Efficient Conflict Driven Learning in a Boolean Satisfiability Solver

Lintao Zhang
Dept. of Electrical Engineering
Princeton University
lintaoz@ee.princeton.edu

Conor F. Madigan
Department of EECS
MIT
cmadigan@mit.edu

Matthew H. Moskewicz
Department of EECS
UC Berkeley
moskewcz@alumni.princeton.edu

Sharad Malik
Dept. of Electrical Engineering
Princeton University
malik@princeton.edu

ABSTRACT

One of the most important features of current state-of-the-art SAT solvers is the use of conflict based backtracking and learning techniques. In this paper, we generalize various conflict driven learning strategies in terms of different partitioning schemes of the implication graph. We re-examine the learning techniques used in various SAT solvers and propose an array of new learning schemes. Extensive experiments with real world examples show that the best performing new learning scheme has at least a 2**X** speedup compared with learning schemes employed in state-of-the-art SAT solvers.

1. INTRODUCTION

The Boolean Satisfiability problem (SAT) is one of the most studied NP-Complete problems because of its significance in both theoretical research and practical applications. Given a Boolean formula, the SAT problem asks for an assignment of variables so that the formula evaluates to true, or a determination that no such assignment exists. Various EDA applications ranging from formal verification [2][3] to ATPG [1] use efficient SAT solvers as the basis for reasoning and searching. Many dedicated solvers (e.g. GRASP [6], POSIT [5], SATO [8], rel_sat [7], Walksat [4], Chaff [9]) based on various algorithms (e.g. Davis Putnam [10], local search [4], Stälmark's algorithm [11]) have been proposed to solve the SAT problem efficiently for practical problem instances.

Most often, the Boolean formulae of SAT problems are expressed in Conjunctive Normal Form (CNF). A CNF formula is a logical **and** of one or more *clauses*, where each clause is a logical **or** of one or more *literals*. A literal is either the positive or the negative form of a *variable*. To satisfy a CNF Boolean formula, each clause must be satisfied individually. For a certain clause, if all but one of its literals has been assigned the value 0, then the remaining literal, referred to as a *unit literal*, must be assigned the value 1 in order to satisfy this clause. Such clauses are called *unit clauses*. The process of assigning the value 1 to all unit literals is called *unit propagation*. In the following, we will always assume that the discussed formulae are in CNF format.

Since SAT is known to be NP-Complete, it is unlikely that there exists any SAT algorithm with polynomial time complexity. However, SAT problem instances generated from real world applications seem to have some structure that enables efficient solution. A large class of these instances can actually be solved in reasonable compute time. Due to recent advances in search pruning techniques, several very efficient SAT algorithms for structured problems have been developed (i.e. GRASP [6], rel_sat [7], SATO [8], Chaff [9]). Such solvers can solve many instances with tens of thousands of variables, which cannot be handled by other Boolean reasoning methods [2].

Most of the more successful complete SAT solvers are based on the branch and backtracking algorithm called the Davis Putnam Logemann Loveland (DPLL) algorithm [10]. In addition to DPLL, the algorithms mentioned earlier utilize a pruning technique called learning. Learning extracts and memorizes information from the previously searched space to prune the search in the future. Learning is achieved by adding clauses to the existing clause database. Since learning plays a very important role in pruning the search space for structured SAT problems, it is very important to make it as efficient and effective as possible. In this paper, we will examine the learning schemes in more detail. Our specific focus is on learning that occurs as a consequence of conflicts created during the generation of implications. This is referred to as conflict driven learning. For the rest of the paper, we will use the term learning in only this context, without necessarily prefacing it each time with the "conflict driven" adjective.

1.1 DPLL with Learning

The basic DPLL algorithm is the basis for most of the existing complete SAT solvers. In [6] and [7], the authors augmented the DPLL algorithm with learning and non-chronological backtracking schemes to facilitate the pruning of the search space. The pseudo-code of the DPLL algorithm with learning is illustrated in Figure 1.

Function `decide_next_branch()` chooses the branching variable based on various heuristics. Each decision has a decision level associated with it. The function `deduce()` propagates the effect of the decision variable being assigned.

