literal is assigned a *True* value, if it's not, it is assigned a *False* value. The $decide()$ function selects a literal from the set of unassigned literals of our problem using the branching heuristic currently implemented, and the $split()$ function assigns a value to it (true or false). For the standard DP algorithm, our $decide()$ function decides which of the unassigned literals to split randomly. First a *False* value is assigned to the literal, and in case of a conflict, we backtrack and assign a *True* value. Finally, we return satisfied if all the clauses of the formula are satisfied.

DP algorithm with MOM's heuristic:

The MOM heuristic is a widely used deterministic heuristic used to decide on which literal to split; that is in the $decide()$ function of our DP architecture. For this heuristic, the smallest non-satisfied clauses are isolated and the constituent literal that maximizes the following function is selected

$$\hat{l} = \underset{l}{argmax}\Big((f(l) + f(\neg l)) \times 2^k + f(l) * f(\neg l)\Big)$$

where l is the literal, $f(l)$ is the frequency of the literal, and k is a tunable parameter [Johnson and Trick1996]. We experimentally found k=1 to be the most effective value for the parameter.

This function essentially underpins the branching strategy for selecting the next literal, by assigning a priority score to each [Marques-Silva1999b]. Intuitively the literal that maximizes this function is the literal whose value assignment would result in the most unit clauses that will get removed by the next wave of simplifications. However, it does not take into account further steps ahead; that is, downstream implications.

DP algorithm with TK1 heuristic:

This heuristic is also used in the $decide()$ function of our DP architecture to select the next literal to be split. TK1 however does not only go through the clauses of minimum size, but instead goes through all clauses and picks the literal that minimizes the following function

$$\hat{l} = \underset{l}{argmin}\Big(\alpha \times cs(l) - \beta \times cf(l) - (1 - \beta - \alpha) \times MOM\Big)$$

where l is the literal, $cs(l)$ is the sum of the sizes of the clauses containing l, $cf(l)$ is the number of clauses containing l, α and β are tunable parameters, and MOM is the score for that literal obtained using the MOM function.

To tune the parameters α and β we looped through all values for $alpha$ and $beta$ between 0 and 1 for intervals of 0.05, with the constraint that $\alpha + \beta = 1$, and kept the optimal values. These turned out to be $\alpha = 0.45$, $\beta = 0.5$, thus the contribution of the MOM score component was diminished.

Our priority function is a mixture of the MOM's function but also incorporates information about how important the selected literal is in terms of clause implications, and a penalty with respect to size of the clauses in which is involved, giving preference to lower cumulative clause size. Thus we are not only restricted to the clauses of minimum size but we capture literals that are more constrained than the ones captured for regular MOM's heuristic. Intuitively, our function accomplishes this by minimizing for literals that belong to large clauses and maximizing for number of clauses affected. Therefore, we would expect that branching on these literals would cause in greater simplifications in our propositional formula, further constraining the search and ultimately making it more efficient.

III. EXPERIMENTS

Task and Data

The aim of this study was to investigate the effect of different branching heuristics extensions on the search dynamics for the DP algorithm. We addressed this question by assessing algorithm performance on Sudoku puzzles.

Sudoku, is a logic-based, number-placement puzzle. Given a partially filled square grid, typically a 9x9 grid composed of nine 3x3 square subgrids, the objective is to fill the grid with digits to complete the puzzle, subject to the constraints that each row, each column, and each of the subgrids contain each of the digits from 1-9 only once and at least once. Typically Sudoku puzzles have one solution, but some may have more than one. In general, an $n^2 \times n^2$ Sudoku grid of $n \times n$ sized subgrids is considered as an NP-complete computational problem. As the n and as the number of clues increase, combinatorial explosion occurs, resulting in the computational solution time to increase exponentially in the worst case [Weber2005].

We encode the Sudoku puzzles using propositional logic in clausal normal form (CNF). The problem then becomes a satisfiability problem that is solved by finding a configuration of truth value assignments to the literals that make the formula true. The formula is satisfied only if the given Sudoku puzzle has a solution.

For this experiment we used two datasets of 9x9 Sudoku puzzles. A set of 1000 easy Sudoku puzzles, each having contain 21 or more hints, and a set of 35 very hard Sudoku puzzles that each have 17 hints. Additionally, for each Sudoku game we are using the Sudoku rules encoded in DIMACS format.

Evaluation

To evaluate algorithm performance, we looked at the number of splits over all runs. This indicates the total number of assignment decisions made by each algorithm on each instance/game, which serves as a proxy for the number of branches explored; that is, amount of search conducted by the algorithm on each instance. Therefore this metric captures the how much each heuristic constrains the search is [Marques-Silva1999a].

We also collect the number of backtracks for each instance. These represent the number of times each algorithm assigned a value to the selected literal that lead to a dead end. Therefore we compute the ratio of successful splits (we define these are splits that did not result in a backtrack) by subtracting the number of backtracks from the number of splits and dividing by the number of splits. This can be considered as an indicator of the quality of the splits, although our heuristics do not explicitly control the value assignment during splitting.

