Chaff: Engineering an Efficient SAT Solver

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

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

Ying Zhao, Lintao Zhang, Sharad Malik
Department of Electrical Engineering
Princeton University
{yingzhao, lintaoz, sharad}@ee.princeton.edu

ABSTRACT

Boolean Satisfiability is probably the most studied of combinatorial optimization/search problems. Significant effort has been devoted to trying to provide practical solutions to this problem for problem instances encountered in a range of applications in Electronic Design Automation (EDA), as well as in Artificial Intelligence (AI). This study has culminated in the development of several SAT packages, both proprietary and in the public domain (e.g. GRASP, SATO) which find significant use in both research and industry. Most existing complete solvers are variants of the Davis-Putnam (DP) search algorithm. In this paper we describe the development of a new complete solver, Chaff, which achieves significant performance gains through careful engineering of all aspects of the search – especially a particularly efficient implementation of Boolean constraint propagation (BCP) and a novel low overhead decision strategy. Chaff has been able to obtain one to two orders of magnitude performance improvement on difficult SAT benchmarks in comparison with other solvers (DP or otherwise), including GRASP and SATO.

Categories and Subject Descriptors

J6 [**Computer-Aided Engineering**]: Computer-Aided Design.

General Terms

Algorithms, Verification.

Keywords

Boolean satisfiability, design verification.

1. Introduction

The Boolean Satisfiability (SAT) problem consists of determining a satisfying variable assignment, *V*, for a Boolean function, *f*, or determining that no such *V* exists. SAT is one of the central NP-complete problems. In addition, SAT lies at the core of many practical application domains including EDA (e.g. automatic test generation [10] and logic synthesis [6]) and AI (e.g. automatic theorem proving). As a result, the subject of practical SAT solvers has received considerable research attention, and numerous solver algorithms have been proposed and implemented.

Many publicly available SAT solvers (e.g. GRASP [8], POSIT [5], SATO [13], rel_sat [2], WalkSAT [9]) have been developed, most employing some combination of two main strategies: the Davis-Putnam (DP) backtrack search and heuristic local search. Heuristic local search techniques are not guaranteed to be complete (i.e. they are not guaranteed to find a satisfying assignment if one exists or prove unsatisfiability); as a result, complete SAT solvers (including ours) are based almost exclusively on the DP search algorithm.

1.1 Problem Specification

Most solvers operate on problems for which *f* is specified in conjunctive normal form (*CNF*). This form consists of the logical AND of one or more *clauses*, which consist of the logical OR of one or more *literals*. The *literal* comprises the fundamental logical unit in the problem, being merely an instance of a variable or its complement. (In this paper, complement is represented by ¬.) All Boolean functions can be described in the *CNF* format. The advantage of *CNF* is that in this form, for *f* to be satisfied (*sat*), each individual *clause* must be *sat*.

1.2 Basic Davis-Putnam Backtrack Search

We start with a quick review of the basic Davis-Putnam backtrack search. This is described in the following pseudo-code fragment:

```
while (true) {
  if (!decide()) // if no unassigned vars
    return(satisifiable);
  while (!bcp()) {
    if (!resolveConflict())
       return(not satisfiable);
  }
}

bool resolveConflict() {
  d = most recent decision not 'tried both
ways';

  if (d == NULL) // no such d was found
    return false;

  flip the value of d;
  mark d as tried both ways;
  undo any invalidated implications;
  return true;
}
```

The operation of **`decide()`** is to select a variable that is not currently assigned, and give it a value. This variable assignment is referred to as a *decision*. As each new *decision* is made, a record of that *decision* is pushed onto the *decision stack*.

This function will return false if no unassigned variables remain and true otherwise.

The operation of **`bcp()`**, which carries out Boolean Constraint Propagation (BCP), is to identify any variable assignments required by the current variable state to satisfy *f*. Recall that every *clause* must be *sat*, for *f* to be *sat*. Therefore, if a clause consists of only literals with value 0 and one unassigned literal, then that unassigned literal must take on a value of 1 to make *f sat*. *Clauses* in this state are said to be *unit*, and this rule is referred to as the *unit clause rule*. The necessary variable assignment associated with giving the unassigned literal a value of 1 is referred to as an *implication*. In general, BCP therefore consists of the identification of *unit clauses* and the creation of the associated *implications*. In the pseudo-code from above, **`bcp()`** carries out BCP transitively until either there are no more implications (in which case it returns true) or a *conflict* is produced (in which case it returns false). A *conflict* occurs when implications for setting the same variable to both 1 and 0 are produced.