After making a decision, some clauses may become unit clauses. All the unit literals are assigned 1 and the assignment is propagated until no unit clause exists. All variables assigned as a consequence of implications of a certain decision will have the same decision level as the decision variable. If a conflict is encountered (i.e. a clause, called *conflicting clause*, has all its literals assigned value 0), then the `analyze_conflicts()` function is called to analyze the reason for the conflict and to resolve it. It also gains some knowledge from the current conflict, and returns a backtracking level so as to resolve this conflict. The returned backtracking level indicates the wrong branch decision made previously and `back_track()` will undo the bad branches in order to resolve the conflict. A zero backtracking level means that a conflict exists even without any branching. This indicates that the problem is unsatisfiable. Readers are referred to [6] for a more detailed discussion of the DPLL algorithm.

There are a large number of different SAT solvers that differ mainly in how each of these functions is implemented using different heuristics. A lot of effort has been spent on decision-making (e.g. [13][14][9]), and significant progress has been made on how efficient deduction (e.g. [9][15][8]). However, to our knowledge, only [7] and [6] (and its variations) have discussed implementation of conflict driven learning. The authors are not aware of any evaluation of different conflict driven learning schemes and their influence on the performance of SAT solvers.

```
while(1) {
  if (decide_next_branch()) {          //Branching
    while(deduce()==conflict) {        //Deducing
      blevel = analyze_conflicts();    //Learning
      if (blevel == 0)
        return UNSATISFIABLE;
      else back_track(blevel);         //Backtracking
    }
  }
  else  //no branch means all variables got assigned.
    return SATISFIABLE;
}
```

Figure 1. DPLL with Learning

1.2 Implication Graph

The implication relationships of variable assignments during the SAT solving process can be expressed as an implication graph. A typical implication graph is illustrated in Figure 2. An implication graph is a directed acyclic graph (DAG). Each vertex represents a variable assignment. A positive variable means it is assigned 1; a negative variable means it is assigned 0. The incident edges of each vertex are the reasons that lead to the assignment. We will call the vertices that have directed edges to a certain vertex as its *antecedent vertices*. A decision vertex has no incident edge. Each variable has a decision level associated with it, denoted in the graph as a number within parenthesis. In an implication graph with no conflict there is at most one vertex for each variable. A conflict occurs when there is vertex for both the 0 and the 1 assignment of a variable. Such a variable is referred to as the *conflicting variable*. In Figure 2, the variable V_{18} is the conflicting variable. In future discussion, when referring to an implication graph, we will only consider the connected component that has the conflicting variable in it. The rest of the implication graph is not relevant for the conflict analysis.

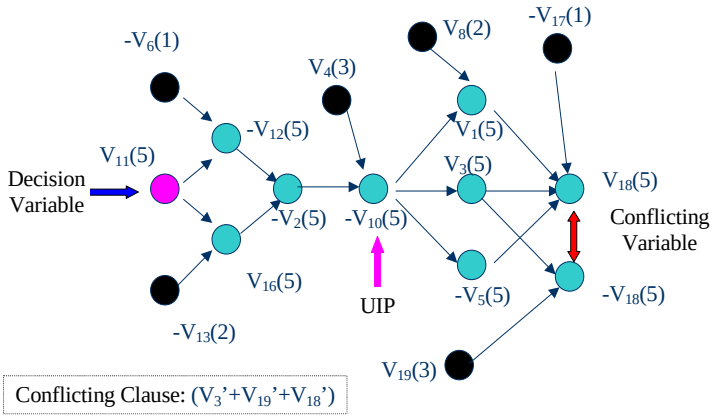

Figure 2. A typical implication graph

In an implication graph, vertex *a* is said to dominate vertex *b* iff any path from the decision variable of the decision level of *a* to *b* needs to go through *a*. A Unique Implication Point (UIP) [6] is a vertex at the current decision level that dominates both vertices corresponding to the conflicting variable. For example, in Figure 2, in the sub-graph of current decision level 5, V_{10} dominates both vertices V_{18} and $-V_{18}$, therefore, it is a UIP. The decision variable is always a UIP. Note that there may be more than one UIP for a certain conflict. In the example, there are three UIPs, namely, V_{11}, V_2 and V_{10}. Intuitively, a UIP is the *single* reason that implies the conflict at current decision level. We will order the UIPs starting from the conflict. In the previous example, V_{10} is the first UIP.

