Adapting a SAT branching heuristic for Sudoku puzzles

Marvin Lau UVA: 12364282 VU: 2580713
Bobbie van Gorp UVA: 11161108 VU: 2581322

***Abstract*— In this research we aim to implement a SAT solver and design a branching heuristic that can be used to efficiently solve Sudoku puzzles encoded as SAT problems. Our proposed heuristic is compared to the existing Jeroslow-Wang heuristic by looking at the number of necessary backtracks on Sudoku problems. We find that splitting on random literals performs reasonably well, while the Jeroslow-Wang heuristic performs very poorly. Our proposed heuristic shows a smaller number of necessary backtracks that is statistically significant, and both the number of splits and runtime are better as well.**

I. Introduction

Solving a (9 by 9) Sudoku is not an easy task for some human beings. Because of the enormous search space - there are 9 possible numbers to fill in for every box that is not given - even a computer can not trivially solve a Sudoku. In this research, the Sudoku problem is translated into a boolean satisfiabiliy (SAT) problem in order to efficiently solve Sudoku puzzles. Although SAT is the original NP-complete problem, meaning that there are no known algorithms that have a polynomial worst case complexity, modern SAT solvers can solve most SAT problems very quickly [1]. For this project we implemented our own SAT solver based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [2] and different branching heuristics to optimize (the number of backtracks in) the process. The first heuristic we used is the relatively early Jeroslow-Wang heuristic (1990) [3], to which we will refer to as JW. To adapt the solver to the problem of Sudoku, we propose our our own heuristic, minimial clauses of minimal Length (MCML), which focuses on variables that occur in a small number of clauses that are of minimal length. A probabilistic version of our proposed heuristic was implemented as well. We used the implemented algorithm and heuristics to test the following hypothesis: *DPLL with our proposed heuristics will outperform DPLL without heuristic and DPLL with the existing JW heuristic on Sudoku puzzles translated into SAT problems in terms of the number of backtracks.* With this hypothesis we intend to find out whether our own heuristic specifically designed for the Sudoku problem will beat a proven heuristic designed for general SAT problems. By comparing the heuristics with many different metrics we hope to gain a deeper understanding of properties of heuristics for SAT-translated Sudoku puzzles.

II. Algorithm design

Our algorithm was designed to process a SAT problem given in DIMACS conjunctive normal form (CNF) format. The general DPLL procedure can be implemented in a number of different, specific manners. Our implementation is a recursive algorithm and a number of design decisions will be discussed here. Four lookup tables are used in order to find related literals and clauses efficiently and to keep track of assignments and changes. One lookup table contains unsatisfied (simplified) clauses mapped to IDs. A second lookup keeps track of unassigned clause IDs for all unassigned literals. This table also make it easier to check for pure literals and unit clauses. Together these two tables can be used to efficiently update new assignments. The third lookup table contains all the literals that have an assignment and the last table stores new assignment changes that need to be checked. Since tautologies are either present or not present from the beginning, i.e. they do not emerge during the procedure, our implementation only tests for the presence of tautologies once, after the unit propagation mentioned above.

Although our algorithm can solve general SAT problems, the primary objective of our implementation is solving Sudoku puzzles encoded as SAT problems. Therefore, the algorithm is designed to optimally make use of the properties of a general DIMACS CNF encoded Sudoku puzzle. The general rules are the same for any (normal) Sudoku and these consist of a large number of lines that describe that a number can only occur once in each row, column and square. Each puzzle instance also contains a small number of unit clauses for the cells that are given. The presence of unit clauses from the beginning means that a large number of variables can be assigned a value by simplifying the clauses alone: the unit clause forces the variable in that clause to have a certain value and by assigning this value to the variable, a number of other clauses can be simplified or satisfied. If clauses are satisfied, the clause ID will be removed from the lookup table and every literal point to that ID will also be updated. If a clause is not satisfied, it can be simplified by removing the literal from the clause and update the tables accordingly. Clauses of the opposite polarity of the literal are also updated in a similar manner. All of this will likely result in new unit clauses, leading to new forced variable assignments. This unit

propagation process may repeat a large number of times. Because of this behavior, the processing helper function, which checks for contradictions in new changes and process the assignments if approved, was implemented recursively as well. In this manner, a single call of the processor function will solve a large part of the Sudoku by unit propagation alone, before the first split has to be made. If meanwhile any contradictions have been found in this initial unit propagation, it means that the initial Sudoku that was given is unsatisfiable. Because the processing helper function is called a very large number of times, it is important that it is implemented efficiently. In our case, the processor function call requires an argument that specifies which variables were assigned a (different) value in the previous step. Along with the lookup table that contains information on which clauses each variable occurs in, the processing function only has to process the clauses in which changes actually happen and can ignore all unrelated clauses.

