(In)Effectiveness of Look-Ahead Techniques in a Modern SAT Solver

Enrico Giunchiglia, Marco Maratea, and Armando Tacchella

DIST, Università di Genova, Viale Causa, 13 – 16145 Genova, Italy
{enrico,marco,tac}@mrg.dist.unige.it

Abstract. In this paper[1] we investigate the effect of adding a failed literal detection method to the traditional unit clause propagation method in the look-ahead component of a modern SAT solver. Our investigation points out that, in all the SAT instances that we have tried, failed literal detection is bound to be ineffective, even assuming it has no overhead.

1 Introduction

In the last couple of years, we have seen a tremendous boost in the performances of SAT solvers, such boost mostly due to Chaff [1]. Chaff owes its efficiency to four components: (*i*) efficient data structures, (*ii*) an innovative look-back method, (*iii*) an effective heuristic, and (*iv*) low-level optimizations of the code. zChaff (Chaff latest incarnation) was the best among the complete SAT solvers on industrial and hand-made benchmarks in the SAT 2002 competition [2]. Thus, when we speak of modern SAT solvers, we have in mind a "Chaff-like" engine.

In this paper we investigate the effect of adding a failed literal detection method to the traditional unit clause propagation method in the look-ahead component of a modern SAT solver. Failed literal detection was first introduced in POSIT [3], used extensively, e.g., in SATZ [4] and RelSAT [5], and similar techniques led to positive results on real-world instances in [6]. Our analysis is mostly experimental and has been performed using our solver Simo modified to incorporate ideas (*i*) - (*iii*) above. We have run our experiments using several challenging real-world benchmarks. On the basis of the collected data, we conclude that enhanced look-ahead based on failed literal detection does not pay off in modern SAT solvers. Further, we show that even assuming we had an oracle answering whether a literal will fail, or an oracle giving us the list of literals which will fail, enhanced look-ahead is not effective.

Throughout the paper, we will assume that the reader is familiar with the topics of Boolean satisfiability and satisfiability search algorithms for Boolean formulas in conjunctive normal form (for an extensive coverage of these topics see, e.g., [7]).

[1] This work is partially supported by MIUR, ASI, and by a grant from the Intel Corporation.

```
DLL-SOLVE()
 1 do
 2   r ← LOOK-AHEAD()
 3   if r = T then
 4     r ← HEURISTIC()
 5   else
 6     r ← LOOK-BACK()
 7 while r = U
 8 return r

LOOK-AHEAD()
 1 r ← UNIT-PROPAGATE(NIL)
 2 if r = F then
 3   return F
 4 else
 5   return FAILED-PROPAGATE()

FAILED-PROPAGATE()
 1 for each open atom a
 2   r ← UNIT-PROPAGATE(a)
 3   LOOK-BACK()
 4   if r = F then
 5     r ← UNIT-PROPAGATE( ¬a)
 6     if r = F then return F
 7   else
 8     r ← UNIT-PROPAGATE( ¬a)
 9     LOOK-BACK()
10     if r = F
11       UNIT-PROPAGATE( a)
12 return T
```

Fig. 1. Overview of Simo.

2 (In)Effectiveness of failed literal detection

For the lack of space, the description of the solvers and the benchmarks used for our experimental analysis is limited to a quick overview (see [8] for more details). We use two versions of Simo: the default configuration and Simo-Fp, i.e., Simo enhanced with failed literal detection as described in Fig. 1. LOOK-AHEAD, Fig. 1 (bottom-left) discriminates the two versions: in Simo, lines 2-5 of LOOK-AHEAD are replaced with the instruction "**return** r"; in Simo-Fp, the implementation of LOOK-AHEAD is exactly as detailed in Fig. 1. The test set consists of 483 real world instances. The benchmarks have been selected considering classical SAT problems and instances submitted to the SAT 2002 competition [2]. All the experiments have been run on two identical Pentium IV 1.8 Ghz, with 512MB of RAM running Linux RedHat 8.0.

2.1 Introducing oracles in Simo-Fp

We can think of three oracle-based versions of Simo-Fp, that we call Simo-Fp(TO), Simo-Fp(FO), and Simo-Fp(FRO). In particular, we assume to have in Simo-Fp(TO), an oracle testing whether a literal will fail, thus saving the time necessary to try the literals which will not be failed; in Simo-Fp(FO), an oracle returning the sequence of literals which will fail in Simo-Fp, thus saving also the time necessary to scan the list of open literals; in Simo-Fp(FRO), an oracle returning the sequence of literals which will fail in Simo-Fp and their reasons, thus saving also the time necessary to calculate the reasons of the failed literals.