In actual implementations, the implication graph is maintained by associating each assigned non-decision (i.e. implied) variable with a pointer to its *antecedent* clause. The *antecedent* clause of a non-decision variable is the clause that was unit at the time when the implication happened, and forced the variable to assume a certain value. By following the antecedent pointers, the implication graph can be constructed when needed.

1.3 Conflict Analysis and Learning

Conflict analysis is the procedure that finds the reason for a conflict and tries to resolve it. It tells the SAT solver that there exists no solution for the problem in a certain search space, and indicates a new search space to continue the search.

The original DPLL algorithm proposed the simplest conflict analysis method. For each decision variable, the solver keeps a flag indicating whether it has been tried in both phases or not. When a conflict occurs, the conflict analysis procedure looks for the decision variable with the highest decision level that has not been flipped, marks it flipped, undoes all the assignments between that decision level and current decision level, and then tries the other phase for the decision variable. This naïve conflict analysis method actually works well for randomly generated problems, possibly because random problems do not have structure, and learning from a certain part of the search space will generally not help searches in other parts.

More advanced conflict analysis relies on the implication graph to determine the actual reasons for the conflict. This kind of *conflict directed backjumping* [7] could back up more than one level of the decision stack. Thus, it is also called *non-chronological backtracking*. At the same time, the conflict analysis engine will also add some clauses to the database. This process is called *learning*. Such learned clauses are called *conflict clauses* as opposed to *conflicting clauses*, which refer to clauses that cause the conflict. The conflict clauses record the reasons deduced from the conflict to avoid making the same mistake in the future search. These clauses state that certain combinations of variable assignments are not valid because they will force the conflicting variable to assume both the value 0 and 1, thus leading to a conflict.

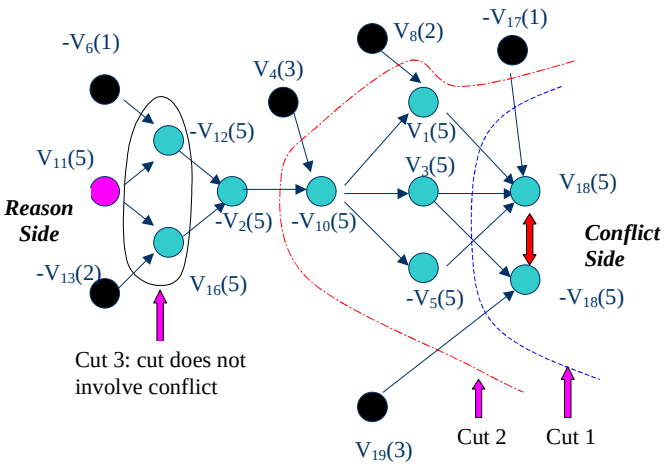

Figure 3. Cuts in the implication graph can be added as clauses

A conflict clause is generated by a bipartition of the implication graph. The partition has all the decision variables on one side (called *reason side*), and the conflicting variable in the other side (the *conflict side*). All the vertices on the reason side that have at least one edge to the conflict side comprise the reason for the conflict. We will call such a bipartition a *cut*. Different cuts correspond to different learning schemes. For example, in Figure 3, clause (V_1'+ V_3' + V_5 + V_{17} + V_{19}') can be added as a conflict clause, which corresponds to cut 1. Similarly, cut 2 corresponds to clause (V_2 + V_4' + V_8' + V_{17} + V_{19}').

We will generalize the UIP concept for decision levels other than the current level in the context of a partition. We will call vertex *a* at decision level *dl* a UIP iff any path from the decision variable of *dl* to the conflicting variable needs to either go through *a*, or go through a vertex of decision level higher than *dl* that is on the reason side of the partition. Note that in this definition, the UIP vertices for a certain decision level are determined by the partition of the vertices in decision levels higher than it. This is unlike the UIPs for the current (i.e. highest) decision level. Also note that the decision variables are UIPs regardless of the partition taken. As in the case of current decision level UIPs, we will order these UIPs starting from the conflicting variable as well.