Statistics

We tested our data for normality using the D'Agnostino-Pearson omnibus normality tests which showed that none of the groups (RAND, MOM's and TK1) followed a normal distribution. Therefore, in order to obtain statistical significance in our comparisons we used a Kruskal-Wallis test, and a Dunn's Multiple Comparison Test on the results.

The error bars in the bar charts represent standard error. Statistical significance is indicated on graphs using standard conventions: n.s. (non-significant): $p > 0.05, * = p \leq 0.05, ** = p \leq 0.01, *** = p \leq 0.001$. The experiments for the RAND heuristic were repeated in triplicate because of its stochastic properties, and for MOM's and TK1 heuristic were performed only once because these are deterministic.

The analyses were performed using Prism 5 (GraphPad Software). The significance level was set at $p = 0.05$.

IV. RESULTS AND ANALYSIS

MOM's heuristic and TK1 improved performance by constraining the search

We began by comparing the performance of the control heuristic (RAND), MOM's heuristic and TK1 heuristic on all 1000 easy Sudoku problems. We detected a statistically significant difference in performance between the three groups using a Kruskal-Wallis test (H = 26.87, $p < 0.0001$). A Dunn's Multiple Comparison test showed that both MOM's (4.10 ± 6.24 , $p < 0.0001$) and TK1 (3.33 ± 4.97, $p < 0.0001$) significantly outperformed the control heuristic (13.74 ± 34.00) by a approximatly 3-fold decrease in splits for MOM and approximately a 4-fold decrease in splits for TK1. However, the difference between MOM and TK1 was not found statistically significant.

Subsequently we made the same comparison using all 35 hard Sudoku puzzles to get a better resolution of the differences between the performance of the DP algorithm using the three branching heuristics. We found the same trend between the three. The three algorithms were significantly different (H = 39.54, $p < 0.0001$), with both MOM (15.71 ± 15.63) and TK1 (15.09 ± 24.72) approximately 90 times better than RAND (917.4 ± 1478), as expected. This time the difference between the two algorithms was not as pronounced, and again was not significantly different. Results are displayed in Table I, and graphically in Figure 1.

The difference between the performance of the RAND heuristic was expected, and verified that our branching heuristics do indeed exert beneficial control over the branching. However, we had anticipated that our heuristic would significantly outperform MOM's heuristic because by incorporating more information about the frequency and cumulative size of the clauses containing the literals, we would capture literals which are overall more distributed and constrained than just selecting the literals with the

maximum occurrences in clauses of minimum sizes. We suggest that the improvement observed in performance is limited because Sudokus have a larger proportion of binary clauses than others, favouring MOM's heuristic over TK1 [Marques-Silva1999a].

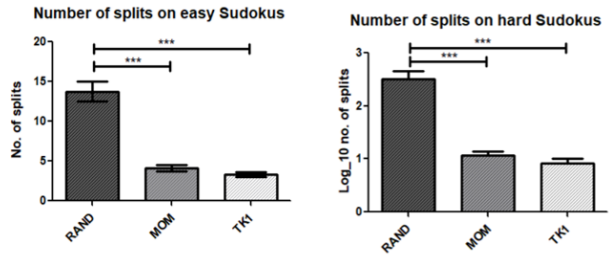

Fig. 1: Comparing the number of splits of our heuristics on easy Sudokus (left) and hard Sudokus (right). For the hard Sudokus we used the log_{10} number of splits because the difference between RAND and MOM and TK1 was enormous

	RAND	MOM's	TK1
Easy	13.74 ± 34.00	4.10 ± 6.24	3.33 ± 4.97
Hard	917.4 ± 1478	15.71 ± 15.63	15.09 ± 24.72

TABLE I: Number of splits $\pm$ standard Deviation to 2 d.p.

MOM's heuristic and TK1 implicitly improve split quality

To further dissect the search dynamics when using the different branching heuristics, we used the ratio between splits that did not result to backtracks, and all splits to assess the quality of the splits of each algorithm, as described in the training section. We found that for the easy Sudoku puzzles, there was no difference between the models (H = 3.089, p = 0.2134), but we did observe an increase of approximately 15% of successful splits in the hard Sudoku puzzles when using MOM's (0.70 ± 0.16, $p < 0.0001$) and TK1 heuristic (0.70 ± 0.18, $p < 0.0001$) as compared to the RAND heuristic (0.55 ± 0.12). Results are displayed in Table II, and graphically in Figure 2.

