Empirical Study of the Anatomy of Modern SAT Solvers

Hadi Katebi[†], Karem A. Sakallah[†], and João P. Marques-Silva[‡]

[†] EECS Department, University of Michigan
{hadik,karem}@umich.edu
[‡] CSI/CASL, University College Dublin
jpms@ucd.ie

Abstract. Boolean Satisfiability (SAT) solving has dramatically evolved in the past decade and a half. The outcome, today, is manifested in dozens of high performance and relatively scalable SAT solvers. The significant success of SAT solving technology, specially on *practical* problem instances, is credited to the aggregation of different SAT enhancements. In this paper, we revisit the organization of modern conflict-driven clause learning (CDCL) solvers, focusing on the principal techniques that have contributed to their impressive performance. We also examine the interaction between input instances and SAT algorithms to better understand the factors that contribute to the difficulty of SAT benchmarks. At the end, the paper empirically evaluates different SAT techniques on a comprehensive suite of benchmarks taken from a range of representative applications. The diversity of our benchmarks enables us to make fair conclusions on the relation between SAT algorithms and SAT instances.

1 Introduction

SAT solving, today, plays a significant role in modeling and solving real world applications. Although first to be proved NP-complete, SAT gained significant attention due to its practical importance, and managed to achieve major advancements in its algorithms and data structures, specially over the past 15 years. There are currently a number of highly scalable SAT solvers, all based on the classic DPLL search framework. These solvers, known as *conflict-driven clause learning (CDCL)* solvers, can generally handle problem instances with several million variables and clauses.

Modern CDCL solvers differ in many aspects, but they all share four major features. These features, proposed at different stages of SAT development, are:

- Conflict-driven clause learning [23, 24]
- Random search restarts [17]
- Boolean constraint propagation using lazy data structures [27]
- Conflict-based adaptive branching [27]

Centered around the above four features, and spurred in large part by SAT competitions and races, a number of performance techniques have also been incorporated in different solvers including:

– Random branching combined with adaptive branching [14]
– Random initial scoring for conflict-based adaptive branching [14]
– Conflict clause minimization [36]
– Literal phase saving [31]
– Random restart strategies [1, 6, 34]

With the above enhancements, SAT solving has seen dramatic progress. However, modern solvers still fail, unpredictably, on many practical problem instances. Furthermore, even for cases where a solver manages to process an instance, it is generally not obvious what features of the solver contributed most to the instance's tractability. And while most researchers in the field would acknowledge that the above enhancements are generally helpful, there is still some debate about their relative importance. Attempts at "dissecting" modern SAT solvers to isolate the relative contribution to overall performance of the various components of their intricate algorithms have been quite rare. An early attempt is reported in [20], but to our knowledge very little has been reported in the open literature since. In this paper, we review all the aforementioned features of modern CDCL solvers, and experimentally characterize their contribution in solving a suite of 1000 benchmarks chosen from 12 diverse application areas. The diversity of our benchmarks allows us to better understand the behavior of modern solvers and their interaction with input instances. The immediate aim of this article is to experimentally verify the validity of some of the widely-accepted "facts" in the SAT community, and to report possible anomalies. As a larger goal, we hope to raise enough incentive for the theoretical computer science community to develop appropriate theoretical/analytical models that can better explain the remarkable success and the unexpected failures of modern SAT solvers.

The remainder of this paper is organized as follows. Section 2 briefly recounts the major developments in SAT technology, and discusses various performance techniques. Section 3 presents the methodology of our study. Section 4 describes our benchmark suite and articulates the rationale behind our choice. The results of the experiments, obtained using a configurable version of **MiniSAT**, are presented and analyzed in Section 5. Finally, the paper ends with conclusions in Section 6.

2 Major Features of CDCL Solvers

The pioneering techniques to solve the SAT problem, referred to as the DPLL algorithm, go back to the early 1960s [12, 11]. DPLL is composed of three main features: *branching*, *unit propagation* (or *Boolean constraint propagation* (BCP)), and *backtracking*. Branching is essential to move forward in the search space, and backtracking is used to return from futile portions of the space. Unit propagation speeds up the search by deducing appropriate consequences, i.e. *implications*, of branching choices. This basic framework was subsequently extended with several algorithmic enhancements that greatly increased its performance and scalability. In the remainder of this section, we review four of the major enhancements, and highlight several of their extensions. The features discussed in this section have been shown, through extensive empirical evidence, to be critical for scalability and performance. These features are presented in chronological order of their appearance.

2.1 Conflict-Driven Clause Learning

The first major enhancement to DPLL came in 1996 with the debut of the **GRASP** solver [23, 24]. **GRASP** introduced a new *learning* mechanism from *conflicting* assignments. The learning procedure in **GRASP** consists of the following steps:

– Analyzing the conflict and deriving an effective learned clause
– Attaching the newly derived learned clause to the original formula clauses
– Performing non-chronological backtracking

Instead of simply negating all the literals of a conflicting assignment, **GRASP** identifies a small set of assignments that are sufficient to expose the conflict by building an *implication graph.* When this so-called *effective* learning is complete, **GRASP** attaches the new learned clause to the original formula clauses, and backtracks non-chronologically to the decision level where the conflict is resolved.