As mentioned before, each non-decision variable has an antecedent clause that represents the reason for the assignment. The antecedent clause is represented in the implication graph as the incoming edges of a vertex. For example, in Figure 3, the vertex $-V_2$ has two incident edges, from $-V_{12}$ and V_{16} respectively. Therefore, the antecedent clause for V_2 is (V_{16}' + V_{12} + V_2'). This clause is already present in the clause database. We can construct additional clauses that are consistent with the clause database by replacing certain literals in the antecedent clause with all of their antecedent literals. For example, in the above mentioned clause, if we replace V_{16}' and V_{12} with their antecedent literals, we get the clause (V_2' + V_6 + V_{11}' + V_{13}). In Figure 3, this corresponds to cut 3. We will call this kind of replacement a *cut not involving conflicts*. This clause can be used as a learned clause and be added to the database. However, for this kind of learning to be useful, there must exist some reconvergence in the part of the graph that is involved in the learning. Otherwise if the involved part of the graph is a tree structure, then all the information in the learned clause will already be present in the original database, and the learned clause will not be useful.

The learning process plays a very important role in pruning the search space of SAT problems. Some state-of-the-art SAT solvers, (e.g. [9]), utilize a technique called random restarts [16] to help the solver from falling into difficult search regions because of bad decisions in the early decision stages. Random restarting periodically undoes all the decisions and restarts the search from the very beginning. As noted in [12], frequent restarting actually does not hurt the solving process even for the unsatisfiable instances. The knowledge of the searched space is already recorded in the learned clauses. Therefore, discarding all the searched spaces and restarting is not a waste of the previous effort as long as the recorded clauses are still present.

2. Different Learning Schemes

In this section, we will take a look at different learning schemes employed in the existing SAT solvers, and propose some new schemes based on different cuts of the implication graph.

Rel_sat [7] is one of the first SAT solvers to incorporate learning and non-chronological backtracking. The rel_sat conflict analysis engine generates the conflict clause by recursively resolving the conflicting clause with its antecedent, until the resolved clause includes only decision variables of the current decision level and variables assigned at decision levels smaller than the current level.

In the implication graph representation, the rel_sat engine will put all variables assigned at the current decision level, except for the decision variable, on the conflict side; and put all the variables assigned prior to the current level and the current decision variable on the reason side. In our example, the conflict clause added would be:

$$(V_{11}\text{'} + V_6 + V_{13} + V_4\text{'} + V_8\text{'} + V_{17} + V_{19}\text{'})$$

We will call this learning scheme *rel_sat scheme*.

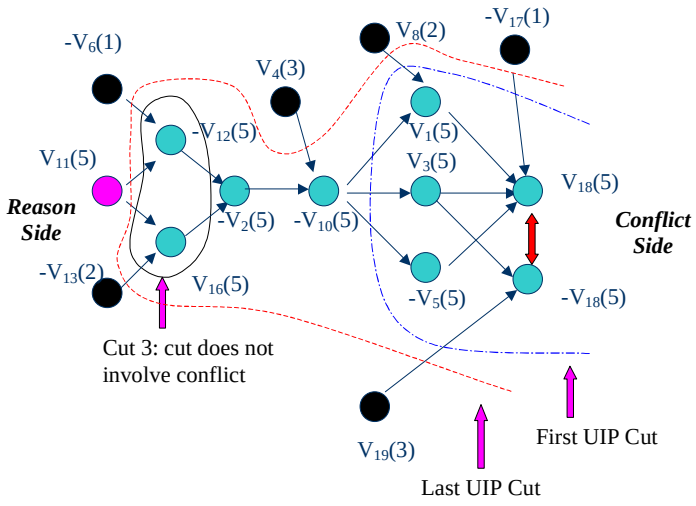

Figure 4. Different cuts

By adding a conflict clause, the reason for the conflict is stated. The maximum decision level of the variables (except the current decision level variable) in this conflict clause is the decision level to backtrack. After backtracking, the conflict clause will become a unit clause, and since the current decision variable is the unit literal, it is forced to flip. Such a clause that causes a flip of the variable is called an *asserting clause*. It is always desirable for a conflict clause to be an asserting clause. The unit variable in the asserting clause will be forced to assume a value and take the search to a new space to resolve the current conflict. To make a conflict clause an asserting clause, the partition needs to have one UIP of the current decision level on the reason side, and all vertices assigned after this UIP on the conflict side. Thus, after backtracking, the UIP vertex will become a unit literal, and make the clause an asserting clause.