Since the oracles cannot be implemented in practice with a single-pass algorithm, we need to calculate the performance of the oracle-based versions using the experimental data of Simo-Fp. In order to accomplish this, we introduce four

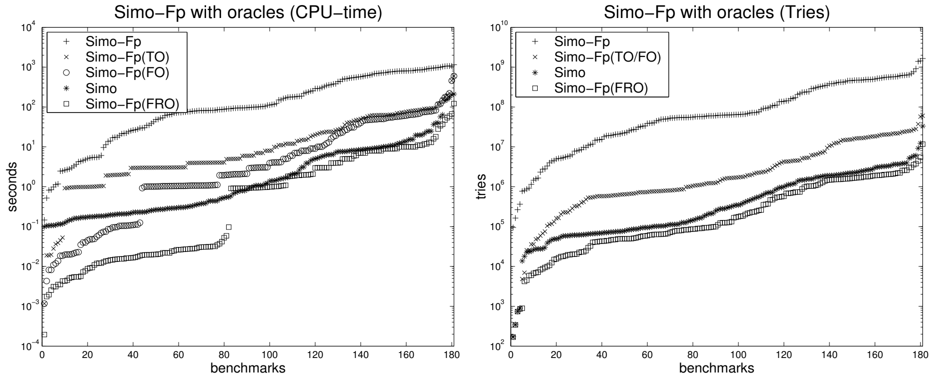

Fig. 2. Simo, Simo-Fp and Simo-Fp(*), considering CPU time (left) and tries (right).

CPU time counters inside FAILED-PROPAGATE (line numbers refer to FAILED-PROPAGATE in Fig. 1): Total time (T_f) is the sum of the run times of each call to FAILED-PROPAGATE; time spent on failed (T_s) is the sum of the run times spent to perform literal propagations when the literals are failed (lines 2-5 and lines 8-11, when the tests on lines 4 and 10 are successful, respectively); time wasted on failed (T_w) is the same as above, but when the literals are not failed (lines 2-3 and lines 8-9, when the tests on lines 4 and 10 are not successful, respectively); time spent on reason (T_r) is the sum of the run times spent to calculate the reason of each failed literal when the literal is failed (lines 2-3 and lines 8-9 when the tests on lines 4 and 10 are successful, respectively). Let T be the CPU time of Simo-Fp, and $T(*)$ be the CPU time of Simo-Fp(*). The performance of the oracle-based algorithms can be calculated as follows: $T(\mathsf{TO}) = T - T_w$, $T(\mathsf{FO}) = T - T_f + T_s$, and $T(\mathsf{FRO}) = T - T_f + T_s - T_r$.

To calculate the tries of the oracle-based versions of Simo-Fp we introduce four more counters (still with reference to FAILED-PROPAGATE in Fig. 1): total tries (N_f) is the sum of the tries performed in each call to FAILED-PROPAGATE; tries spent on failed (N_s) is the sum of the tries spent to perform literal propagations when the literals are failed (the tries performed by UNIT-PROPAGATE in lines 2,5 or 8,11 when the literal is failed); tries wasted on failed (N_w) is the same as above, but when the literals are not failed (the tries performed by UNIT-PROPAGATE in lines 2 or 8 when the literal is not failed); tries spent on reason (N_r) is the sum of the tries spent to calculate the reason of each failed literal when the literal is failed (the tries performed by UNIT-PROPAGATE in lines 2 or 8 when the literal is failed). Let N be the number of tries performed by Simo-Fp, and $N(*)$ the number of tries performed by Simo-Fp(*). The number of tries performed by the oracle-based algorithms can be calculated as follows: $N(\mathsf{TO/FO}) = N - N_w$ and $N(\mathsf{FRO}) = N - N_f + N_s - N_r$. Notice that $N(\mathsf{TO}) = N(\mathsf{FO})$, so we do not need two distinct measures for the tries of Simo-Fp(TO) and Simo-Fp(FO).

Benchmarks					Simo Tries (x1000)			
Family	Sat	Tot	At#	Cl#	Plain	Fp	Fp(FRO)	Fp(TO/FO)
Beijing-1996	8	8	8,226	53,390	1,151	4,475,504	113	5,141
bmc	14	30	10,466	52,995	95,253	5,006,227	33,669	290,623
des	7	7	3,285	20,539	3,073	85,967	786	13,554
fev	0	3	1,324	3,819	1,636	168,740	786	2,814
fpga	10	30	32,612	194,786	10,326	960,441	9,232	48,375
fvp-unsat.2.0	0	5	1,468	15,206	8370	1,047,241	5,438	52,900
mediator	2	2	561.50	12,086	3,689	22,472	1,289	12,833
miters	3	12	2,261	6,119	27,398	2,505,136	23,478	72,405
sss-sat.1.0	79	79	5,022	51,043	75,227	17,553,074	56,333	601,287
vliw-sat.1.1	5	5	20,780	284,509	242	2,657,969	127	8,374

Table 1. Simo, Simo-Fp and Simo-Fp(*) Tries arranged by benchmark family.