Recent solvers, such as **MiniSAT** 2.2.0 [13, 14], perform learning by following the exact same steps as proposed in **GRASP**, but also employ additional enhancements in conflict analysis. One such enhancement is *conflict clause minimization* [36] which aims at eliminating redundant literals from a conflict clause. There are two types of conflict minimization implemented in **MiniSAT**: *local* and *recursive.* In local, *self-subsuming resolution* is applied in reverse assignment order, using antecedents marked in the implication graph. In recursive, the conflict clause is recursively minimized by deleting the literals whose antecedents are dominated by other literals of the clause in the implication graph.

2.2 Random Restarts

In 1998, an experimental study [16], conducted by Gomes et al., revealed that the running times of complete search algorithms, such as SAT, often show a non-negligible amount of unpredictability; there always exists a probability of encountering a problem that takes exponentially more time to solve than any other problems encountered before. They explained this behavior by a phenomenon called *heavy-tailed cost distribution.* To avoid heavy tails (mitigate against exponential run times), Gomes et al. suggested the use of a controlled amount of *randomization* in search algorithms [17]. This allows search procedures to escape from regions of the space that contain no solutions. In SAT solving, randomization takes place in the form of restarts. When a SAT solver encounters a certain number of conflicts, it restarts the search by backtracking to the root level of the search tree. The limit on the number of conflicts varies in different solvers, but one common policy, also adopted in **MiniSAT**, is to use the Luby [1] sequence. Other restarting strategies, such as adaptive [6] and problem-specific [34], are also addressed in more recent publications.

2.3 Boolean Constraint Propagation Using Lazy Data Structures

Triggered by the observation that the run time of constraint solvers was mostly dominated by Boolean constraint propagation, a new efficient and highly scalable data structure and related algorithms were introduced by the **Chaff** solver [27] in 2001. The new scheme, referred to as *two-literal watching*, asserts that the status of a clause, required for the propagation process, can be maintained by watching just two of the literals of the clause that are not assigned to 0. The status is updated only when one of the watched literals is assigned to 0. Using this scheme, the clause becomes unit when no non-0-assigned literal other than the other currently watched literal is found. This scheme was in contrast to earlier mechanisms which determined the status of a clause by monitoring a counter that kept track of assignments to the clause's literals. The two-literal watching scheme enabled the status of clauses to be updated lazily and led to a significant reduction in the overhead of BCP.

2.4 Conflict-based Adaptive Branching

Branching heuristics can have a significant effect on the performance of SAT solvers. Ranging from random decision strategies to complicated cost optimization functions, branching heuristics aim to minimize the number of decision steps, while imposing a minimal computational overhead. One effective heuristic, introduced in **GRASP**, is *dynamic largest individual sum (DLIS)* [22]. DLIS maintains counts of literals in unresolved clauses, and selects the literal with the highest count as its next branching decision. A more recent and more effective decision strategy, however, is *Variable State Independent Decaying Sum (VSIDS)*, introduced in **Chaff** [27]. Unlike previous strategies, VSIDS is highly coupled with the clause learning procedure. It attempts to satisfy conflict clauses (particularly, more recent ones) by keeping a counter for each literal, incrementing the counters at the time of a conflict for the literals that appear in the conflict, and choosing the literal with the highest counter at each round of decision. Since VSIDS updates counters only when a conflict is encountered, it has the advantage of incurring very low overhead.

The original VSIDS, as introduced in **Chaff**, kept a counter for each literal. In **MiniSAT**, counters, called *activities*, are associated with variables. Furthermore, **MiniSAT** takes advantage of *literal phase saving* [31] to avoid solving independent subproblems multiple times, when non-chronological backtracking occurs. First introduced by **RSat** [30], phase saving caches the literals that are erased from the list of assignments during backtracking, and uses them to decide on the phase of the variable that the branching heuristic suggests next. Using this strategy, SAT solvers maintain the information of the variables that are not related to the current conflict, but forced to be erased from the list of assignments by backtracking.

3 MiniSAT Configurations

For the experiments in our study, we chose **MiniSAT** 2.2.0 as the constraint solver. By default, **MiniSAT** performs conflict-driven clause learning and provides the following user-specified options:

- `rnd-freq`: This option applies a controlled amount of random decisions (0% to 100%) to VSIDS. 0 is default.
- `rnd-init`: When enabled, the activities of variables are initialized randomly. By default, all activities are initialized to 0.
- `ccmin-mode`: This is used to set the level of conflict minimization, (0) none, (1) basic (local) and (2) deep (recursive). Deep minimization is default.
- `phase-saving`: This option controls the level of phase saving, (0) none, (1) limited, and (2) full. In full, all the literals erased from the list of assignments during backtracking are cached. In limited, only the literals assigned in the latest decision level are saved. Full phase saving is default.
- `luby`: If deactivated, a power of 2 function (i.e., 2^x) with a base interval of 100 is applied as the restarting sequence. Luby is default.