Another learning scheme is implemented in GRASP [6]. GRASP's learning scheme is different from rel_sat's in the sense that it tries to learn as much as possible from a conflict. There are two cases when a conflict occurs in GRASP's learning engine. If the current decision variable is a *real decision* variable (explained later), the GRASP learning engine will add each reconvergence between UIPs in current decision level as a learned clause. In our example, if V_{11} is a real decision (i.e. it has no antecedent), when the conflict occurs, the GRASP engine will add one clause into the database corresponding to the UIP reconvergences shown in Figure 4. This corresponds to the clause (V_2' + V_6 + V_{11}' + V_{13}).

Moreover, GRASP will also include a conflict clause that corresponds to the partition where all the variables assigned at the current decision level after the first UIP will be put on the conflict side. The rest of the vertices will be put on the reason side. This corresponds to the FirstUIP cut as shown in figure 3. The clause added will be (V_{10} + V_8' + V_{17} + V_{19}').

After backtracking, the conflict clause will be an asserting clause, which forces V_{10} to flip. Note that V_{10} was not a decision variable before. In GRASP, such a decision is a *fake decision*. The decision level of V_{10} remains unchanged. This essentially means that we are not done at the current decision level yet. We will call this mode of the analysis engine *flip mode*.

If the deduction found that flipping the decision variable still leads to a conflict, the GRASP conflict analysis engine will enter *backtracking mode*. Besides the clauses that have to be added in the flip mode, it also adds another clause called the *back clause*. The cut for the back clause will put all the vertices at the current decision level (including the fake decision variable) on the conflict side, and all other vertices on the reason side. For our example, suppose the decision variable V_{11} is actually a fake decision variable, with antecedent clause (V_{21} + V_{20} + V_{11}'). Then, besides the two clauses added before, the GRASP engine will add another clause

$$(V_{21} + V_{20} + V_6 + V_{13} + V_4\text{'} + V_8\text{'} + V_{17} + V_{19}\text{'})$$

This clause is a conflicting clause, and it only involves variables assigned *before* the current decision level. The conflict analysis engine will continue to resolve this conflict, and bring the solver to an earlier decision level. We will call this learning scheme the *GRASP* scheme.

Besides these two learning schemes, many more options exist. One obvious learning scheme is to add only the decision variables involved in the conflict to the conflict clause. In our implication graph representation, this corresponds to making the cut such that only the decision variables are in the reason side, and all the other variables are in the conflict side. We will call this scheme *Decision* scheme. Note that it is no good to include *all* the decision variables of the current search tree in the conflict clause, because such a combination of decisions

Decision Strategy		Microprocessor Formal Verification[19]			Bounded Model Checking [18]			
		fvp-unsat.1.0(4)	sss.1.0(48)	sss.1.0a(9)	barrel (8)	longmult(16)	queueinvar(10)	satplan(20)
VSIDS	1uip	532.8	24.56	10.63	1012.62	2887.11	6.58	39.34
	2uip	746.87	27.32	16.96	641.64	2734.57	16.37	41.37
	3uip	2151.26	69.12	47.66	656.56	2946.73	19.44	57.16
	alluip	0.68(3)	1746.27(2)	79.09	1081.57(1)	11160.25	18.07	71.86
	rel_sat	2034.09	193.93	82.51	292.33(1)	5719.73	14.4	96.61
	mincut	1612.74(2)	2293.18	11.15	4119.34(1)	7321.69(5)	100.94	43.84
	grasp	2224.44	94.64	33.99	654.54	6196.82	97.82	309.03
	decision*	0(4)	1022.57(17)	227.37(3)	541.96(2)	1421.35(4)	334.39(4)	193.36(3)
FIXED	1uip_f	11.36(3)	15307.13(3)	2997.37	281.48(1)	3141.8	817.07(5)	18(2)
	2uip_f	23.07(3)	18844.51(3)	2646.99	344.34(1)	4279.07	777.2(5)	29.51(2)
	3uip_f	40.75(3)	3985.23(9)	4109.86	432.77(1)	4440.49	860.3(5)	37.62(2)
	alluip_f	0(4)	4063.44(25)	0.28(8)	699.42(1)	11375.32	2025.08(5)	1136.44(2)
	relsat_f	80.94(3)	3114.83(16)	4261.25(4)	293.09(1)	4396.73	478.37(6)	3323.71(3)
	mincut_f	0(4)	5619.4(15)	590.79(4)	3408.28(1)	5232.69(5)	3206.36(4)	373.78(2)
	grasp_f	22.51(3)	6497.99(8)	3382.47	326.46(1)	5597.1	792.12(5)	149.8(2)
	decision_f*	0(4)	415.76(42)	40.57(8)	479.61(1)	1006.58(6)	1.27(8)	600.87(10)

