Knowledge Representation 2019 - A SAT Solver and its application to sudoku problems

Luisa Ebner
VU Amsterdam
Student Nr. 2638040

Philipp Lintl
University of Amsterdam
Student Nr. 2646385

1 The SAT solver

1.1 Design and Implementation

In this report, we present our implementation of a classical, generally applicable Boolean satisfiability (SAT) solver. Despite the unfavourable property of exponential run time (NP-completeness) in the worst case, SAT solvers have proven to be a valuable and effective general-purpose tool in various application domains. [Gomes et al., 2008] In essence, SAT solvers provide a generic combinatorical reasoning and search platform. Due to the use of increasingly smart implementation techniques and growing hardware capacity, state of the art SAT Solvers are yet incredibly efficient in solving hard, structured problems with over a million variables and several million constraints. [Gomes et al., 2008] As the representational formalism underlying any SAT solver is propositional logic, the user is demanded to primarily translate a given problem into a propositional representation. This translation is commonly referred to as 'SAT encoding' and goes along with a substantial enlargement of the problem representation. Now, our specific SAT project aims at the implementation of a generally valid SAT solver, applicable to any problem representable in propositional logic. The same should thereupon be applied to the specific problem of solving sudokus.

1.1.1 The SAT Encoding

Initially, we were provided with sudoku problems in `.txt` format, where every sudoku was represented as e.g.

....23...8.....4.....1..1..8..6.....7...4.....5..6..2..2.....35...4.....

These 81 symbols can be placed onto a 9×9 sudoku grid, where columns and rows indicate the symbols position in the grid. Whereas the dots indicate yet blank grid fields, the numbers are the sudoku's respective 'givens'. It has become prevalent, to pass SAT problems to SAT solvers in DIMAC format, which is also the case for our special SAT implementation. Therefore, we started off decoding the given `.txt` files into DIMAC format. Applied to the example above, this results in:

152 0
163 0
218 0
...

Every line break indicates a new clause, zeros represent conjunction symbols and spaces stand for disjunction. The number 152 corresponds to a two being placed in the first row and fifth column of the sudoku field.

Additionally, we were given the general sudoku rules, already in DIMAC format. Finally, we merged the givens and the rules as one joint DIMAC file, which comprises a complete SAT problem F in conjunctive normal form (CNF).

1.1.2 The SAT concept

First and foremost, we considered the general purpose programming language Python most suitable for our SAT solver implementation. Our choice was not only reasoned by routine and ease of use, but also by the large standard library and its enabling for clear, elegant list and dictionary operations.

In order to manipulate F in the process of solving boolean constraints, we represent the problem in one list containing lists of tuples. Whereas the cnf is one superior list, every clause is a list of the appearing tuples. As a literal is either a variable or its negation, we chose to depict it as a tuple of the variable name and a boolean. Referring to the example above, the DIMAC variable "-152" is represented as (152, False), as part of a cnf of the form

$$[[(152, True)], [(163, True)], [(218, True)], \ldots]$$

In general, the purpose of our SAT solver is to answer the following:
Given a problem F in CNF, does F have a satisfying assignment and if so, how does it look like exactly?

Thus, our SAT solver is supposed to either state that F has proven unsatisfiable or output the truth assignment dictionary comprising a satisfying assignment of F. This is typically referred to as a *complete solution* for a SAT problem. In the case of sudokus, a complete solution to a given puzzle amounts to a dictionary with exactly 729 elements mappings from variable (as key) to FALSE or TRUE (as value).

As to that, we define an empty dictionary called *thruth assignments*, that is then to be filled in the process of gradual problem solving and output in the case of satisfiablility.

We eventually decided to base our SAT solver on a list approach. In the process of the project, we additionally implemented the SAT problem using the supposedly more efficient dictionaries. However, we were not able to overcome errors within the removing of clauses after truth assignments. As to that, we deliberately decided to follow the initial list approach. We thereby favor reliable and accurate solutions over increased efficiency.

1.1.3 The DPLL algorithm

We realized the SAT solver based on the the DPLL alogorithm as a complete search algorithm for deciding the satisfiability of a CNF-SAT problem and - if possible - finding the satisfying truth assignment.

In a nutshell, our DPLL implementation runs by choosing a literal, assigning to it a truth value, simplifying the clauses and then recursively checking if the simplified cnf-sat problem is satisfiable.

At the beginning of the solving process, we check the cnf for tautologies, which are possibly to be removed.

