Two Different Heuristics in a SAT Solver Using Davis-Putnam Algorithm

Introduction

A boolean satisfiability (SAT) problem consists of assigning the right truth values to variables of a boolean function, whereby the function can either be satisfiable or unsatisfiable. SAT problem solving plays an important role in field like Electronic Design Automation (EDA) and Artificial Intelligence (AI) (Moskewicz, Madigan, Zhao, Zhang and Malik, 2001). One of the main approaches at solving these problems is using a form of the Davis-Putnam (DP) algorithm, which attempts to find a satisfiable combination of truth assignments of a logical formula, written in clausal normal form (Davis and Putnam, 1960).In this paper an implementation of the DP algorithm, with two different split heuristics, is discussed via solving sudoku problems.

Implementation of the Davis-Putnam Algorithm

This study implements the Davis-Putnam algorithm to solve SAT problems that represent a sudoku puzzle. Present implementation continuously runs through two stages until a solution is found, or until it proves that no solution exists, and can be illustrated according to the following pseudo code.

```
initial_simplify()
dp(clauses, variables):
      while unprocessed variables:
            dp_simplify()
      if no solution:
            split():
                  apply_heuristic()
                  dp(clauses, variables)
```

The initial simplification removes all clauses that contain a tautology. This simplification has to be done only once. Therefore, it is implemented outside the DP-algorithm to reduce unnecessary processing. This is implemented by comparing the number of literals in each clause with the number of variables in the clause. A difference in length suggests the presence of a tautology. After

that, the DP-algorithm checks all clauses for a length of either zero or one. A length of zero indicates that a contradiction exists, and therefore that the problem is unsolvable. Unit clauses have a length of one, and can be removed from the list of clauses, while adding the boolean value of the literal to the list of resolved variables. Every time a variable's truth value is resolved, the value is added to a list called ‘unprocessed’. As long as there are variables in this list, the DP-algorithm will call itself again.

At the start, the DP function iterates over all unprocessed variables, and removes them from any clause that contains an instance of it. Then only the affected clauses are tested for length zero (contradiction) or length one (new unit clause). This process continues until a contradiction is encountered in aforementioned process, or until no size reductions are possible anymore. By merging the variable removal and simplification procedure into the same iteration, the number of total iterations is reduced significantly.

If no solution has been found after applying all possible simplifications, the algorithm performs a split operation by assigning a truth value to one of the unresolved variables and then rerunning the algorithm. In this study two additional split heuristics were implemented, namely a Dynamic Largest Combined Sum (DLCS) split and a Dynamic Largest Individual Sum (DLIS). Eventually the DP-algorithm causes the clause list to shrink. Considering no contradictions or empty clauses are found, the clause list will be empty indicating a solution to the problem.

In the DLCS split-heuristic the frequency of a variable is determined by the occurrence-frequency of it in both literal forms by adding these frequencies up to an absolute total. To the variable with the highest occurrence-frequency, a truth value is assigned based on the highest occurrence-frequency in either its positive or negative literal form. The idea behind the heuristic is to solve as many clauses as possible at once by assigning a truth value to only one variable. In the DLIS split heuristic the frequency of a literal itself is determined. In this split heuristic the unassigned literal with the highest occurrence- frequency gets a truth value assigned to it based on if it is a positive literal (True) or negative literal (False). The difference between the two heuristics is that in the first case it

looks at the total occurrence frequency of a variable and then to the occurrence frequency of the corresponding literal forms, while in the second case it looks at the individual occurrence frequency of a literal itself.

Measurements

In this study two topics will be discussed. First, the necessity of checking for pure literals is discussed. Second, different assign-strategies of the DLCS split heuristics will be discussed. These topic will be discussed via testing en comparing implementation strategies of the DP-algorithm discussed above. Sudoku problems, presented in DIMACS format, are used as input for the SAT solver, together with Sudoku rules written in clausal normal form. For the tests 1000 9x9 Sudoku problems are used. Important to mention is a cut-off of a 10000 DP calls, in these cases the algorithm will stop running due to extreme running times. In the cases where the DP-algorithm uses its normal split procedure, the algorithm will pick the first unassigned literal available, because no random picking is implemented. This due to the lack of reproducibility in testing with random picking.