* timeout set to 600s instead of 3600s

Table 1. Run time for different learning schemes

will never occur again in the future unless we restart. Thus the learned clause will hardly help prune any search space at all.

The decision variables are the last UIPs for each decision level. A more careful study of the Decision scheme suggests making a partition not limited at decision variables, but after any UIPs of each of the decision levels. One of the choices is to make a partition after the first UIP of each decision level. We will call this scheme the *All UIP* scheme. Because UIPs of a lower level depend on the partition, we need to do the partition from the current decision level up to decision level 1. More precisely, the solver needs to find the first UIP of the current decision level, then all variables of current decision level assigned after it that have a path to the conflicting variable will be on the conflict side; the rest will be on the reason side. Then the solver will proceed to the decision level prior to current one, and so on, until reaching decision level 1.

Another learning scheme is to make the conflict clause as small as possible. This corresponds to the problem that given an implication graph, remove the smallest number of variables (may include decision variables, but not the conflicting variable) from the implication graph so that there exist no path from the decision variables to the conflicting variable. This is a typical vertex min-cut problem, which can be transformed into an edge min-cut problem and solved with maxflow-mincut algorithms.

It may also be desirable to make the conflict clause as relevant to the current conflict as possible. Therefore, we may want to make the partition closer to the conflicting variable. As we mentioned earlier, to make the conflict clause an asserting clause, we need to put one UIP of the current decision level on the reason side. Therefore, if we put all variables assigned after the first UIP of current decision level that have paths to the conflicting variable on the conflict side, and everything else on the reason side, we get a partition that is as close to the conflict as possible. We will call this *1 UIP* scheme.

The 1 UIP scheme is different from the All UIP scheme in that we only require the current decision level have its first UIP at the reason side just before the partition. We may also require that the immediately previous decision level have its first UIP on the reason side just before the partition. This will make the conflict have only one variable that was assigned at the immediate previous decision level. We will call this the *2 UIP* scheme, similarly, we can have *3 UIP* scheme etc., up to All UIP, which makes the conflict clause have only one variable for any decision level involved.

Other than the min-cut scheme, all the other above-mentioned learning schemes can be implemented by a traversal of the implication graph. The time complexity of the traversal is O(V+E). Here V and E are the number of vertices and edges of the implication graph respectively. The time complexity for finding a min-cut of the implication graph can be implemented in $O(VE\lg(V^2/E))$ time [17].

A good learning scheme should reduce the number of decisions needed to solve certain problems as much as possible. The effectiveness of a learning scheme is very hard to determine *a priori.* Generally speaking, a shorter clause

	9vliw_bp_mc (unsat, 19148 vars)			longmult10 (unsat, 4852 vars)			sat/bw_large.d.cnf (sat, 6325 vars)		
	1UIP	rel_sat	GRASP	1UIP	rel_sat	GRASP	1UIP	rel_sat	GRASP
Branches(x10e6)	1.91	4.71	1.07	0.13	0.19	0.12	0.019	0.046	0.027
Added Clauses(x10e6)	0.18	0.76	1.06	0.11	0.17	0.36	0.011	0.030	0.071
Added Literals (x10e6)	73.13	292.08	622.96	17.16	28.07	77.47	0.56	1.61	6.12
Added lits/cls	405.08	384.31	587.41	150.17	162.96	213.43	50.87	52.94	86.19
Num. Implications(x10e6)	69.85	266.21	78.49	71.72	108.00	74.83	7.23	17.84	16.81
Runtime	522.26	1996.73	2173.14	339.99	612.64	762.94	23.44	52.64	124.13

Table 2. Detailed information on three large test cases for three learning schemes using VSIDS branching heuristic

contains more information than a longer clause. Therefore, a common belief is that the shorter the learned clause, the more effective the learning scheme. However, SAT solving is a dynamic process. It has complex interplay between the learning schemes and the search process. Therefore, the effectiveness of certain searching schemes can only be determined by empirical data for the entire solution process.