Before the splitting rule is deployed, the DPLL algorithm always checks for unit clauses. This so called *unit propagation* is a preprocessing step to the central dpll splitting rule. Whenever a clause contains only a single, unassigned literal, this clause can only be satisfied by assigning the boolean value, that makes this literal true. As to that, no assignment choice is necessary. Instead all the clauses, that become true under the assignment from the formula, and all literals that become false from the remaining clauses can be removed. In many cases, this simplification step eases past a considerably large part of the naive search space. Furthermore, it enables for a reduced memory usage.

Another possible simplification step would be the elimination of pure literals as variables, that occur with merely one polarity in the cnf. Pure literals can always be assigned in a way that makes all clauses containing them true. Thus, these clauses do not constrain the search anymore and can be deleted. However, its implementation demanded for frequent looping over the entire cnf list, heavily slowing down the solving process rather than easing it. For the experiments, we thus decided to omit this preprocessing step.

Our overall approach is to incrementally build up simplified cnf lists by dropping unit clauses and the following propagation of truth assignments. Due to the use of list comprehension, we did not copy the initial cnf, gradually removing entries, but create new lists, which guarantee the backtracking in the dpll recursion. The latter is achieved by calling the dpll function whenever a split is made.

The implementation of this splitting rule or branching step can be of various kinds. In the basic solver, a random literal is chosen out of the current cnf. In order to make the search process more efficient, we alternatively implemented three heuristics, determining the literal choice and its boolean value assignment for a smarter branch selection. The same will be presented explicitly in chapter 1.2.

After choosing a truth value for a selected literal, the dpll function is called for the current cnf list, as well as the current truth assignment dictionary. If no conflict arises in the following steps the original formula is satisfiable and the final truth assignment dictionary contains the complete satisfying assignment. If otherwise a conflict arises, the latest recursive check is done assuming the opposite truth value. This may be referred to as a one-step backtrack in the space of partial truth assignments. The same guarantees a complete covering of the search space plus an efficient solving process whenever errors appear in the late course of truth assignments. One might call this a branching procedure, that prunes the search space based on falsified clauses.

Unsatisfiability of a given partial assignment is detected if one clause becomes empty, i.e. if all its variables have been assigned in a way that makes the corresponding literals false. Satisfiability of the formula is detected either when all variables are assigned without generating the empty clause, or, in modern implementations, if all clauses are satisfied. Unsatisfiability of the complete formula

can only be detected after exhaustive search.

1.2 Implemented Heuristics

The key feature of DPLL is the efficient pruning of the search space on the basis of falsified clauses. Since its introduction in the early 1960's, the main improvements to DPLL have been smart branch selection heuristics, extensions like clause learning and randomized restarts, and well-crafted data structures such as lazy implementations and watched literals for fast unit propagation. [Gomes et al., 2008]

Also referred to as the *decision strategy*, the variable selection heuristic is a central part to the efficiency and performance of any SAT solver. It is drawn on, whenever no further simplification steps are possible, but the problem is yet unsolved. Then, the heuristic determines according to which criteria one literal is selected and assigned a truth value. Thereafter, again simplification steps are applied in order to check for satisfiability. We chose three particular heuristics out of a variety of such strategies and compare them to the basic DP algorithm, which chooses a random literal.

1.2.1 DLCS

The DLCS, or Dynamic Largest Combined Sum heuristic, differs in the way that the number of occurrences of a literal within the current unresolved clausal normal form are considered for the assignment of a truth value. In particular, the variable v (in our special problem case e.g. 152) with most occurrences in all clauses both with and without negation, is chosen. So, v with $max\ CP(v) + CN(v)$, whereas $CP(v) = \sum v$ and $CN(v) = \sum -v$ is chosen and set to true, if $CP > CN$ and false vice versa.
It follows the notion that a frequently occurring variable corresponds to a large constraint to the search space and leaves a much smaller search space after allocation, since it affects many cases and should therefore resolve many clauses. Thus, it should form the *quickest way forward*. [Marques-Silva, 1999]

1.2.2 DLIS

The DLIS, or Dynamic Largest Individual Sum is comparable to the DLCS heuristic, but instead of considering the largest total sum, it selects the variable with the largest number of occurrences either positively or negatively. In terms of the aforementioned formulas, v with $max\ \{CP(v), CN(v)\}$. The assigned value corresponds to true, if CP(v) is larger and false vice versa. It coincides with the motivation of the DLCS heuristic, but assumes less conflicts to be encountered, as it presumably almost only appears in one form.