As stated in the implementation of the DP-algorithm section of this paper the algorithm checks for the presence of pure literals. However, performance gains by checking for pure literals can be questionable, as this procedure is very performance intensive and has to be run every time the DP-algorithm gets called. Therefore a comparison will be made between running the DP-algorithm with and without checking for pure literals. The first hypothesis will therefore be that running the algorithm without checking for pure literals will not result in significantly more computing power, measured in the number of times a literal is searched for in the clause list.

As stated in the implementation section of this paper the DLCS split heuristic picks a variable to assign a truth value to based on the highest occurrence-frequency of a variable. Next the algorithm decides whether it assigns a truth value to either the positive or negative literal form of that variable.. However, taking the Sudoku rules clause list used in this study as an example, the negative literal form of a variable will always have a higher

occurrence-frequency than it's positive literal form. This will result in the DLCS split heuristic always picking the negative literal form and assigning it a `False` boolean value. The same goes for the DLIS split heuristic. The problem with this is that in the case of a 9x9 Sudoku problem there are 729 variables that need to get a truth value assigned to. Of these variables, 81 need to be assigned to `True` and 648 need to assigned to `False`. Therefore if the split heuristic is always going to take the negative literal form and assigning it to `False`, this may result in more computing power then when it searches for the 81 cases it needs to get assigned `True`. Therefore the second hypothesis and third hypothesis of this study will be that the standard implementation of the DLCS split heuristic and the DLIS split heuristic will use significantly more computing power, measured in number of DP-calls, than simply assigning the truth value to the positive literal form of the variable with the highest occurrence-frequency.

Analyses

To compare differences in computing power with and without checking for pure literals 1000 4x4 Sudoku problems were run by the algorithm. Table one shows the descriptives of the results. A significant difference in number of literal searches was found $t(999) = -172$, $p < .001$.

Table 1. *Literal Search*

Algorithm	Mean	Standard Deviation
Normal	22450	773.2
Without Pure Literal Check	23740	902.0

To compare split heuristics 708 9x9 Sudoku problems were run by the algorithm. The algorithm tried to solve each Sudoku with three split heuristics. Each of these split heuristics tried four literal picking strategies, namely it's normal strategy, picking the positive literal, picking the negative literal and a reversed version of it's standard strategy. This way, the algorithm tried to solve each Sudoku 12 times. Computing power was measured by the number of

DP-calls the algorithm made. Table two shows the descriptives of the results. A paired samples t-test showed a significant difference in DP-calls between normal DLCS and positive literal picking with the DLCS , $t(707) = 3278$, $p < .001$. However also a positive difference was found in DP-calls between positive literal picking and negative literal picking of the DLCS, $t(707) = 22699$, $p < .001$. Also, a significant difference was found for DP-calls between normal DLIS and positive literal picking $t(707) = 3141$, $p = 0.002$. No difference however was found between positive and negative literal picking of the DLIS as the results where exactly the same.

Table 2. *Means with Standard deviations in parentheses of DP-calls of all split heuristics*

	Standard Split	DLCS	DLIS
Normal	2498 (4184)	7317 (4395)	7355 (4380)
Negative	3288 (4666)	3196 (4510)	7321 (4395)
Positive	2498 (4198)	7276 (4413)	7321 (4395)
Reversed	3288 (4666)	3347 (4596)	7355 (4380)

Conclusion

In this study comparisons were made using different implementation strategies on a Davis-Putnam algorithm. Results of comparison between running the algorithm with and without checking for pure literals showed a significant difference, indicating more computing power for an algorithm without checking for pure literals. This was not in line with the previous determined hypothesis of this study, which expected the results to be the other way around. There seems to be evidence however for the second and third hypotheses of this study stating a significant difference in picking either positive or negative literal form over the standard implementation of both the DCLS split heuristic and the DLIS split heuristic, as results showed significant differences in both cases. These results may be due to the fact that this study used Sudoku problems to test its hypothesis and the way in which these problems are formatted. However, this can serve as quite a good example of why it is not always useful to implement the DLCS and DLIS split heuristics in its standard form.

Literature

Davis, M., & Putnam, H. (1960). A computing procedure for quantification theory. Journal of the ACM (JACM), 7(3), 201-215.

Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., & Malik, S. (2001, June). Chaff: Engineering an efficient SAT solver. In Proceedings of the 38th annual Design Automation Conference (pp. 530-535). ACM.