At the time a *decision* is made, some variable state exists and is represented by the *decision stack*. Any *implication* generated following a new *decision* is directly triggered by that *decision*, but predicated on the entire prior variable state. By associating each *implication* with the triggering *decision*, this dependency can be compactly recorded in the form of an integer tag, referred to as the *decision level* (DL). For the basic DP search, the DL is equivalent to the height of the decision stack at the time the implication is generated.

To explain what **`handleConflict()`** does, we note that we can invalidate all the implications generated on the most recent decision level simply by flipping the value of the most recent decision assignment. Therefore, to deal with a conflict, we can just undo all those implications, flip the value of the decision assignment, and allow BCP to then proceed as normal. If both values have already been tried for this decision, then we backtrack through the decision stack until we encounter a decision that has *not* been tried both ways, and proceed from there in the manner described above. Clearly, in backtracking through the decision stack, we invalidate any implications with decision levels equal to or greater than the decision level to which we backtracked. If no decision can be found which has not been tried both ways, that indicates that *f* is not satisfiable.

Thus far we have focused on the overall structure of the basic DP search algorithm. The following sections describe features specific to Chaff.

2. Optimized BCP

In practice, for most SAT problems, a major potion (greater than 90% in most cases) of the solvers' run time is spent in the BCP process. Therefore, an efficient BCP engine is key to any SAT solver.

To restate the semantics of the BCP operation: Given a formula and set of assignments with DLs, deduce any necessary assignments and their DLs, and continue this process transitively by adding the necessary assignments to the initial set. Necessary assignments are determined exclusively by repeated applications of the *unit clause rule*. Stop when no more necessary assignments can be deduced, or when a conflict is identified.

For the purposes of this discussion, we say that a clause is *implied iif* all but one of its literals is assigned to zero. So, to implement BCP efficiently, we wish to find a way to quickly visit all clauses that become *newly implied* by a single addition to a set of assignments.

The most intuitive way to do this is to simply look at every clause in the database clauses that contain a literal that the current assignment sets to 0. In effect, we would keep a counter for each clause of how many value 0 literals are in the clause, and modify the counter every time a literal in the clause is set to 0. However, if the clause has N literals, there is really no reason that we need to visit it when 1, 2, 3, 4, … , N-1 literals are set to zero. We would like to only visit it when the "number of zero literals" counter goes from N-2 to N-1.

As an approximation to this goal, we can pick any two literals not assigned to 0 in each clause to watch at any given time. Thus, we can guarantee that until one of those two literals is assigned to 0, there cannot be more than N-2 literals in the clause assigned to zero, that is, the clause is not implied. Now, we need only visit each clause when one of its two *watched literals* is assigned to zero. When we visit each clause, one of two conditions must hold:

(1) The clause is not implied, and thus at least 2 literals are not assigned to zero, including the other currently watched literal. This means at least one non-watched literal is not assigned to zero. We choose this literal to replace the one just assigned to zero. Thus, we maintain the property that the two watched literals are not assigned to 0.

(2) The clause is implied. Follow the procedure for visiting an implied clause (usually, this will generate a new implication, unless the unless the clause is already sat). One should take note that the implied variable must always be the other watched literal, since, by definition, the clause only has one literal not assigned to zero, and one of the two watched literals is now assigned to zero.

It is invariant that in any state where a clause can become newly implied, both watched literals are not assigned to 0. A key benefit of the two literal watching scheme is that at the time of backtracking, there is no need to modify the watched literals in the clause database. Therefore, unassigning a variable can be done in constant time. Further, reassigning a variable that has been recently assigned and unassigned will tend to be faster than the first time it was assigned. This is true because the variable may only be watched in a small subset of the clauses in which was previously watched. This significantly reduces the total number of memory accesses, which, exacerbated by the high data cache miss rate is the main bottleneck for most SAT implementations. Figure 1 illustrates this technique. It shows how the watched literals for a single clause change under a series of assignments and unassignments. Note that the initial choice of watched literals is arbitrary, and that for the purposes of this example, the exact details of how the sequence of assignments and unassignments is being generated is irrelevant.