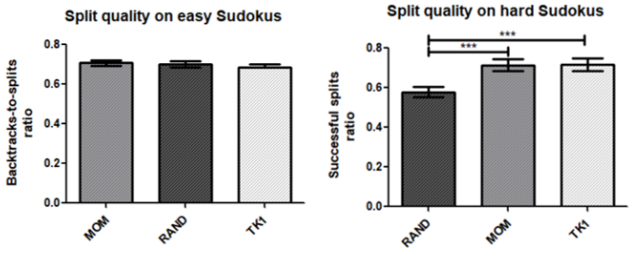

Fig. 2: Comparing the ratio of successful splits of our heuristics on easy Sudokus (left) and hard Sudokus (right).

	RAND	MOM's	TK1
Easy	0.70 ± 0.19	0.71 ± 0.20	0.68 ± 0.20
Hard	0.55 ± 0.12	0.70 ± 0.16	0.70 ± 0.18

TABLE II: Ratio of successful splits with all splits $\pm$ standard Deviation to 2 d.p.

Our results indicate that MOM and TK1 implicitly bias towards selecting literals that their true value is False, therefore after splitting there is no backtrack, since we always test the False value before assigning a True value to the selected literal.

Limitations

The main limitation of our study is that we only looked at a specific SAT problem, therefore our results are not very representative on how our solver can generalize to other SAT problems. As mentioned earlier, since the most literals in a Sudoku propositional problem have a False value in order to satisfy the formula, our policy of first assigning a False value to the selected literal when splitting strongly favours search constriction. Also, the fact that Sudoku propositional formulas have a larger proportion of binary clauses than others, and have no pure literals; properties that may not be shared by the majority of SAT problems.

Another limitation is that it is a very hard task to determine the exact relative difficulty between Sudoku puzzles. This depends on many variables such as the number of hints given, the size of the grid and the number of solutions, for which there does not exist an accurate metric to our knowledge. This can be inferred by the relatively large variance in our results from the deterministic branching heuristics for both the easy and hard Sudokus, signifying that there is a corresponding high variance in difficulty within our two Sudoku sets. This may serve as an indicator that Sudokus may not be the most transparent SAT problem to use as a testing ground for comparing our algorithms.

Future Work

A natural extension of our study would be to apply our algorithms to different classes of well known NP-complete problems, such as vertex cover, graph k -colorability, and Hamiltonian path.

V. Conclusion

In the present study our aim was to analyze the effect of branching heuristics to the search dynamics of the well known DP algorithm. We introduced a new branching heuristic, similar to MOM's heuristic, that aims to identify the most constrained literal in a propositional formula in order to constrict the search.

To analyze the performance of our heuristic, we used two sets of Sudoku puzzles that vary on difficulty, based on the number of hints given, and compared it to the performance of the default DP algorithm with the RAND heuristic, and the DP algorithm with the basic MOM's heuristic. Our results showed that MOM's heuristic and TK1 outperformed the RAND by restricting the search; that is, pruning the search tree based on dynamic feedback during the search. However, given our results, even though we did detect a small improvement when using TK1 compared MOM's

heuristic, it was not found to be significant, therefore we cannot accept our hypothesis that TK1 yields improved constraining of the search. Given the distinct properties of Sudoku propositional formulas, such as the lack of pure literals and the high proportion of binary clauses indicate that our heuristic should be tested on different SAT problems to fairly examine it's effect on the search dynamics.

We conclude that the properties of the task at hand are important in choosing a heuristic. For optimal results, the heuristic should take advantage of such properties, but tailoring the heuristics used on our task would mean that we reduce the algorithms' ability to generalize.

REFERENCES

[Freeman1995] Freeman, J. W. (1995). Improvements to propositional satisfiability search algorithms.

[Johnson and Trick1996] Johnson, D. S. and Trick, M. A. (1996). *Cliques, coloring, and satisfiability: second DIMACS implementation challenge, October 11-13, 1993*, volume 26. American Mathematical Soc.

[Li and Anbulagan2000] Li, C.-M. and Anbulagan, L. (2000). Heuristics based on unit propagation for satisfiability problems. 1.

[Lynce and Ouaknine2006] Lynce, I. and Ouaknine, J. (2006). Sudoku as a sat problem. In *PROCEEDINGS OF THE 9 TH INTERNATIONAL SYMPOSIUM ON ARTIFICIAL INTELLIGENCE AND MATHEMATICS, AIMATH 2006, FORT LAUDERDALE*. Springer.

[Marques-Silva1999a] Marques-Silva, J. (1999a). The impact of branching heuristics in propositional satisfiability algorithms. In Barahona, P. and Alferes, J. J., editors, *Progress in Artificial Intelligence*, pages 62–74, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Marques-Silva1999b] Marques-Silva, J. (1999b). The impact of branching heuristics in propositional satisfiability algorithms. In *EPIA*.

[Weber2005] Weber, T. (2005). A sat-based sudoku solver. In *12 th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR 2005*, pages 11–15.

[Zuim et al.2008] Zuim, R., Sousa, J., and Jr, C. (2008). Decision heuristic for davis putnam, loveland and logemann algorithm satisfiability solving based on cube subtraction. *Computers Digital Techniques, IET*, 2:30 – 39.