2.2 Simo vs. Simo-Fp(*)

In the following, we use *tries* as a CPU independent performance measure, instead of branches. The number of tries is the number of times that a literal is assigned a value, for whatever reason, be it a choice of the heuristic, a unit literal, a failed literal, or a tentative assignment performed during FAILED-PROPAGATE. Considering the oracles presented in the previous subsection, we compare Simo and Simo-Fp with the data of Simo-Fp(TO), Simo-Fp(FO), and Simo-Fp(FRO). For all such real and oracle-based versions of Simo, the distributions of the run time and the number of tries are summarized in Fig. 2. The x-axis in both plots is an ordinal in the range (0-180), and the y-axis is, respectively, CPU seconds in Fig. 2 (left) and number of branches in Fig. 2 (right). The total number of problems visualized is 181 out of 483 since we discarded (i) instances in which either Simo or Simo-Fp exceeded the time out, and (ii) instances in which the run time of Simo was less than 0.1 seconds. Both plots in Fig. 2 are obtained by ordering the results of Simo and Simo-Fp independently and in ascending order.

By looking at Fig. 2 we can immediately conclude that aggressive failed literal detection is bound to be ineffective, both in terms of run time and, more interestingly, also in terms of search space explored. The only version of Simo-Fp that can barely compete with Simo is Simo-Fp(FRO), the version of Simo-Fp embodying the most powerful oracle presented in Sub. 2.1. In spite of its power, the number of tries performed by Simo-Fp(FRO) is, on average, only about 80% of the number of tries performed by Simo. As we can deduce from the plots, Simo-Fp performances are influenced by two major factors: (i) the time (and the tries) spent to check whether a given literal is failed or not, and (ii) the time (and the tries) spent to calculate the reasons of failed literals. By looking at Fig. 2 (right) we can see two order-of-magnitude gaps in the number of tries: one between Simo-Fp and Simo-Fp(TO/FO), which confirms point (i), and one between Simo-Fp(TO/FO) and Simo-Fp(FRO), which confirms point (ii).

To complete our experimental analysis, we need to confirm that the cumulative results of Fig. 2 are true also of each single family, i.e., there are no compensation effects among different families of benchmarks. In Table 1 we present the data regarding Simo, Simo-Fp and the oracle-based versions of Simo-Fp. Each row of the Table contains data about a single family of benchmarks. For each family we report: the number of benchmarks left in the family after instances have been filtered out as described in Sub. 2.2 (column Tot); the number of satisfiable instances (column Sat); the average number of atoms and clauses (columns At# and Cl#, respectively); the total number of tries performed by Simo, Simo-Fp, Simo-Fp(TO/FO) and Simo-Fp(FRO) divided by 1,000. Notice that clauses and atoms statistics have been rounded to 1, and tries statistics have been rounded to 1,000. Although we cannot show here the complete data, compensation effects are absent also when looking at single instances in each family. In other words, there is no single instance in our test set on which Simo-Fp(TO/FO) performs less tries than Simo or Simo-Fp(FRO), while on all the instances Simo-Fp(FRO) performs less tries than Simo.

3 Conclusions

In this paper we have presented strong empirical evidence that enhanced look-ahead based on failed literal detection does not pay off in modern SAT solvers, at least in the case of real-world problems. In particular we showed that (i) the number of tries performed by Simo-Fp is, on average, about three orders of magnitude bigger than the number of tries performed by Simo; (ii) the number of tries performed by Simo-Fp(TO/FO) is, on average, about one order of magnitude bigger than the number of tries performed by Simo; (iii) the number of tries performed by Simo-Fp(FRO) is, on average, only about 80% of the number of tries performed by Simo.

References

1. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proc. of DAC*, 2001.
2. L. Simon, D. Le Berre, and E. A. Hirsch. The SAT2002 Competition.
3. J. W. Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, University of Pennsylvania, 1995.
4. C. M. Li and Anbulagan. Heuristics Based on Unit Propagation for Satisfiability Problems. In *Proc. of IJCAI*, pages 366–371. Morgan-Kauffmann, 1997.
5. R. J. Bayardo, Jr. and R. C. Schrag. Using CSP Look-Back Techniques to Solve Real-World SAT instances. In *Proc. of AAAI*, pages 203–208. AAAI Press, 1997.
6. F. Bacchus. Enhancing Davis Putnam with Extended Binary Clause Reasoning. In *Proc. of AAAI*. AAAI Press, 2001.
7. J. Gu, P. W. Purdom, J. Franco, and B. W. Wah. Algorithms for the Satisfiability (SAT) Problem: A Survey. In *Satisfiability Problem: Theory and Applications*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 19–153. AMS, 1997.
8. E. Giunchiglia, M. Maratea, and A. Tacchella. (In)Effectiveness of Look-Ahead Techniques in a Modern SAT Solver, 2003. Technical report available at `http://www.mrg.dist.unige.it/~tac/Reports/failed.ps`.