Investigation of Sudoku's Complexity

Knowledge Representation - Project 1

Ioannis Gatopoulos 2654227 **Filip Knyszewski** 12271136

March 3, 2019

1 Introduction

In 1993, and important discovery by Crawford & Auton [1] reviled the magic ratio 4.26; it turns out that the complexity peak of random 3-SAT problems is very stable across problem sizes. This ratio of clauses to variables gives us a clue that the hard instances are lying around this magic number, as the problems before that tend to be under-constrained and SAT, and after that are over-constrained and UNSAT. Inspired by this discovery, in this report we are going to search for a magic number, which will reveal where one of the most famous SAT problems, the Sudoku, becomes hard. Our hypothesis is that we are expecting to see a Gaussian like distribution over the number of unit clauses (the number of already filled numbers on the grid), where of a particular number the problem becomes more complex. Especially, we are expecting this number to be 17, which for Sudoku is not just like any other numbers; it is the minimum number of clues that a 99 Sudoku puzzle can start with and still yield a unique solution [2]. Like the 3-SAT complexity, We believe that the problem before 17 tents to be more trivial, since the number of possible solutions is getting bigger and the number of restrictions (that the already filled up numbers yield) are less and less, and, controversially, as the number of unit clauses is getting bigger than 17, the number of acceptable chooses and the number of clauses that need to be learned are getting less and less. To investigate this hypothesis, we implemented the Davis Putnam Logemann Loveland (DPLL) algorithm [3], where most of the current state-of-the-art SAT solvers are based on and it is described on section 2. Furthermore, to make the algorithm more efficient, we implemented two decision based heuristics (Jeroslow-Wan and Mom's) and compare their performances on section 3. Lastly, on section 4, we provide the results of the experiment, along with discussion.

2 Design

Presented in 1960 as a procedure for first-order logic, DPLL belongs to the 'Complete algorithms' class of SAT solvers (i.e. can prove both satisfiability and unsatisfiability) and it's basic idea is the (i) simplification and (ii) case splitting of a formulae `F` in CNF, until F is the empty set (`sat`) or contains an empty clause (`unsat`). Besides the functionality of this algorithm, the fist and an a crucial part is the design of the data structure that the clauses of `F` would be stored and modified. An efficient one, could have a huge impact on the running time of the algorithm, but also can give more tools to implement more heuristics effectively.

Data Structure As we mentioned, we need a data structure for storing the clauses and be able to carry the following operations; detect unit clauses, remove learned ones and assign literals with True or False assignment which may give information for other literals. A trivial observation is that we only need to check the clauses that the literal appears, because it is only affect those. With these in mind, as initial structure to the CNF rules, we implemented two dictionaries:

```
rules = { #clause:  {literal_1:  'assign_1', literal_2:  'assign_2' ,...} ...}   and
    literal_dict = { literal_1:  {'assign_1', { clause_1, clause_2 ,...} ...} } .
```

where `#clause` is an integer which gives every clause with a unique id and '`assign_N`' is a string that describes a literal as unassigned, false and true literal with the values '`?`', '`0`' and '`1`' respectively. This unique id will give us the opportunity to remove clauses efficiently, reach easily a clause and modify its literal assignments. The `literal_dict` contains every literal, which is mapped with its assignment and with the clause id that it appears. It is important to mention that this dictionary contains the literals only in their non-negative form. The motivation behind it is that when we learn a literal, we simultaneously lean and its negative form. Therefore, for every literal, we could easily track and modify the clause that the negative also appears, making the process more effective and faster. Moreover, we constructed a new set, `true_literals`, that contains all the assigned literals in their form that they are True ('`1`'). For example, if `a = '1'`, then `a` $\in$ `true_literals`, otherwise, if `a = '0'`, then $\neg$`a` $\in$ `true_literals`.

All of the previous data structures are connected in the following way: first, we fill the `true_literals` set with unit clauses of the formulae (if there are not any, we split). Then, for every literal of this set, using the `literal_dict`, we get the clause ids that it appears and we modify the `rules` dictionary. It is important to note that since the `literal_dict` contains only non-negative forms of literals but `true_literals` not, that we are taking the absolute value of them, and, when is needed with if statements we can get its original form. We continue this process updating these data structures until `F` is the empty set (`sat`) or contains an empty clause (`unsat`). In this way, we (i) avoid to loop over all the classes and (ii) with modify the literal in their negative form in one time, making the overall algorithm more accurate. However, in between there are three important procedures; **simplification**, **split** and **backtracking**.