One of the SATO[13] BCP schemes has some similarities to this one in the sense that it also watches two literals (called the head and tail literals by its authors) to detect unit clauses and conflicts. However, our algorithm is different from SATO's in

that we do not require a fixed direction of motion for the watched literals while in SATO, the head literal can only move towards tail literal and vice versa. Therefore, in SATO, unassignment has the same complexity as assignment.

3. Variable State Independent Decaying Sum (VSIDS) Decision Heuristic

Decision assignment consists of the determination of which new variable and state should be selected each time **`decide()`** is called. A lack of clear statistical evidence supporting one decision strategy over others has made it difficult to determine what makes a good decision strategy and what makes a bad one. To explain this further, we briefly review some common strategies. For a more comprehensive review of the effect of decision strategies on SAT solver performance, see [7] by Silva.

The simplest possible strategy is to simply select the next decision randomly from among the unassigned variables, an approach commonly denoted as RAND. At the other extreme, one can employ a heuristic involving the maximization of some moderately complex function of the current variable state and the clause database (e.g. BOHM and MOMs heuristics).

One of the most popular strategies, which falls somewhere in the middle of this spectrum, is the dynamic largest individual sum (DLIS) heuristic, in which one selects the literal that appears most frequently in unresolved clauses. Variations on this strategy (e.g. RDLIS and DLCS) are also possible. Other slightly more sophisticated heuristics (e.g. JW-OS and JE-TS) have been developed as well, and the reader is referred again to [7] for a full description of these other methods.

Clearly, with so many strategies available, it is important to understand how best to evaluate them. One can consider, for instance, the number of decisions performed by the solver when processing a given problem. Since this statistic has the feel of a good metric for analyzing decision strategies — fewer decisions ought to mean smarter decisions were made, the reasoning goes — it has been used almost exclusively as the comparator in the scant literature on the subject. However, not all decisions yield an equal number of BCP operations, and as a result, a shorter sequence of decisions may actually lead to more BCP operations than a longer sequence of decisions, begging the question: what does the number of decisions really tell us? The same argument applies to statistics involving conflicts. Furthermore, it is also important to recognize that not all decision strategies have the same computational overhead, and as a result, the "best" decision strategy — even if that determination is based on a good combination of the available computation statistics — may actually be the slowest if the overhead is significant enough. All we really want to know is which strategy is fastest, regardless of the computation statistics. No clear answer exists in the literature, though based on [7] DLIS would appear to be a solid all-around strategy. However, even RAND performs well on the problems described in that paper. While developing our solver, we implemented and tested all of the strategies outlined above, and found that we could design a considerably better strategy for the range of problems on which we tested our solver. This strategy, termed *Variable State Independent Decaying Sum (VSIDS)* is described as follows:

(1) Each variable in each polarity has a counter, initialized to 0.
(2) When a clause is added to the database, the counter associated with each literal in the clause is incremented.
(3) The (unassigned) variable and polarity with the highest counter is chosen at each decision.
(4) Ties are broken randomly by default, although this is configurable
(5) Periodically, all the counters are divided by a constant.

Also, in order to choose the variable with the highest counter value even more quickly at decision time, a list of the unassigned variables sorted by counter value is maintained during BCP and conflict analysis (using an STL set in the current implementation).

Overall, this strategy can be viewed as attempting to satisfy the conflict clauses but particularly attempting to satisfy *recent* conflict clauses. Since difficult problems generate many conflicts (and therefore many conflict clauses), the conflict clauses dominate the problem in terms of literal count, so this approach distinguishes itself primarily in how the low pass filtering of the statistics (indicated by step (5)) favors the information generated by recent conflict clauses. We believe this is valuable because it is the conflict clauses that primarily drive the search process on difficult problems. And so this decision strategy can be viewed as directly coupling that driving force to the decision process.

