An Experimental Evaluation of Conflict Diagnosis and Recursive Learning in Boolean Satisfiability

Fadi A. Aloul and Karem A. Sakallah
Department of Electrical Engineering and Computer Science
University of Michigan
AnnArbor, MI 48109-2122
{faloul, karem}@eecs.umich.edu

***Abstract**: Interest in propositional satisfiability (SAT) has been on the rise lately, spurred in part by the recent availability of powerful solvers that are sufficiently efficient and robust to deal with the large-scale SAT problems that typically arise in electronic design automation application. In this paper we experimentally compare two recent search procedures: recursive learning and conflict diagnosis. Both techniques can be viewed as a form of dynamic identification of "useful" implicates of the function being tested. We also examine the performance of the basic Davis-Putnam search on variants of the input formula that are augmented statically, in a pre-processing step, with additional clauses. The broad conclusions of the paper are that supplanting the input formula with more clauses has the potential of significantly reducing the search effort. This gain, however, must be weighed against the extra effort required to generate the additional clauses. Dynamic clause identification, therefore, is generally more effective than a static pre-processing scheme. Furthermore, clauses identified from conflicts seem generally to be more useful in reducing overall search time than those found by RL.*

1 Introduction

Interest in the direct application of Boolean satisfiability solvers to various EDA computational tasks has been on the rise over the past few years, with applications as diverse as ATPG [9, 14], layout, timing verification, combinational equivalence checking [12], and functional verification. Several modern SAT solvers and algorithms are now available with competing claims to runtime efficiency and robustness [1, 7, 8, 10, 11, 12, 13, 16]. Unfortunately, the "cultural diversity" of the authors of these tools makes it difficult to perform meaningful comparisons among the various approaches and to draw conclusions about their effectiveness in various application domains. In this paper we report some results of a comprehensive experimental evaluation of the two demonstrably most effective of these techniques: recursive learning [7, 8] and conflict diagnosis [11]. We compare these approaches by placing them in a common framework that contrasts their search strategies.

Different approaches can be used to check for the satisfiability of large propositional problems. A straightforward, if impractical, method for determining the satisfiability of a function is to create a complete representation for it, i.e. to "solve it". For example, if the number of variables is sufficiently small, the function truth table can be constructed and checked for 1-entries. This, of course, does not scale very well and is not a useful approach in general. A better approach might be to create a symbolic representation of the function such as a BDD [2] and to determine if the BDD has a 1-terminal. For large functions, however, the BDD sizes might be too large to fit into available computer memory. In any case, approaches that attempt to construct a complete representation of the function before checking its satisfiability end up doing more work than necessary. That's why search-based approaches that intelligently sample the variable space are often more effective in solving SAT problems.

The paper is organized as follows. In the next section, we introduce the mathematical framework that will be used throughout the paper. Afterwards, in Section 3, we describe the SAT solvers used in this paper. Furthermore, we list the major ideas that most search algorithms take into account. Section 4 presents an experimental evaluation of conflict diagnosis and recursive learning. Finally, Section 5 concludes the paper and provides some perspective on future work.

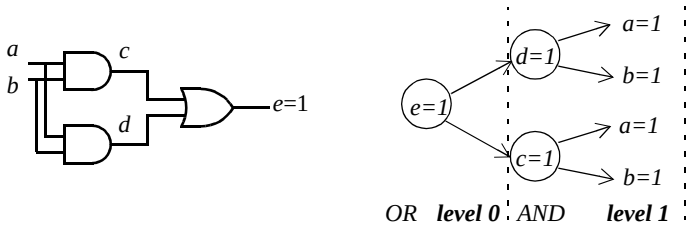

Figure 1: Recursive Learning Example

2 Preliminaries

A *conjunctive normal form* (CNF) formula φ on n binary variables $x_1, \ldots, x_n$ is the conjunction (AND) of m *clauses* $\omega_1, \ldots, \omega_m$ each of which is the disjunction (OR) of one or more literals, where a literal is the occurrence of a variable or its complement. The size of a clause is the number of its literals. A formula φ denotes a unique n-variable Boolean function $f(x_1, \ldots, x_n)$ and each of its clauses corresponds to an implicate of f [6]. Clearly, a function f can be represented by many equivalent CNF formulas. A variable x is *monoform* if it is possible to write a CNF formula for the function f in which all literals on x are either exclusively x or x'. A clause ω_i *subsumes* another clause ω_j if $\omega_i \rightarrow \omega_j$; a clause is a *prime* implicate if it is not subsumed by another implicate. A formula is *complete* if it consists of the entire set of prime implicates [6] for the corresponding function. In general, a complete formula will have an exponential number of clauses. We will refer to a CNF formula as a *clause database* (DB) and use "formula," "CNF formula," and "clause database" interchangeably. The satisfiability problem (SAT) is concerned with finding an assignment to the arguments of $f(x_1, \ldots, x_n)$ that makes the function equal to 1 or proving that the function is equal to the constant 0.

