SAT vs. CSP Solver on Identical Problems

Julian Straub - JulianStraub@gatech.edu

Problem Definition

In this project we want to investigate whether it is faster to solve a Satisfiability (SAT) problem using a SAT solver or to solve the dual Constraint Satisfaction Problem (CSP) using a CSP solver. Being able to solve SAT problems is important in many fields of industry and development. One of the most important examples where it is necessary to solve large SAT problems is electronic design automation (EDA). Those techniques aid the development of advanced electronic circuits on which the technological progress of our society relies.

To enable fast prototyping processes high speed SAT solvers are essential. As described in [6] it is possible to convert any SAT problem into a CSP. Hence the question arises whether it is possible to get faster SAT solvers in resolving the associated CSP instead of the original SAT problem.

Related Work

In literature there have been attempts to compare the theoretic expected performance of SAT and CSP solvers on specific problem types [6]. However, there has been little investigation in direct comparison between solvers of the two different types.

Most SAT solvers are a variant of the DPLL algorithm [3], which performs a backtracking search to find a satisfiable solution given a knowledge base consisting of logical sentences. A high-performance general SAT solver is the Chaff algorithm [5], which according to its developers outperforms other available SAT solvers by at least one order of magnitude on hard SAT problems.

CSP solvers are mostly based on backtracking search combined with techniques such as back-jumping, look-ahead and constraint propagation to enforce arc-consistency. One of the best solvers for CSPs that was recently developed is Minion [4].

Obviously both the SAT and the CSP solvers rely on backtracking search methods but deploy different problem specific techniques to speed up the process of finding a solution. This fact makes the comparison even more interesting since we could gain insight into which techniques are more efficient.

Approach

We want to compare the performance of the state of the art SAT and CSP solvers on a given SAT problem. Since the code for the previously mentioned solvers Chaff and Minion is openly available from the respective developers we were able to deploy the original algorithms for the desired comparison.

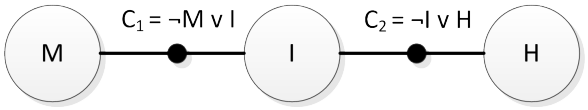

Figure 1: SAT problem $(\neg M \vee I) \wedge (\neg I \vee H)$ expressed in a factor graph.

A general SAT problem can be given as a set of logical expressions. For the purpose of a unified interface for SAT solvers, the convention is to give this set of logical sentences in the conjunctive normal form (CNF). This is possible since all logical expressions, no matter how complex, can be reformulated to have the CNF. The standard for benchmark SAT problems in research is the DIMACS CNF format [1]. Since the zChaff implementation of Chaff that we use takes such files as input we were able to make use of benchmarks available from SATLIB [2].

In order to be able to compare the performance of zChaff against the CSP solver Minion we have to translate the SAT problem to a CSP. This conversion can be done since a CSP is the dual problem to a SAT problem [6]. For example the SAT problem

$$(M \Rightarrow I) \wedge (I \Rightarrow H) \Longleftrightarrow (\neg M \vee I) \wedge (\neg I \vee H)$$

converts to a CSP with four Boolean variables M, I_1, I_2 and H with the three constraints

$$\begin{gathered}(M, I_1)\epsilon\{(0,0);(0,1);(1,1)\} \\ (I_2, H)\epsilon\{(0,0);(0,1);(1,1)\} \\ I_1 = I_2\end{gathered}$$

The benchmark SAT problems from SATLIB contain hundreds of clauses, which makes it impossible to convert each of the problems to a CSP by hand. Looking at the structure of both problems we are able to formulate a algorithm to convert from a SAT to a CSP problem.

Figure 1 depicts the factor graph representation of the SAT problem of the example above. The three variables M, I and H are connected by the two factors $C_1 = \neg M \vee I$ and $C_2 = \neg I \vee H$. Solving this graph means finding assignments for all three variables that satisfy both constraints.