Of course, another key property of this strategy is that since it is independent of the variable state (except insofar as we must choose an unassigned variable) it has very low overhead, since the statistics are only updated when there is a conflict, and correspondingly, a new conflict clause. Even so, decision related computation is still accounts for ~10% of the run-time on some difficult instances. (Conflict analysis is also ~10% of the run-time, with the remaining ~80% of the time spent in BCP.) Ultimately, employing this strategy dramatically (i.e. an order of magnitude) improved performance on all the most difficult problems without hurting performance on any of the simpler problems, which we viewed as the true metric of its success.

4. Other Features

Chaff employs a conflict resolution scheme that is philosophically very similar to GRASP, employing the same type of conflict analysis, conflict clause addition, and UIP-identification. There are some differences that the authors believe have *dramatically* enhanced the simplicity and elegance of the implementation, but due to space limitations, we will not delve into that subject here.

4.1 Clause Deletion

Like many other solvers, Chaff supports the deletion of added *conflict clauses* to avoid a memory explosion. However, since the method for doing so in Chaff differs somewhat from the standard method, we briefly describe it here. Essentially, Chaff uses scheduled lazy clause deletion. When each clause is added, it is examined to determine at what point in the future, if any, the clause should be deleted. The metric used is relevance, such that when more than N (where N is typically 100-200) literals in the clause will become unassigned for the first time, the clause will be marked as deleted. The actual memory associated with deleted clauses is recovered with an infrequent monolithic database compaction step.

4.2 Restarts

Chaff also employs a feature referred to as restarts. Restarts in general consist of a halt in the solution process, and a restart of the analysis, with some of the information gained from the previous analysis included in the new one. As implemented in Chaff, a restart consists of clearing the state of all the variables (including all the decisions) then proceeding as normal. As a result, any still-relevant clauses added to the clause database at some time prior to the restart are *still present* after the restart. It is for this reason that the solver will not simply repeat the previous analysis following a restart. In addition, one can add a certain amount of transient randomness to the decision procedure to aid in the selection of a new search path. Such randomness is typically small, and lasts only a few decisions. Of course, the frequency of restarts and the characteristics of the transient randomness are configurable in the final implementation. It should be noted that restarts impact the completeness of the algorithm. If all clauses were kept, however, the algorithm would still be complete, so completeness could be maintained by increasing the relevance parameter N slowly with time. GRASP uses a similar strategy to maintain completeness by extending the restart period with each restart (Chaff also does this by default, since it generally improves performance).

Note that Chaff's restarts differ from those employed by, for instance, GRASP in that they do not affect the current decision statistics. They mainly are intended to provide a chance to change early decisions in view of the current problem state, *including* all added clauses and the current search path. With default settings, Chaff may restart in this sense thousands of times on a hard instance (sat or unsat), although similar results can often (or at least sometimes) be achieved with restarts completely disabled.

5. Experimental Results

On smaller examples with relatively inconsequential run times, Chaff is comparable to any other solver. However, on larger examples where other solvers struggle or give up, Chaff dominates by completing in up to *one to two orders of magnitude less* time than the best public domain solvers.

Chaff has been run on and compared with other solvers on almost a thousand benchmark formulas. Obviously, it is impossible to provide complete results for each individual benchmark. Instead, we will present summary results for each class of benchmarks. Comparisons were done with GRASP, as well as SATO. GRASP provides for a range of parameters that can be individually tuned. Two different recommended sets of parameters were used (GRASP(A) and GRASP(B)). For SATO, the default settings as well as –g100 (which restricts the size of added clauses to be 100 literals as opposed to the default of 20) were used. Chaff was used with the default cherry.smj configuration in all cases, except for the dimacs pret* instances, which required a single parameter change to the decision strategy. All experiments were done on a 4 CPU 336 Mhz UltraSparc II Solaris machine with 4GB main memory. Memory usage was typically 50-150MB depending on the run time of each instance.