If the main DPLL function cannot find any new changes to simplify, then it will pick a variable from a list of unassigned variables and split using a branching heuristic. A recursive call follows, which passes on copies of the lookup tables so that the original lookup tables can be used if backtracking is needed to assign the opposite value for the literal that was assigned in the split.

While the recursive call of the main DPLL procedure itself requires full copies of the necessary objects to be able to successfully backtrack, for the function call to the processor function a pointer to same tables suffices as any changes due to unit propagation are forced moves. This reduction in the number of copies required saves memory and drastically improves runtime of the algorithm.

Lastly a note on the implementation: The algorithm is implemented in Python because the relative simplicity of this language allows us to concentrate on improving the problem solver instead of spending a lot of time on the implementing and debugging the solver. It can be run with the following command: ./SAT -Sn input_file_name.

III. Heuristics

In this section, a small overview is given of heuristics that are used in the experiment and elaborating on their characteristics.

A. Jeroslow-Wang One and two sided

The Jeroslow-Wang (JW) heuristic has a one (JW1) and two (JW2) sided approach [3]. Each literal has a score that is computed in relation to the N clauses that it occurs in: $\sum_{i=1}^{N} 2^{-|c_i|}$. The literal with the highest score will then be picked to split on. The general intuition behind JW is to give literals that occur in small clauses a higher value and since it focuses on summation over all the clauses, more clauses also results in a higher score. One could say that it focuses on literals that occur in maximal clauses of minimal length. The approach mentioned above is for the one-sided approach as the truth value assignments is based on the polarity of the literal with the highest score. The two-sided approach combines the scores of both polarities for each variable and uses that combined score to determine the variable with the highest score. The truth value assignment of that literal is determined based on the individual scores of each polarity, where the polarity with the higher score will be used as truth value assignment.

B. MCML

Our proposed heuristics MCML1 and MCML2 are adaptations of JW1 and JW2. We suggest to focus on literals with minimal clauses of minimal length. It has a similar intuition as JW that small clauses have a higher weight. This is desired as small clauses can lead to unit clauses and propagation. However the difference with our method is to focus on literals with minimal clauses. The computation of the score for each literal is the summation of the length of clauses that the literal occurs in: $\sum_{i=1}^{N} |c_i|$. The literal with the lowest score is then picked with its corresponding polarity as truth assignment for the variable and ties are randomly decided. In Sudoku, there are in general more negative literals occurrences than positive literal occurrences. By prioritizing literals that occur in the minimal number of clauses, positive literal have a higher chance to be picked as they are less frequent in the set of clauses. Positive assignments are preferred over negative assignments because they are more useful in Sudoku. Positive assignments lead to more unit propagation: a single positive assignment forces a lot of variables to be assigned false. Positive assignments are also more important in contradictions. If two positive literals concerning the same cell in Sudoku are true, it is a contradiction. In the case where two negative assignments concerning the same cell set to true, it tells you that a cell does not have e.g. the values 2 and 3. This does not immediately lead to contradictions. Thus by assigning positive literals to true, the algorithm cuts to the core of the problem more quickly and avoids deviating too long in a search space that will eventually lead to long a backtrack.

C. PMCML

This heuristic is an probabilistic approach of the MCML. Where the MCML is greedy and always picks the literal with the lowest score, the PMCML assigns probabilities to each of the literal according to the same intuition of minimal clauses of minimal length. The

probability of picking a literal is based on the score from before: $score_{literal} = \sum_{i=1}^{N} |c_i|$. As each literal has this score, it is normalized in the following way such that the lowest score has the highest probability:

$$p(literal) = \frac{2^{-score_{literal}}}{\sum_{i=1}^{N} 2^{-score_i}}$$

Then a literal is picked according to these probabilities.