In a CSP we are given a set of possible assignments for each variable, like for example in the map coloring problem R, G and B, and we search for valid assignments of those values to the respective variables

M	I	$C_1 = \neg M \vee I$
0	0	1
0	1	1
1	0	0
1	1	1

I	H	$C_2 = \neg I \vee H$
0	0	1
0	1	1
1	0	0
1	1	1

Table 1: Truth tables of the constraints

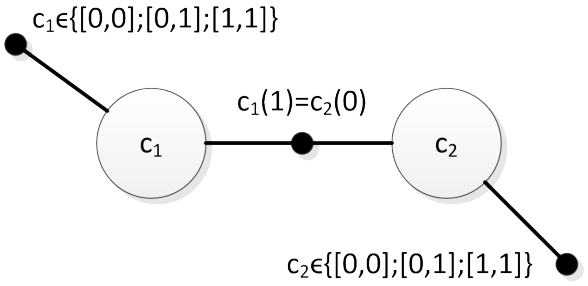

Figure 2: CSP problem derived from $(\neg M \vee I) \wedge (\neg I \vee H)$ expressed in a factor graph

subject to certain constraints. In the map coloring problem we have the constraint that no neighboring states may have the same color.

Coming back to the example problem, we realize that the two constraints $C_1 = \neg M \vee I$ and $C_2 = \neg I \vee H$ of the SAT problem can be expressed in two truth tables as shown in Table 1. For the CSP we now introduce two 2-D Boolean variables c_1 and c_2 that are associated with the two SAT constraints. The fact that each of the SAT constraints has to be true in order to get a valid solution to the problem, restricts the set of values that c_1 and c_2 are allowed to take on. Looking at Table 1, we see that $c_1 \epsilon \{[0,0];[0,1];[1,1]\}$ and $c_2 \epsilon \{[0,0];[0,1];[1,1]\}$ because these are the assignments for which the two SAT constraints C_1 and C_2 are true. Finally, we see that the SAT variable I shows up in both C_1 and C_2. This is modeled in the CSP by the constraint $c_1(1) = c_2(0)$.

Figure 2 displays the factor graph representation of the CSP that we retrieved from the original SAT problem. We now have two Boolean 2-D variables c_1 and c_2 that are constrained to take on only a certain set of values i.e. $c_1 \epsilon \{[0,0];[0,1];[1,1]\}$ and $c_2 \epsilon \{[0,0];[0,1];[1,1]\}$. This is expressed as two unary factors in the graph. Both variables are connected by a binary factor specifying that $c_1(1) = c_2(0)$

This example lets us deduce an algorithm to convert from a SAT to a CSP problem:

For each constraint in the SAT problem:

- Create a new Boolean variable c_i in the CSP with $dim(c_i)$ =Number of literals in the i^{th} SAT constraint.
- From the truth table of the SAT constraint take all assignments that are true and add them to a binary constraint in the CSP on the new variable c_i.

For each variable in the SAT problem:

- Add equality constraints between $c_i(k)$ and $c_j(l)$ if and only if the k^{th} literal in the i^{th} SAT constraint is the same as the l^{th} literal in the j^{th} SAT constraint.

This algorithm has been implemented in matlab to produce input files for the Minion solver. In the CSP problem definition language of Minion it is possible to specify the unary constraints by binary tables containing all valid value assignments of a Boolean vector variable. Also, equality constraints between elements of Boolean vectors can easily be added.

Evaluation

For the evaluation several of the benchmark SAT problems from SATLIB [2] were converted to CSP problems in the Minion input format [4] using the algorithm described in the previous section. Then both zChaff and Minion were used to solve those problems and the time to solve the problem was measured for each algorithm and each problem.

As seen in [5] the cumulative time to solve a set of similar SAT problems will be used as a measure for the performance of the two solvers. Table 2 lists the resulting total times for the two SAT problem sets "jnh" and "aim-sat". An important fact is that both sets contained only satisfiable problems i.e. the existed a solution to the problem. Figure 3 gives a more detailed view on the statistics of the individual solving-times for both algorithms on the "aim-sat" problem set.

Table 3 shows the results for the SAT problem set "aim-unsat". This is a set containing only unsatisfiable problems. It has to be noted that seven of the problems could not be proven to be unsatisfiable by Minion in a time less than 15 minutes. Those were aborted and not taken into the statistic.