Table 1 provides the summary results for the DIMACs [4] benchmark suite. Each row is a set of individual benchmarks grouped by category. For GRASP, both options resulted in several benchmarks aborting after 100secs, which was sufficient for both SATO and Chaff to complete all instances. On examples that the others also complete, Chaff is comparable to the others, with some superiority on the hole and par16 classes, which seem to be among the more difficult ones. Overall, most of the DIMACs benchmarks are now considered easy, as there are a variety of solvers that excel on various subsets of them. Note that some of the DIMACS benchmarks, such as the large 3-sat instance sets 'f' and 'g', as well as the par32 set were not used, since none of the solvers considered here performs well on these benchmark classes.

The next set of experiments was done using the CMU Benchmark Suite [11]. This consists of hard problems, satisfiable and unsatisfiable, arising from verification of microprocessors (for a detailed description of these benchmarks and Chaff's performance on them, see [12]). It is here that Chaff's prowess begins to show more clearly. For SSS.1.0, Chaff is about an order of magnitude faster than the others and can complete all the examples within 100secs. Both GRASP and SATO abort the 5 hard unsat instances in this set, which are known to take both GRASP and SATO significantly longer to complete than the sat instances. Results on using randomized restart techniques with the newest version of GRASP have been reported on a subset of these examples in [1]. We have been unable to reproduce all of those results, due to the unavailability of the necessary configuration profiles for GRASP (again, see [12]). However, comparing our experiments with the reported results shows the superiority of Chaff, even given a generous margin for the differences in the testing environments. For SSS.1.0.a Chaff completed all 9 of the benchmarks – SATO and GRASP could do only two. For SSS-SAT.1.0, SATO aborted 32 of the first 41 instances when we decided to stop running any further instances for lack of hope and limited compute cycles. GRASP was not competitive at all on this set. Chaff again completed all 100 in less than 1000secs, within a 100sec limit for each instance. In FVP-UNSAT.1.0 both GRASP and SATO could only complete one easy example and aborted the next two. Chaff completed all 4. Finally for VLIW-SAT.1.0 both SATO and GRASP aborted the first 19 of twenty instances tried. Chaff finished all 100 in less than 10000 seconds total.

For many of these benchmarks, only incomplete solvers (not considered here) can find solutions in time comparable to Chaff, and for the harder unsatisfiable instances in these benchmarks, no solver the authors were able to run was within 10x of Chaff's performance, which prohibited running them on the harder problems. When enough information is released to run GRASP and locally reproduce results as in [1], these results will be revisited, although the results given would indicate that Chaff is still a full 2 orders of magnitude faster on the hard unsat instances, and *at least* 1 order of magnitude faster on the satisfiable instances.

7. Conclusions

This paper describes a new SAT solver, Chaff, which has been shown to be at least an order of magnitude (and in several cases, two orders of magnitude) faster than existing public domain SAT solvers on difficult problems from the EDA domain. This speedup is not the result of sophisticated learning strategies for pruning the search space, but rather, of efficient engineering

of the key steps involved in the basic search algorithm. Specifically, this speedup is derived from:

- a highly optimized BCP algorithm, and
- a decision strategy highly optimized for speed, as well as focused on recently added clauses.

8. References

[1] Baptista, L., and Marques-Silva, J.P., "Using Randomization and Learning to Solve Hard Real-World Instances of Satisfiability," Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP), September 2000.

[2] Bayardo, R. and Schrag, R.: Using CSP look-back techniques to solve real-world SAT instances, in Proc. of the 14th Nat. (US) Conf. on Artificial Intelligence (AAAI-97), AAAI Press/The MIT Press, 1997, pp. 203–208.

[3] Biere, A., Cimatti, A., Clarke, E.M., and Zhu, Y., "Symbolic Model Checking without BDDs," Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99), number 1579 in LNCS. Springer-Verlag, 1999. (http://www.cs.cmu.edu/~modelcheck/bmc/bmc-benchmarks.html)

[4] DIMACS benchmarks available at ftp://dimacs.rutgers.edu/pub/challenge/sat/benchmarks

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

[6] Kunz, W, and Sotoffel, D., Reasoning in Boolean Networks, Kluwer Academic Publishers, 1997.

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

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

[9] McAllester, D., Selman, B. and Kautz, H.: Evidence for invariants in local search, in Proceedings of AAAI'97, MIT Press, 1997, pp. 321–326.