3. Experimental Results and Discussions

In this section, we empirically compare the learning schemes discussed in previous sections. We have implemented all the above-mentioned learning schemes in the SAT solver zChaff (an independent implementation that shares several features with the Chaff SAT solver [9]). zChaff utilizes the highly optimized Chaff Boolean Constraint Propagation (BCP) engine and VSIDS decision heuristic, thus making the evaluation more interesting because some large benchmarks can be evaluated in reasonable amount of time.

zChaff's default branching scheme VSIDS depends on the added conflict clauses and literal counts for making branching decisions. Therefore, different learning schemes may also affect the decision strategy. To make a fair comparison, we also implemented a fixed branching strategy, which will branch on the first unassigned variable with the smallest variable index. The variables indices are predetermined for a given problem instance. Default values are used for all other settings.

Our test suite consists of benchmarks from bounded model checking [18] and microprocessor formal verification [19]. We also included a benchmark from the AI planning community to cover a wider range of applications. The experiments were conducted on dual Pentium III 833Mhz machines running Linux, with 2G bytes physical memory. Each CPU has 32k L1 cache and 256k L2 cache. The time out limit for each SAT instance is 3600 seconds. For the Decision learning scheme, because of the large number of aborts, the time out limit was set to 600 seconds to reduce total experiment time.

The run times of different learning schemes on the benchmarks are shown in Table 1. Each benchmark class has multiple instances, shown in the parentheses following each benchmark name. In the result section, the times shown are the total time spent on the solved instances only. The number of aborted instances is shown in the parentheses following run time. If there is no such number, all the instances are solved. For example, fvp-unsat-1.0 has 4 instances. Using the default VSIDS decision strategy, the min-cut learning scheme took 1612.74 seconds to solve two of them, and aborted on the other two.

From the experiment data we can see that for both default and fixed branching heuristics, the 1UIP learning scheme clearly outperforms other learning schemes. Contrary to general belief, the decision schemes that generate small clauses, e.g. min-cut, all UIP and Decision do not perform as well.

We selected the best of our proposed schemes, i.e. the 1UIP scheme, and compared it with the learning schemes employed in other state-of-the-art SAT solvers, i.e. the GRASP scheme and the rel_sat scheme. We selected three difficult test cases from three classes of the benchmark suites. Detailed solver statistics for these schemes are presented in Table 2. The branching heuristic employed was VSIDS. From this table we find that the results are quite interesting. As we mentioned before, the GRASP scheme tries to learn as much as possible from conflicts. Actually, it will learn more than one clause for each conflict. For a given conflict, the clause leaned by 1UIP scheme is actually also learned by the GRASP scheme. Therefore, in most of the cases, the GRASP scheme needs a smaller number of branches to solve the problem. However, because the GRASP scheme records more clauses, the BCP process is slowed down. Therefore, the total run time of the GRASP scheme is more than 1UIP scheme. On the other hand, though rel_sat and 1UIP both learn one clause for each conflict, the learned clause of rel_sat scheme is not as effective as 1UIP as it needs more branches.

4. Conclusions and Future Research

This paper explores the conflict driven learning techniques widely used in current state-of-the-art Boolean satisfiability solvers. We have examined the strength and weaknesses of current learning schemes employed in various SAT solvers.

We have generalized conflict driven learning into a problem of partitioning the implication graph, and have proposed some new learning schemes. These learning schemes were implemented in the SAT solver zChaff and extensive experiments were conducted. We have found that different learning schemes can affect the behavior of the SAT solver significantly. The learning scheme based on the first Unique Implication Point of the implication graph has proved to be quite robust and effective in solving SAT problems in comparison with other schemes.