3 SAT Solvers Description

Most modern SAT algorithms can be classified as enhancements to the basic Davis-Putnam backtrack search approach [3]. The DP procedure performs a depth-first search in the n-dimensional space of the problem variables and can be viewed as consisting of three main engines: 1) a *decision* engine that makes *elective* assignments to the variables; 2) a *deduction* engine that determines the consequences of these assignments, typically yielding additional *forced* assignments to, i.e. implications of, other variables; and 3) a *diagnosis* engine that handles the occurrence of conflicts (i.e. assignments that cause the formula to become unsatisfiable) and backtracks appropriately. The deduction engine in the DP procedure is based on the application of two rules: 1) the *unit clause rule* which forces the assignment of the only unassigned variable in a clause whose other literals are all 0; and 2) the *pure literal rule* which forces the assignment of monoform variables to the values that satisfy all the clauses containing them. Repeated application of the unit clause rule over a given clause DB is referred to as *Boolean Constraint Propagation* (BCP) [15].

Conflict Diagnosis (CD) can be viewed as adding "why" analysis to the diagnosis engine. The procedure is called after each conflict to analyze the causes of the conflict and generate adequate information to prevent the conflict from occurring in other parts of the search space. The generated information is represented by conflict-induced clauses which are added to the clause database. Another enhancment to CD involves non-chronological backtracking. By tracing the causes of a conflict, the search process can backtrack directly to the decision assignments that led to the conflict.

On the other hand, Recursive Learning (RL) can be viewed as adding "what-if" analysis to the deduction engine. The procedure was initially introduced in the context of solving structural SAT problems arising in test pattern generation. In our experimental evaluation, we adapt the structural RL procedure to solve CNF satisfiability problems. It dynamically identifies necessary assignments after each decision by examining the justification options for each unresolved clause containing the most-recently decided variable. Variables that are implied to the same value under all possible justifications correspond to *necessary* assignments that are recorded and that can potentially yield further implications. Note that these assignments cannot be identified by BCP. A simple example of RL is shown in Fig. 1. Assuming the assignment

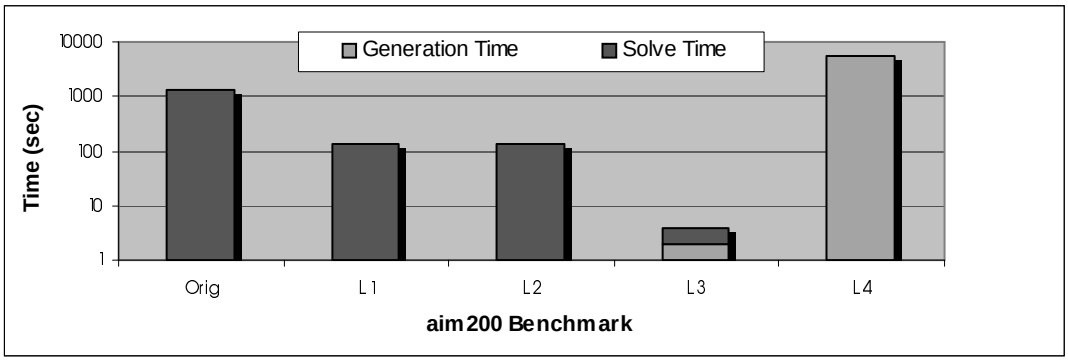

Figure 2: Generation and Search times v.s. Formula Composition

$e = 1$, BCP would be unable to derive further assignments. However, by exploring the two choices for justifying this assignment, namely $c = 1$ and $d = 1$ we conclude that the assignment $a = b = 1$ is necessary because it is the same for all justifications. This "what-if" analysis can be performed recursively for each level in the problem up to a user-specified upper bound.

An important distinction that will be critical in explaining performance differences between various SAT algorithms is whether they are *formula* or *function satisfiers*. A *formula* satisfier performs the search for a satisfying assignment assuming a fixed set of input clauses; it attempts to satisfy a function *f* based on a specific formula φ. The DP algorithm is an example of a formula satisfier. A *function* satisfier, on the other hand, may modify the formula φ representing the function being satisfied to improve search efficiency. One way to classify function satisfiers is based on when and how they modify their input formula. *Static function satisfiers* "pre-process" the input formula, augmenting it with extra prime implicates and removing from it any subsumed clauses. In contrast, *dynamic function satisfiers* identify "useful" clauses adaptively "during" the search, either by the deduction or the diagnosis engines. These clauses help generate further implications and may optionally be stored in the clause database for possible future use. CD and RL are examples of dynamic function satisfiers.