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

[11] Velev, M., FVP-UNSAT.1.0, FVP-UNSAT.2.0, VLIW-SAT.1.0, SSS-SAT.1.0, Superscalar Suite 1.0, Superscalar Suite 1.0a, Available from: http://www.ece.cmu.edu/~mvelev

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

[13] Zhang, H., "SATO: An efficient propositional prover," Proceedings of the International Conference on Automated Deduction, pages 272-275, July 1997.

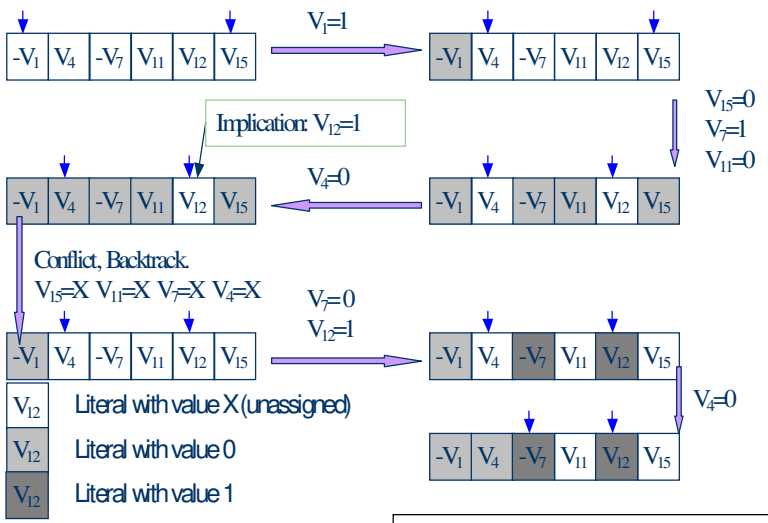

Figure 1: BCP using two watched literals

All times are in seconds.

I = total number of instances in set

A = number of instances aborted. If a number n in () follows this, then only n instances in the set were attempted due to frequency of aborts.

Time = total user time for search, including aborted instances

* = SATO was run with (B) for this set.

= GRASP was run with (B) for this set.

^ = Chaff was run with (B) for this set.

All solvers run with (A) options unless marked. Shown result is for whichever set of options was better for each set.

GRASP options (A):

```
 +T100 +B10000000 +C10000000 +S10000
 +V0   +g40 +rt4 +dMSMM +dr5
```

GRASP options (B):

```
  +T100  +B10000000 +C10000000 +S10000
  +g20 +rt4 +dDLIS
```

SATO options (A): -g100

SATO options (B): [default]

Chaff options (A): cherry.smj config

Chaff options (B): cherry.smj config

```
 plus maxLitsForConfDriven = 10
```

Table 1		GRASP		SATO		Chaff	
	I	Time	A	Time	A	Time	A
ii16	10	241.1#	1	1.3	0	6.2	0
ii32	17	2.3#	0	2.1*	0	0.6	0
ii8	14	1.2	0	0.2	0	0.1	0
aim200	24	6.5	0	0.3	0	0.3	0
aim100	24	0.6#	0	0.1	0	0.1	0
pret	8	5.9#	0	0	0	0.7^	0
Par8	10	0.1#	0	0.1	0	0.1	0
ssa	8	2.7#	0	4.2	0	0.3	0
jnh	50	5.7#	0	0.7	0	0.6	0
dubois	13	0.3	0	0.1	0	0.2	0
hole	5	221.8#	2	99.9*	0	97.6	0
par16	10	845.9#	7	256*	0	42.6	0

Abort timeout was 100s for these sets.

Table 2		GRASP		SATO		Chaff	
	I	Time	A	Time	A	Time	A
SSS 1.0	48	770	5&	16795	5	48	0
SSS 1.0a	8	6031	6	790	6&	20	0
SSS-SAT 1.0	100			33708	32 (41)	457	0
FVP-UNSAT 1.0	4	2018	2 (3)	2007	2 (3)	735	0
VLIW-SAT 1.0	100		19 (20)		19 (20)	3143	0

Abort timeout was 1000s for these sets, except for &'ed sets where it was 100s.