IV. Experimental Design

In order to test our hypothesis, we designed experiments in which we compare the performance of the different heuristics tested on different Sudokus from different datasets. For each heuristic, all Sudokus were run multiple times with different seeds in order to get reliable results. Although we suspected to be most interested in the number of backtracks because our hypothesis is formulated in terms of this measure, we included a large number of other performance measures to gather as much data as possible. The following metrics were collected: Runtime, number of splits, number of backtracks, number of positive assignments, number of negative assignments, positive to negative assignments ratio, number of clauses learned, number of clauses at start, number of variables at start, number of initial boxes filled, number of clauses before loop, number of variables before loop and number of variables assigned before loop. After looking at the results we decided that the most interesting metrics were number of backtracks, number of splits and runtime and we did not use the other collected measures. We ran the algorithms with each heuristic on the 35 'damnhard' Sudokus with 3 runs per Sudoku, on the 95 'top95' Sudokus with 5 runs per Sudoku and on the 2365 'top2365' Sudokus with 5 runs per Sudoku. Then we tested for statistical significance using the student's t-test.

V. Results & Discussion

In this section, the results of each case are presented and discussed. First, the mean results of all heuristics on damnhard Sudokus can be seen in Table I. It seems that the difference in results can be divided into three categories; Random, JW and our heuristics. From the table it can be observed that category JW has the highest runtime, amount of splits and amount of backtracks. Random has reasonable results, but our proposed heuristics seems to be faster and they require 10 times less splits and backtracks. All versions of our heuristics show similar results when comparing them with each other. Box plots showing one heuristic of each category also support the difference between the three, which can be seen in Figures 1, 2 and 3. Each boxplot has an outlier, but for MCML there are multiple outliers. Futhermore, the difference in results between JW1, JW2 and the rest is quite noticeable and is supported by a t-test of which the results can be seen in Table II. The difference between Random and our heuristics is significant as well.

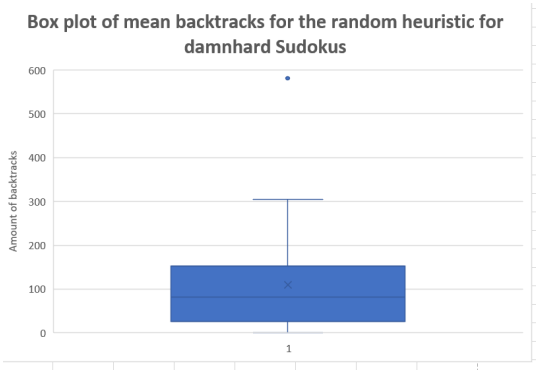

Fig. 1: Box plot of Random on damnhard

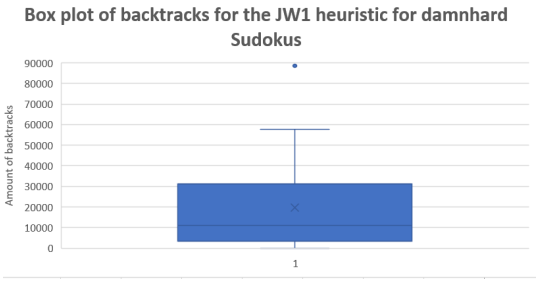

Fig. 2: Box plot of category JW on damnhard

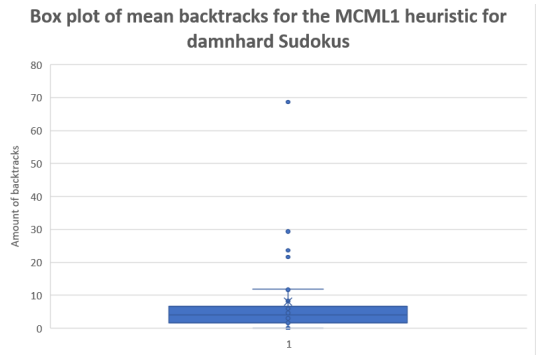

Fig. 3: Box plots of category MCML on damnhard

Since the results of JW1 and JW2 were so poor, they have been excluded from the top95 and top2365 cases, also because these were computationally more expensive. The results of the top95 Sudokus are similar to the damnhard case. Random has reasonable results, but not as good as results of our heuristics. This difference proves to be significant when using a t-test. An overview of the average number of backtracks for all heuristics