Simplification As described in DPLL, the simplification step contains the three following rules: tautology elimination, unit propagation and pure literal deletion. To our implementation, after we initially search for tautologies and unit clauses, where in the latter case we fixed them to '`1`', the simplification step takes place when modify the rules dictionary (see previous paragraph). If a true literal appears on a clause, it means that the clause is true, so we remove it. Otherwise, we fill the literal of the clause with '`0`', and we make the following checks; if the number of the unknowns is only one, we want to make that literal true. However, if it already in our truth values as False, we backtrack. We loop until no further rule is applicable which means that the number of clauses did not change.

Split After we checked that the simplification of `F` is no more possible and did not backtrack, we pick the non-negative form of an random unassigned literal, and we assign it to true. To help us with the backtracking, he also created two additional parallel lists; one that contains all the literals that we split with (`split_lit`), and a second one, that contains Boolean values which indicate if we tried to assign the literal with the opposite value (`split_bool`). We choose lists because they are ordered data structures and we can keep a chronological diary of the literals that we split with, and that the index of a literal of the former list, will give us its Boolean value from the second (parallel). Lastly, we save the `rules` before this assignment, in case that it is the wrong choice and we need to backtrack.

Backtracking At this stage, a chronological backtracking is taking place; we loop backwards for the latest literal that we split (using (`split_lit`)) and choose that one that we have not tried to mark it as '`0`' (using (`split_bool`). If such a literal exists, we re-assign it and continue with the simplification step. Otherwise, if we even tried the first (root) literal that we split with both of the possible values ('`1`' and '`0`'), and still one or more clauses are unsatisfiable, it means that this literal can not take either values. So in this case, we conclude that the formula is `unsat`.

3 Heuristics

As we described in the split paragraph, the DPLL algorithm proposes to pick randomly the literal that we are going to split with, giving equal importance all of them. However, selecting a 'good' literal can dramatically reduce the running time of the algorithm. In the effort to find such one, SAT solvers are using heuristics and they are called *decision heuristics*, which they define 'good' in different ways. In this report, we implemented two of the most famous in the literature, the **Jeroslow-Wang** and the **MOMs** heuristic.

3.1 Jeroslow-Wang heuristic

The Jeroslow-Wang (JW) [4] is a score based decision heuristic which is based on the observation that branching on a literal appearing in shorter clauses (say, a 2 literal clause), may produce faster/earlier implications or conflicts than for larger clauses. So for every unassigned literal, JW computes a numeral score that is based on the length of the clauses which it appears in. This score is computed by the following equation

$$J(l) = \sum_{l \in c, c \in F} 2^{-|c|}$$

where l is stand for literal, c for clause, $|c|$ for the length of the clause and F for the hole formula. This gives an exponentially higher weight to literals in shorter clauses. Moreover, because we update the scores of literals as clauses are learned, we made JW a dynamic heuristic. In contrast to static heuristics, the literal selection is influenced by the going of the solving algorithm. Experiments showed that the reason behind the success of JW heuristic is that it creates simpler sub-problems. This, in turn, leads to faster SAT solving [5].

3.2 MOMs heuristic

The Maximum Occurrence's in Clauses of Minimum Size (MOMs) heuristic [6] is based on the number of occurences of literals in the smallest unresolved clauses. The choice of which literal to split on is given by choosing the one that maximizes:

$$(f^*(l) + f^*(l')) \cdot 2^k + f^*(l) \times f^*(l')$$

where $f^*(l)$ is the number of occurrences of literal l in the smallest non-satisfied clauses and k is a tuning parameter. The intuition behind this heuristic is that literals found in the clauses of smallest length will be the most constrained in the whole formula. Due to this, branching on these will maximize the probability of making a correct assignment of remaining open literals.

The implementation of the heuristic is straightforward. Whenever a new literal needs to be chosen, the equation above is calculated for every literal. $f^*(l)$ is calculated by simply iterating over all smallest clauses that contain literal l and counting the occurrences. After calculating the score of each literal the one with maximum score is selected. The parameter k is set to 1.5,

3.3 Experiments and Analysis of heuristics

To assess the performance of the heuristics measured above, the algorithm was ran on two datasets of sudokus, one of average difficulty and another one of harder difficulty. The runs were made, one without using heuristics, another with Jeroslaw-Wang and finally with MOMs heuristic. The results can be found in 1 and 2.