1.2.3 Jeroslow-Wang

The third heuristic we implemented is the Jeroslow-Wang heuristic. [Jeroslow and Wang, 1990] The original publication proposes a one and two-sided approach, whereas the former was found to outperform the other. [Marques-Silva, 1999] We therefore decided to look at the One sided variant. Given a literal l, the deciding value forms as such

$$J(l) = \sum 2^{-|w|}, \tag{1}$$

with w being the length of the clause that is currently summed over and literal l appears in. In the one-sided case, the assignment is given by the *maximum* $J(l)$ value. According to (1) the literal, that appears most in small clauses is chosen. This is beneficial, as it presumably leads to less conflicts due to less other literals being involved. However, while experimenting, choosing the largest value did not result in a noticeable difference to DLCS and DLIS. We therefore additionally consider the *minimum* $J(l)$ value as an alternative heuristic that shall be considered. We reason that the fewer clauses a literal appears in, the less likely it is to run into conflict after a split.

2 Experiments

2.1 Evaluation

There is a number of eligible metrics for this task. The most intuitive, namely runtime, shall only be considered descriptively. The actual value is not representative due to inefficiencies within our code and the dependence on the computing unity. We rather focus on the number of conflicts, that appear and require a backtrack, the number of splits being made and the number of resolved unit clauses.

2.2 Data

A total of 22000 sudokus was provided. We evaluate our hypotheses on 5159 of these from different collections. Temporal delays caused by failed efforts to increase efficiency, prevented us from experimenting on more sudokus. To show that our algorithm can solve all given sudokus, we saved all solutions at `./data/solved_sudokus`. Out of the 5159 selected sudokus, 2936 turned out to

be solvable solely by unit propagation. However, the comparison among heuristics requires splits, which is why we only consider the remaining sudokus.

2.3 Hypothesis 1

The first Hypothesis section deals with the comparison of the applied heuristics. Considering that these were developed to improve performance over the basic version, we assume that all the heuristics lead to significant performance boosts over the basic DP. We assume DLCS and DLIS to yield comparable results, as the cnf of this problem contains primarily negative instances of a variable. Therefore, both formulas should usually split the same literal. The Jeroslow-Wang heuristic is supposed to outperform these two, as it additionally considers the clause lengths. We furtherly assume the minimum version of the Jeroslow-Wang heuristic to significantly boost its maximum alternative. We base this reasoning on the fact that more longer clauses appear and thus a smaller $J(l)$ value indicates the appearance in a lot of long clauses, which leads to large clauses being erased.

2.3.1 Experimental Design

As mentioned above, due to inefficiencies and delays we were only able to consider 5159 sudokus. To compare the heuristics against each other, we simply picked sudokus of each of the given collections and solved them with every heuristic. During experimentation, we found that many sudokus are solved by solely applying unit propagation and thus do not allow for comparison of heuristics. That is why, we initially considered each collection and then only looked at results with at least one conflict. Evaluations are based on the number of splits, conflicts and removed unit clauses.

2.3.2 Experimental Results

At first, we describe the evaluation descriptively. Figure 1 depicts the conflicts, Figure 2 the number of splits. They all evaluate the random heuristic best before JW-min. DLCS, DLIS and JW-max are similarly the worst.

Significance is assessed by a Wilcoxon rank sum tests, as descriptive statistics in the form of histograms indicate not normally distributed data (e.g Figures 5/6). Table1 reveals that in fact the random heuristic yields the least conflicts, whereas JW-min is significantly better than the remaining, which are very similar.

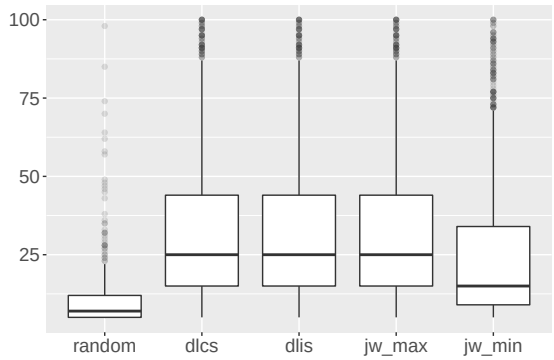

Figure 1: Boxplot of conflicts across heuristics

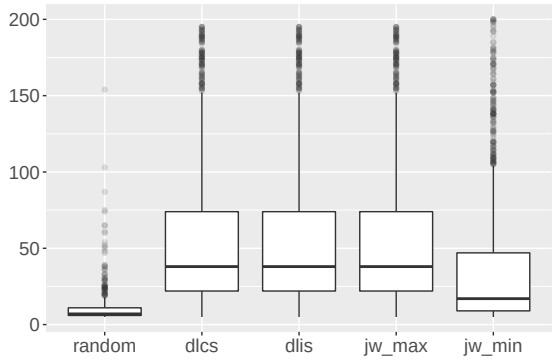

Figure 2: Boxplot of splits across heuristics

2.3.3 Conclusion

Surprisingly, the basic DP algorithm performs best. As suggested in the literature, using the randomized version of both DLCS and DLIS could improve performance over the ordinary versions. for this specific case, simple counting heuristics seem to even worsen the performance, whereas using the minimum version of JW improves it.