can be seen in Figure 4 in which this difference can be observed. There are slightly more outliers, which can be due to the increase in the number of puzzles. Moving on to the top2365 case, an overview of mean results of number of backtracks can be seen in III, where again it can be observed that Random has higher mean runtime, number of splits and number of backtracks compared to our heuristics. However, it must be noted that the difference between Random and the proposed heuristics has become smaller. Looking at the box plot of all heuristics in Figure 5, it can be observed there are many more outliers as the amount of Sudokus puzzles has increased with almost 25 times. Random actually has many more outliers up to 1000 and one additional outlier around the range of 1500 backtracks. The box plot axes is capped to 200 for clear visualization. Our proposed heuristics have no outliers outside this range. Next, the results of statistical test for the top2365 case can be seen in Table IV. Although it seems that the results are similar to the damnhard and top95 case, there is one significant difference for the PMCML1 heuristic. The difference between PMCML1 and any other of our own heuristic turns out to be significant. Looking at the mean results in Table III, of our heuristics, the PMCML1 was the only heuristic that had an average of almost 13 backtracks, while the others were around the range of 11 backtracks. This difference seems small at first, but is significant as the mean is computed from a large number of puzzles and runs.

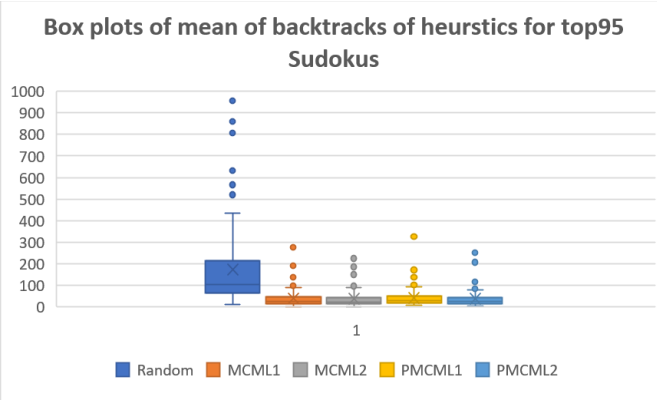

Fig. 4: Box plots of all heuristics on top95

From the results, the performance of heuristics could be ranked in three categories, which all had an noticeable difference that was significant between them. JW1 and JW2 performed significantly worse than other heuristics. The JW heuristic focuses on variables that occur in a large number of clauses that have minimal length. Due to this characteristic it will mainly split on negative literals as there are many more negative literal occurrences in Sudoku. The reason that our heuristics adapts well to Sudoku might be because it does the

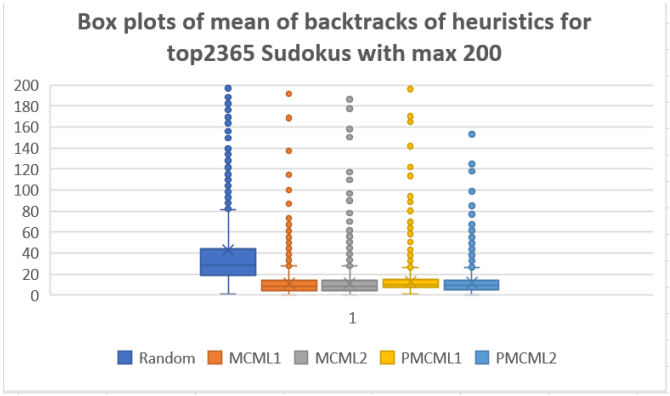

Fig. 5: Box plots of all heuristics on top2365 for all mean results of backtracks capped at 200

opposite. Its core is to focus on positive assignments and unit propagation's, which is achieved by choosing the literal with minimal clauses of minimal length and unit propagation. Because of this, it may detect early contradictions as described in section III, resulting in a more consistent performance. This could be the explanation for the small number of outliers and the shorter distance between first and third quartile, which can be seen in the boxplots of Figures 4,5,3. To continue, the four versions of our proposed heuristic outperform the random heuristic, although its performance is reasonable well. Still, since it is stochastic, the random performance cannot always get consistent results, leading to many outliers and a greater distance between min and max. PMCML1 and PMCL2 also introduce a stochastic element with probabilities, but these still perform well. It could be the case that the probability difference is quite high as it uses $2^{-|score|}$ as numerator and sum of this for each literal as the denominator, which can exponentially differ with difference in length. Thus in most cases it will choose the literal with the lowest score and act similarly to the MCML heuristic. However, there was one significant difference between the results of PMCML1 and the other proposed heuristics in the top2365 case. It could be the case that the stochastic element in so many cases caused the difference to be significant. The reason why this does not apply for PMCML2 could be that the contribution of the opposite polarity counteracts it. Lastly, the difference between one and two sided approach in each heuristic is not significant, except for the PMCML1 case as described previously.