	Classic	Jeroslaw	MOMs
# Splits	5.97±8.48	**2.99** ± 3.66	4.12 ± 5.63
# Backtracks	4.95±8.15	**1.65** ± 3.15	3.10 ± 5.20
Time(s)	0.72±0.88	**0.35** ± 0.31	0.45 ± 0.50

Table 1: Results of running the algorithm with each heuristic on a dataset of 1000 easy sudokus

	Classic	Jeroslaw	MOMs
# Splits	100.46 ± 156.25	**12.23** + 21.82	37 ± 96.02
# Backtracks	98.14 ± 156.33	**9.85** ± 20.95	34.6 ± 95.17
Time(s)	9.30 ± 13.99	**1.26** ± 2.02	3.35 ± 8.58

Table 2: Results of running the algorithm with each heuristic on a dataset of 35 hard sudokus

To evaluate the statistical significance of these results a t-test was run comparing each of the measures. For the easy sudokus it was found that all differences were statistically significant within a 5% confidence interval by a large margin and with a p value below 0.0001. For the hard problems, the statistical significance of the differences is given in 3.

The first interesting outcome, although expected, is the fact that both heuristics do in fact improve performance when compared to the classic no heuristics version of the algorithm. This shows that indeed, making the right

	Splits	Backtracks	Time
Random/Jeroslaw	0.0015	0.0015	0.0013
Random/MOMs	0.0445	0.0439	0.0357
Jeroslaw/Moms	0.1413	0.1374	0.1638

Table 3: Results of t-test on the differences between the three implementations

choice for which literal to split on, has a major impact on the performance of the SAT solver. In the case of easy sudokus, using MOMs heuristic led to a reduction on all three metrics in relation to the classic algorithm, for both sudoku difficulties, evidencing its superiority to performing random choices for what literal to split on next. This is outcome is naturally expected since the MOMs heuristic focuses on the selection of smaller size literals while random selection accounts for nothing when selecting.

The Jeroslaw heuristic turned out to be the best performing one on all metrics for the easy sudokus, out performing both the classic version and the MOMs heuristic. For the hard sudokus the situation was slightly different, the differences between these heuristics wasn't statistically significant.

4 Exeriments and Results

To test out hypothesis we set up the experiment as follows: For every sudoku we will take its original truth values and randomly sample from them. The number of samples taken goes from 2 to the original amount of given truths and we run the algorithm on each of these sampled sets. Finally, we build plots as a function of the number of truth values to determine the validity of the initial hypothesis.

The sudokus used for the experiment are all part of the 1000sudokus.txt file. Since we are investigating the difficulty of the sudokus based on the number of given truth values, the actualy difficulty of the original problem is not highly relevant. Also, this file contains sudokus with up to 24 starting truth values, allowing us to investigate our hypothesis in a range of 2 to 24.

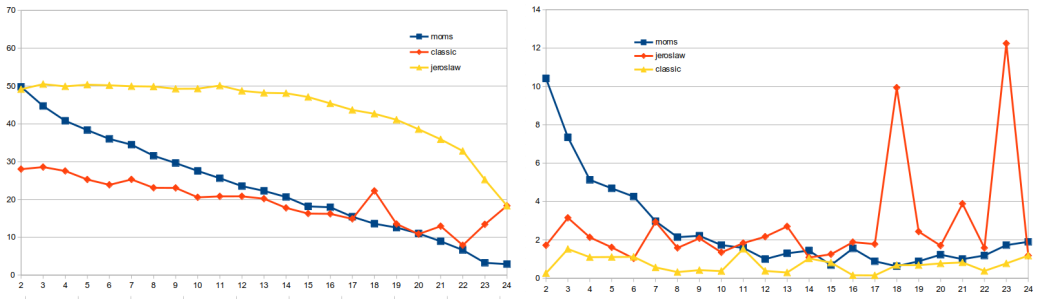

Figure 1: Number of splits (left) and number of backtracks (right) plotted against number of truth values.