2.4 Hypothesis 2

Our second hypothesis concerns the number of given literals beforehand. Particularly, we assume sudokus with fewer givens to be harder, with a linear correspondence between the number of givens and evaluation measure. We base this notion on the fact that less given literals should lead to a larger constraint space. As a result, fewer clauses can be removed, which should result in more splits in total and thus worse evaluation measures.

2.4.1 Experimental Design

We take the same 5759 runs and additionally consider the number of given literals. We then test for a significant relation between the number of givens

Significant differences in number of conflicts
DP vs. DLCS, DLIS, JW_{min}**,** JW_{max} **(*)**
DLCS vs DLIS (**)
DLCS vs. JW_{min} **(***)**
JW_{min} **vs** JW_{max} **(****)**

Table 1: Pairwise Wilcoxon tests on individual evaluations. $(*)$: random outperformed all others $(**)$: not significant $(***)$,$(****)$ JW_{min} significantly better. Significance according to p value « 0.05

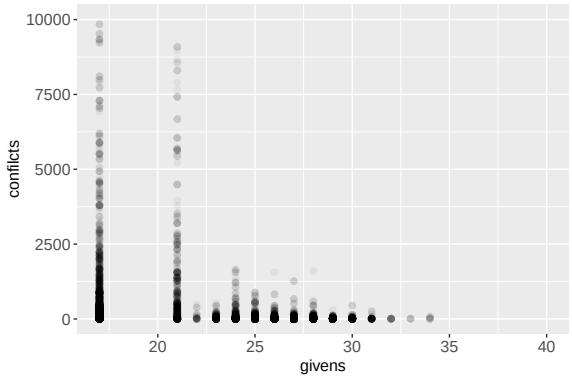

Figure 3: Scatterplot givens vs. conflicts

and the evaluation measure values of the respective sudokus. We conduct a spearman permutation correlation test to assess significance for the random heuristic.

2.4.2 Experimental Results

Plotting both conflicts and removed unit clauses against the given literals (Figure 374) indeed reveal a tendency. For smaller number of givens, more extreme evaluation measures appear. That said, there could be an influence of the number of givens on the evaluation measure at hand.

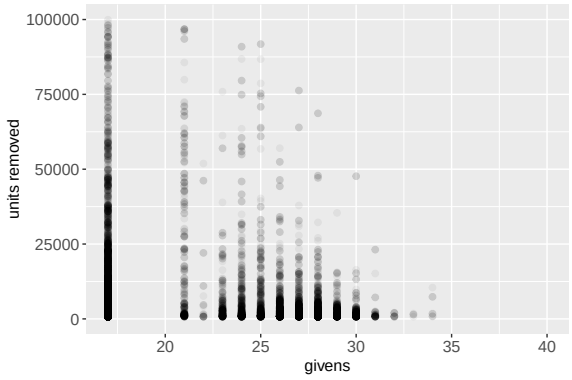

Figure 4: Givens vs. removed unit clauses

We assess significance by a spearman rank correlation test on the givens versus conflicts, removed units and splits. All three tests are highly significant and therefore confirm our assumption: In our sample, less givens lead to worse performance.

2.4.3 Conclusion

Further ideas include filtering the entire dataset of sudokus for a balanced distribution of given literals. Only that way, a representative data subset would have been given. Unfortunately, time constraints again limited our exploration of the 22000 sudokus.

Statistical tests were conducted in R statistical software.

References

C. P. Gomes, H. Kautz, A. Sabharwal, and B. Selman. Satisfiability solvers. *Foundations of Artificial Intelligence*, 3:89–134, 2008.

R. G. Jeroslow and J. Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, pages 167–187, 1990.

J. Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *EPIA*, 1999.

Figures

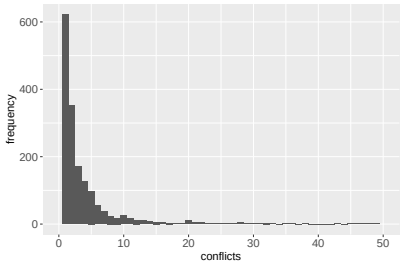

Figure 5: Histogram of conflicts within random

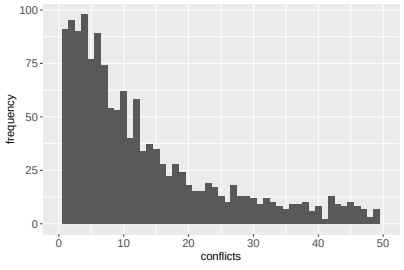

Figure 6: Histogram of conflicts within jw$_{min}$