VI. Conclusion

Apart from implementing a SAT solver capable of solving Sudokus, we also managed to successfully implement different versions of both JW and our proposed heuristic. We can conclude from the results that our

Heuristic	Runtime in s	# Splits	# Backtracks
Random	0,23004	114,057	109,838
JW1	48,1317	19653,5	19626,4
JW2	37,7035	15696,8	15667,8
MCML1	0,06776	10,219	8,04762
MCML2	0,06046	10,2857	8,06667
PMCML1	0,07198	11,6952	9,62857
PMCML2	0,07064	11,6095	9,72381

TABLE I: Averages metrics across all Sudokus and runs from damnhard

proposed heuristic does indeed significantly decrease the number of backtracks necessary to solve a SAT encoded Sudoku puzzle, compared to both Random DPLL and DPLL with the JW heuristic. Apart from the number of backtracks, the runtime of the algorithm and the number of splits required to reach a solution are lower than the Random and JW as well. The JW heuristic got the worst results, while our heuristics is significantly greater than the other heuristics. In addition, it is more consistent as supported by the distance between the first and third quartile and outliers. In general, the probabilistic approach of our heuristic does not seem to make any significant differences. Further research could compare the performance of the proposed MCML heuristic with existing heuristics on SAT problems other than Sudoku to see if there will then be a significant difference still.

Heuristic pair	T-stat	± Two tail critical value	Reject H_0
Random - JW1	-5,5383	2,0322	Yes
Random - JW2	-4,8935	2,0322	Yes
Random - MCML1	5,0918	2,0301	Yes
Random - MCML2	5,0841	2,0301	Yes
Random - PMCML1	5,0092	2,0301	Yes
Random - PMCML2	4,9352	2,02619	Yes
JW1 - JW2	0,07064	1,99601	No
JW1 - MCML1	5,5673	2,03224	Yes
JW1 - MCML2	5,5673	2,03224	Yes
JW1 - PMCML1	5,5668	2,03224	Yes
JW1 - PMCML2	5,5668	2,03224	Yes
JW2 - MCML1	4,9305	2,03224	Yes
JW2 - MCML2	4,9305	2,03224	Yes
JW2 - PMCML1	4,9300	2,03224	Yes
JW2 - PMCML2	4,9299	2,03224	Yes
MCML1 - MCML2	-0,0059	1,99547	No
MCML1 - PMCML1	-0,5006	1,99547	No
MCML1 - PMCML2	-0,3633	2,00665	No
MCML2 - PMCML1	-0,4705	1,99547	No
MCML2 - PMCML2	-0,3506	2,00404	No
PMCML1 - PMCML2	-0,0204	2,00488	No

TABLE II: Statistical T-test results of damnhard Sudokus

Heuristic	Runtime in s	# Splits	# Backtracks
Random	0,092477	46,70376	42,83349
MCML1	0,082988	14,5266	11,60930
MCML2	0,067534	14,4323	11,48237
PMCML1	0,079803	15,47958	12,97581
PMCML2	0,073529	14,58841	11,72533

TABLE III: Averages metrics across all Sudokus and runs from top2365

Heuristic pair	T-stat	± Two tail critical value	Reject H_0
Random - MCML1	21,739	1,96088	Yes
Random - MCML2	21,8202	1,96088	Yes
Random - PMCML1	20,9264	1,96091	Yes
Random - PMCML2	21,7207	1,96089	Yes
MCML1 - MCML2	0,29462	1,96047	No
MCML1 - PMCML1	-3,44881	1,96048	Yes
MCML1 - PMCML2	-0,27942	1,964047	No
MCML2 - PMCML1	-3,75292	1,964048	Yes
MCML2 - PMCML2	-0,58282	1,96047	No
PMCML1 - PMCML2	3,28229	1,96047	Yes

TABLE IV: Statistical T-test results of top2365 Sudokus

References

[1] Cook, S. A. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing* (1971), ACM, pp. 151–158.

[2] Davis, M., Logemann, G., and Loveland, D. A machine program for theorem-proving. *Commun. ACM 5*, 7 (July 1962), 394–397.

[3] Jeroslow, R. G., and Wang, J. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence 1*, 1 (Sep 1990), 167–187.