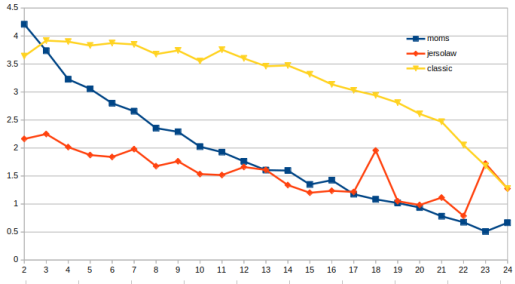

Figure 2: Runtime plotted against number of truth values

The graphs provide us with some interesting insights regarding on what goes on with the different heuristics and different numbers of initial truth values.

Starting with the number of splits plot, we can see a a clear pattern for all heuristics showing a reduction of the required amount of splits with the increase of the number of givens. This is a somewhat expected results since having more givens naturally decreases the size of the solution tree and so each split has a higher chance of

ending in a working solution. Both the classic and the MOMs split number reduction seems to behave linear in contrst with the Jeroslaw heuristic which remains constant at 50 up to around 12 givens, after which it decreases increasingly faster.

The plot of number of backtracks is probably the most surprising one. For the classic version of the algorithm and the Jeroslaw heuristic it seems as if the number of backtracks is random or at least with no clear distinct pattern. Backtracking usually has quite severe impact on runtime and hence on the difficulty of the algorithm so this goes against the initial hypotheses of having a clear distinct peak at a certain number of givens. We are indeed able to observe to peaks at 18 and 23 givens but upon inspection of the data it turns out that both of these were the result of outliers, runs that ran for an unreasonable amount of time and skewed the averages. The MOMs heuristic is the only that shows a clear pattern, requiring a large amount of backtracks when fewer hints are given. This means that the heuristic has trouble in making good decisions in regards to which literal to choose, and often requires changing its decision. This, associated to the higher amount of splits performed by this heuristic, most likely explains its inferior performance when compared to Jeroslaw.

Finally the runtime plot indicates what was already shown earlier in 1. The heuristics pretty much always outperform the classic version but have quite similar runtimes between themselves, with the Jeroslaw being superior for most of the range of givens. One very surprising result is the fact that runtime actually seemed to take the longest for sudokus with a small amount of givens, problems which are easily solved by humans. The possible justification for this is the fact that by constraining the problem less, we exponentially increase the search tree of possible solutions, leading to a higher runtime. This is also evidenced by the larger amount of required splits for the lower bounds of givens, for all heuristics.

When looking back at the formulation of our Hypothesis we can conclude that it has not been confirmed in anyway by the given data. There seems to be no clear peak in either the number of splits, or backtracks, the two main indicators of the difficulty of a sudoku. The source of the observed peaks was checked and it was confirmed that these were the influence of outliers in the data.

5 Conclusion

In this report, we investigated the complexity of Sudoku. After we presented our own design of the DPLL algorithm, we compared its performance against the Jeroslow-Wang and MOMs heuristic. Due to its ability to create simpler sub-problems, the latter turned out to be more efficient in any of the three measurements that we compared with (execution time, number of backtrack and splits) and both of these decision heuristics seemed to outperform the standard DPLL method. Finally, we investigate our hypothesis, by taking a set of Sudokus and removing clauses one by one. Although we did not see the Gaussian shaped like distribution that we were expecting, we conclude in some interesting facts. The computer process Sudokus in a different way that the humans do, and this is clear by the number of backtracks that the algorithm does when there are only 2-3 numbers of the grid; this case is trivial for humans. Furthermore, it is important to mention that the distribution of the given numbers of the Sudoku on the grid, play a crucial role in defying its hardness. For future research, it will be interesting to find hard Sudokus for every number of variables in range 2 to 26, repeat the experiment and compare it with these results.

References

[1] Experimental results on the crossover point in random 3-SAT, Crawford and Auton, 1993 (link)

[2] The Science behind Sudoku - UT Computer Science (link)

[3] Davis, Logemann, and Loveland 1962

[4] Jeroslow, R., Wang, J.: Solving propositional satisfiability problems, Annals of Mathematics and Artificial Intelligence, 1, 167187 (1990) (link)

[5] Guiding CNF-SAT Search by Analyzing Constraint-Variable Dependencies and Clause Lengths (link)

[6] Cliques, coloring, and satisfiability: second DIMACS implementation challenge, October 11-13, 1993, by Johnson, David S and Trick, Michael A