In the context of the DP deduction rules, different formulas representing the same function may possess different "reasoning powers" and may yield vastly different implications. Indeed, it is easy to show that a "complete" formula contains all possible implications to any set of decisions and will not lead to conflicts and backtracking. Unfortunately, a complete formula will generally have an exponential number of clauses, and will likely yield no run time savings. Identifying the "right" formula for a given function, i.e. the formula that consists of just the right number and type clauses to minimize search time, is a clearly desirable but unfortunately unattainable goal.

4 Performance Evaluation

We present results for the DIMACS challenge benchmarks [4]. The benchmarks represent a variety of non-EDA SAT instances and SAT formulations for single stuck-at and bridging faults. All experiments were conducted on a 333 MHz Pentium Pro running Linux and equipped with 512 MByte of RAM. All procedures were implemented in C++. The CPU time limit and conflict limit were set to 1000 seconds and 50M respectively, unless stated otherwise.

4.1 Formula Satisfiers vs. Static Function Satisfiers

To establish a baseline for our experimental study, we examined the behavior of the DP procedure on different formulas representing the same function. For each benchmark formula, we created up to four variants by adding consensus clauses using a *truncated* iterative consensus procedure [5]: noting that the unrestricted application of iterative consensus may lead to the creation of an exponential number of clauses, the truncated version of the procedure discards generated consensus clauses whose size exceeds a

Benchmark	#I	Davis-Putnam		Recursive Learning		Conflict Diagnosis		Conflict Diagnosis & Recursive Learning	
		#S	ST	#S	ST	#S	ST	#S	ST
aim	**72**	**40**	34679	**72**	189	**72**	5	**72**	4
bf	**4**	**0**	4000	**1**	3001	**2**	2001	**3**	1035
dub	**13**	**2**	11751	**13**	0	**13**	1	**13**	0
f	**3**	**0**	3000	**0**	3000	**0**	3000	**0**	3000
g	**4**	**0**	4000	**0**	4000	**0**	4000	**0**	4000
hanoi	**2**	**1**	1529	**1**	1044	**2**	201	**1**	1022
hole	**5**	**5**	427	**4**	1731	**5**	905	**4**	1678
ii	**41**	**21**	20129	**30**	15864	**34**	9127	**34**	11761
jnh	**50**	**50**	285	**50**	30	**50**	35	**50**	24
par	**30**	**20**	12223	**10**	20004	**18**	13375	**13**	19600
pret	**8**	**4**	4621	**4**	4009	**8**	3	**8**	3
ssa	**8**	**4**	4030	**6**	2002	**6**	2002	**7**	1520
TOTAL	**240**	**147**	100675	**191**	54874	**210**	34655	**205**	43647

Table 1: Davis-Putnam vs. Recursive Learning vs. Conflict Diagnosis

given limit L unless they subsume an existing clause. The iterative consensus procedure time limit was 10,000 seconds.

Fig. 2 shows the result on the aim-200 benchmark (other benchmarks show similar results and are omitted for lack of space). As this diagram clearly illustrates, the addition of consensus clauses to a formula leads to a reduction in the number of decisions and conflicts. On the other hand, the total run time tends to initially decrease then dramatically increase as more clauses are added. In addition, the proportion of the run time due to clause generation becomes dominant as L increases. We should note that the consensus procedure did not generate additional clauses for some benchmarks because they already consisted of all the prime implicates.

4.2 Formula Satisfiers vs. Dynamic Function Satisfiers

As mentioned earlier, rather than pre-processing an input formula to identify extra clauses, dynamic satisfiers generate clauses *during* the search process. We compare here two such dynamic satisfiability schemes: Conflict Diagnosis and Recursive Learning.

We decided to use GRASP [11] as our conflict analysis tool. RL was implemented as a new software layer on top of DP. We enhanced RL with clause recording. The maximum recursive learning limit was set to 1 level. The experimental results of running DP, RL, and CD are shown in Table 1. **#I**, **#S**, **ST** represent the total number of instances, number of solved instances, and total solve time respectively. As can be seen, both CD and RL outperform the DP procedure. Yet, CD was faster than RL in all cases. This is justified by the fact that RL was too time consuming during the search process because it was executed after each decision. In contrast, CD was executed only after each conflict.