SAT Problem	N	zChaff	Minion
jnh	16	0.008s	32.414s
aim-sat	41	0.048s	1593.0s

Table 2: Cumulative times to solve sets of satisfiable benchmark problems from SATLIB. N is the Number of problem instances in the set that were solved.

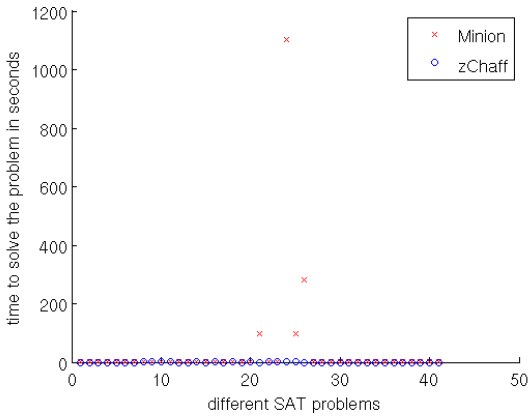

Figure 3: Statistic of the aim-sat benchmark problems for Minion and zChaff.

SAT Problem	N	zChaff	Minion
aim-unsat	17	0.06s	1307.9s

Table 3: Cumulative times to solve sets of unsatisfiable benchmark problems from SATLIB. N is the Number of problem instances in the set that were solved.

Note that every single SAT problem that was evaluated, satisfiable or unsatisfiable, was solved faster by zChaff than by Minion.

Discussion

The results shown in the previous section clearly indicate that zChaff's performance in solving SAT problems is superior to the one of Minion on the dual CSP problems. While the time it took zChaff to solve satisfiable CNFs with hundreds of clauses was barely noticeable, Minion was never faster than zCHaff and in the worst case needed more than three orders of magnitude more time to find a solution. This performance difference gets even clearer when we look at the results of the unsatisfiable problems. Unsatisfiable problems are more difficult, because they require the solver to show that non of all the combinations of possible variable assignments can be true. Here again zChaff shows its performance. It is able to show that the benchmark SAT problems are indeed unsatisfiable in virtually no time. In contrast to that seven of the unsatisfiable CSP problems took Minion too long, so that they had to be aborted. In the cases where Minion found that the CSPs had no solution, the processing times were again more than two orders of magnitude slower than zChaff's.

Obviously zChaff does a much better job in propagating the binary constraints than Minion does. It also seems that the non-forgetting restarts performed by zChaff help to speed up the search. Non-forgetting in this case means that zChaff only clears the assignments to variables but does not forget about clauses added by inference. This way the algorithm does not run into the wrong direction for a long time as Minion seems to. This behavior of Minion can be inferred from the fact that the long cumulative times originate from a small number of very large run times as can be seen in Figure 3.

All in all the results are quite clear: It does not make sense to use Minion instead of zChaff as a SAT solver. This however might only be originating from the implementation of Minion. Hence it would be interesting to try other high performance CSP solvers like the ILOG solver or GeCode instead. For this however the output of the developed SAT to CSP translator would have to be adapted to the specific way those solvers need the CSP problem to be specified.

All in all the project was very interesting since in developing the algorithm to convert from a SAT problem in CN form to a CSP we gained a lot of insight into the structure of SAT and CSP problems. Especially the formalism of the factor graph helped to grasp the nature of those two problem categories in a graphical way. From this it also became clear that the problems are not that dissimilar in their very essence.

References

[1] Dimacs cnf format. http://www.cs.ubc.ca/~hoos/SATLIB/Benchmarks/SAT/satformat.ps.

[2] Satlib benchmark problems. http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html.

[3] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[4] Ian P. Gent, Chris Jefferson, and Ian Miguel. Minion: A fast scalable constraint solver. In *Proceedings of ECAI 2006, Riva del Garda*, pages 98–102. IOS Press, 2006.

[5] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM.

[6] Toby Walsh. Sat v csp. In *Principles and Practice of Constraint Programming CP 2000*, Lecture Notes in Computer Science, chapter 32, pages 441–456. 2000.