For future research, it would be interesting to explore the possibility of mixing different learning schemes in a single run of the SAT solving process. As the SAT solving process progresses, the number of literals in the learned clauses grows quickly. For some of the problems, in the later stages of the solving process each learned clause may have several thousand literals. The clause database may become tens of times larger than the original size; very quickly making the solving process memory limited. It would be interesting to explore the possibility of using different learning schemes during different stages of the solution process. This can lead to better control of the size of the learned clause.

As pointed out before, learning is very important to the success of the SAT solvers, but there is still little work on learning in current research of SAT solving algorithms. In fact, there is very little insight about why learning is useful and what makes one learning scheme better than the other. For example, relevance based learning and clause length bounded learning are common practices in current SAT algorithms, but not much attention has been given to choosing optimal parameters for different problems. As SAT based methods becomes more and more popular in a wide range of applications, it is increasingly important for us to answer these questions in a systematic way. This paper represents a step in that direction.

5. REFERENCES

[1] P. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Combinational Test Generation Using Satisfiability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, 1167-1176, 1996.

[2] M. Velev, and R. Bryant, "Effective Use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessors," *Proceedings of the Design Automation Conference*, July 2001.

[3] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. "Symbolic Model Checking without BDDs," *Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99)*, number 1579 in LNCS. Springer-Verlag, 1999.)

[4] B. Selman, H. Kautz, and B. Cohen. "Local Search Strategies for Satisfiability Testing." *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, vol. 26, AMS, 1996.

[5] J. W. Freeman, "Improvements to Propositional Satisfiability Search Algorithms," Ph.D. Dissertation, Department of Computer and Information Science, University of Pennsylvania, May 1995.

[6] J. P. Marques-Silva and K. A. Sakallah, "GRASP: A Search Algorithm for Propositional Satisfiability," *IEEE Transactions on Computers*, vol. 48, 506-521, 1999.

[7] R. Bayardo, and R. Schrag, "Using CSP look-back techniques to solve real-world SAT instances," *Proceedings of the 14th Nat. (US) Conf. on Artificial Intelligence (AAAI-97), AAAI Press/The MIT Press*, 1997.

[8] H. Zhang. "SATO: An efficient propositional prover," *Proceedings of the International Conference on Automated Deduction*, July 1997.

[9] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. "Chaff: Engineering an efficient SAT Solver," *Proceedings of the Design Automation Conference*, July 2001.

[10] M. Davis, G. Logemann, and D. Loveland. "A machine program for theorem proving.," *Communications of the ACM*, (5):394-397, 1962.

[11] G. Stalmarck, "A system for determining prepositional logic theorems by applying values and rules to triplets that are generated from a formula," Technical report, European Patent N 0403 454 (1995), US Patent N 5 27689.

[12] M. Moskewicz, L. Zhang, Y. Zhao, S. Malik and C. Madigan, "Chaff: A Fast SAT Solver for EDA Applications", Presented at the Dagstuhl seminar on SAT v.s. BDD, March 2001. Dagstuhl, Germany.

[13] J. P. Marques-Silva, "The Impact of Branching Heuristics in Propositional Satisfiability Algorithms," *Proceedings of the 9th Portuguese Conference on Artificial Intelligence (EPIA)*, September 1999.

[14] Chu Min Li and Anbulagan, "Heuristics based on unit propagation for satisfiability problems," *Proceedings of the fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97)*, Pages 366-371, Nagoya (Japan), 1997

[15] Chu Min Li, "Integrating equivalency reasoning into Davis-Putnam Procedure," *Proceedings of AAAI 2000*, 2000.

[16] C. P. Gomes, B. Selman, and H. Kautz, "Boosting Combinatorial Search Through Randomization", *Proceedings of AAAI-98*, Madison, WI, 1998.

[17] A. Goldberg, R. Tarjan. "A new approach to the maximum flow problem," *Proceedings of the eighteenth annual ACM Symposium on Theory of Computing*, 1986.

[18] Bounded Model Checking benchmarks available at http://www.cs.cmu.edu/~modelcheck/bmc/bmc-benchmarks.html

[19] M. N. Velev, Microprocessor Formal Verification Benchmarks, available at http://www.ece.cmu.edu/~mvelev