As a measure of the effectiveness of the recorded clauses, we combined both RL and CD and applied it with clause recording to the DIMACS benchmarks. The obtained experimental results are shown in Table 1. Our results indicate that the combined version was slower than running CD alone. This may be explained by the fact that calling RL after each decision incurs an overhead in detecting necessary assignments. We should note that J. Silva et. al [12] shows that a combined version of RL & CD with clause recording performs better than CD with clause recording. However, in his experiments, RL was applied only once as a preprocessing step before applying CD during the search process.

5 Conclusions & Future Work

Our experimental results indicate that techniques of improving the deduction and diagnosis engine narrow down to augmenting the formula with additional clauses. Additional clauses or implications can be added in a pre-processing step (statically) using the consensus procedure or during the search process (dynamically) using conflict diagnosis or recursive learning. Pre-processing necessarily has no knowledge of how the search process will evolve and may create more "useless" clauses than necessary. On the other hand, dynamic search can augment the clause database with more "useful" clauses than blind pre-processing.

We presented a fair comparison between recursive learning and conflict diagnosis to measure the effect of identified necessary assignments in each. The assignments were converted to clauses and added to the clause database. RL implements a form of forward reasoning as a result of decision assignments, on the other hand, CD implements a form of backward reasoning as a result of conflicts. Furthermore, we tested the effect of combining RL and CD. The set of experiments suggest that assignments and clauses identified by conflict diagnosis appear more effective in reducing the search time than those identified by recursive learning.

Despite the improvements of engines employed by SAT search techniques, further study of the engines is needed. Improvements to the deduction and diagnose engines consist of identifying essential prime implicates and probing the CNF formula for necessary assignments. Another area of improving search techniques is controlling the growth of the clause database. A smart evaluation method should be devised to delete unused recorded clauses (clauses generated by consensus or conflict analysis) during the search process. Finally, correlating the behavior of these search techniques to the structure of the formulas may provide some useful insights.

6 References

[1] R. Bayardo Jr. and R. Schrag, "Using CSP Look-Back techniques to Solve Real-World SAT Instances," in Proc. of the National Conference on Artificial Intelligence, pp. 203-208, July 1997.

[2] R. Bryant, "Graph-based algorithms for boolean function manipulation," in Proc. of IEEE Transactions on Computers 35(8), pp. 677-691, 1986.

[3] M. Davis and H. Putnam, "A Computing Procedure for Quantification Theory," Journal of the Association for Computing Machinery, vol. 7, pp. 201-215, July 1960.

[4] DIMACS Challenge benchmarks in *ftp://Dimacs.rutgers.EDU/pub/challenge/sat/benchmarks/cnf*.

[5] G. Hachtel and F. Somenzi, "Logic Synthesis And Verification Algorithms," Kluwer Academic Publishers, 1996.

[6] J. Hayes, "Introduction to Digital Logic Design," Addison-Wesley, 1993.

[7] W. Kunz, D. Pradhan, "Recursive Learning: A new Implication Technique for Efficient Solutions to CAD-problems: Test, Verification and Optimization," IEEE Transaction on Computer-Aided Design, 13(9), pp. 1143-1158, September 1994.

[8] W. Kunz and D. Stoffel, "Reasoning in Boolean Networks," Kluwer Academic Publishers, Boston, MA, 1997.

[9] T. Larrabee, "Test Pattern Generation Using Boolean Satisfiability," IEEE Transactions on Computer-Aided Design, 11(1), pp. 4-15, January 1992.

[10] B. Selman and H. Kautz, "Domain-Independent Extensions to GSAT: Solving Large Structured Satisfiability Problems," in Proc. of the International Joint Conference on Artificial Intelligence, 1993.

[11] J. Silva and K. Sakallah, "GRASP-A New Search Algorithm for Satisfiability," in Proc. of the International Conference on Computer Aided Design, November 1996.

[12] J. Silva and T. Glass, "Combinational Equivalence Checking Using Satisfiability and Recursive Learning," in Proceedings of the Design and Test in Europe Conference, March 1999.

[13] G. Stalmarck, "System for Determining Propositional Logic Theorems by Applying Values and Rules to Triplets that are Generated from Boolean Formula," United States Patent no. 5,276,897, 1994.

[14] P. R. Stephan, R. K. Brayton and A. L. Sangiovanni-Vincentelli, "Combinational Test Generation Using Satisfiability," IEEE Transactions on Computer-Aided Design, 15(9), pp. 1167-1176, September 1996.

[15] R. Zabih and D. A. McAllester, "A Rearrangement Search Strategy for Determining Propositional Satisfiability," in Proc. of the National Conference on Artificial Intelligence, pp. 155-160, 1998.

[16] H. Zhang, "SATO: An Efficient Propositional Prover," in Proc. of International Conference on Automated Deduction, pp. 272-275, July 1997.