We will refer to the default configuration of **MiniSAT** as CDCL. To assess the contribution of the four major enhancements to DPLL described in Section 2, we instrumented **MiniSAT** with the following additional options:

- **Disable clause learning** (`dis-learn`): When activated, **MiniSAT** reverts to DPLL-style search, i.e, it no longer performs clause learning, or non-chronological backtracking. In our implementation, we still account for conflict analysis, since VSIDS requires this procedure to correctly update variable counts. Note that, since learning is disabled, we discard the result of conflict analysis (i.e., the derived learned clause).
- **Disable restarts** (`dis-restart`): **MiniSAT** applies a Luby restart mechanism with a base interval of 100. In other words, it restarts the search whenever the number of conflicts reaches 100, 100, 200, 100, 100, 200, 400, ... By using this option, restarting is disabled during search.
- **Disable two-watched-literals** (`dis-2WL`): Enabling this option forces **MiniSAT** to perform counter-based BCP.
- **Disable VSIDS** (`DLIS`): When activated, **MiniSAT** applies the DLIS branching heuristic; otherwise it defaults to the VSIDS heuristic.

In our study, we conducted two sets of experiments. In the first set, we measured the relative contribution of each of the four major CDCL features by disabling them one at a time to determine the impact of a feature's absence on performance. These configurations of **MiniSAT** are denoted by ¬CL (no clause learning), ¬RST (no restarts), ¬2WL (counter-based BCP), and ¬VSIDS) (DLIS branching). Our reference for comparison was the default CDCL configuration which enables all of these features. In the second set of experiments, we started with CDCL under default settings for all options and explored the effect of a) adding randomness to VSIDS branching, b) adding randomness to the initial variable activities, c) adjusting the amount of conflict clause minimization, d) changing the level of phase saving, and e) modifying the restart policy.

Table 1: Benchmark families

Family	Instances	SAT	UNS	UNK	Description
`atpg`	100	28	72	0	Circuit testing
`bioinf`	30	8	12	10	Bioinformatics
`config`	50	15	35	0	Product configuration
`crypto`	30	26	3	1	Cryptanalysis
`equiv`	30	5	25	0	Equivalence checking
`fpga`	50	25	22	3	FPGA routing
`hbmc`	250	88	146	16	Hardware bounded model checking
`hverif`	200	125	75	0	Hardware verification
`netcfg`	10	7	2	1	Network configuration
`plan`	80	51	24	5	Planning
`sverif`	120	57	52	11	Software verification
`termrw`	50	26	22	2	Term rewriting
Total:	1000	461	490	49	

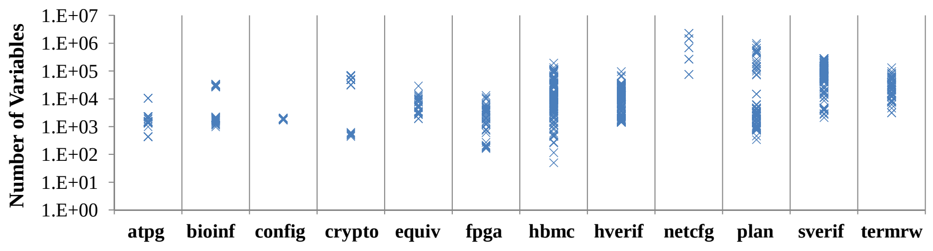
4 Benchmarks

We assembled a suite of 1000 CNF instances from 12 diverse application areas. The list of benchmark families, along with the total number of instances (column "Instances"), and the number of satisfiable, unsatisfiable and unknown instances (columns "SAT", "UNS" and "UNK", respectively) are shown in Table 1[1]. These benchmarks were chosen based on a number of factors including:

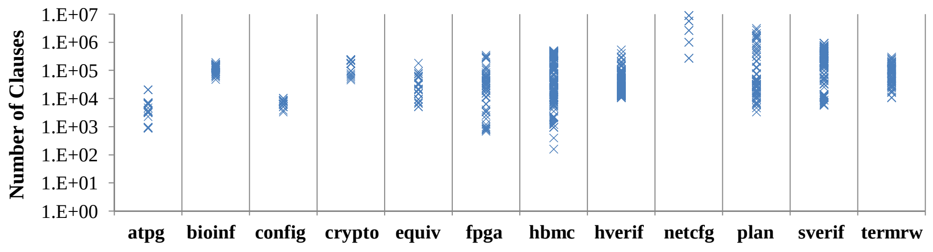
- Representation of real-world problem domains where SAT had been successfully applied over the last decade and a half.
- Representation of benchmark archives that are used to rank solvers in SAT Competitions (`http://www.satcompetition.org/`) and SAT Races (`http://baldur.iti.uka.de/sat-race-2010/`).
- Inclusion of a reasonable number of *easy* problem instances to enable all solver configurations to finish on at least some instances.
- Weighting the participation of each family (in terms of the number of instances representing it) by the relative success of applying SAT solving technology to that family in the recent past.

Our suite consists of benchmarks dated from the early 1990s to today. The oldest benchmarks are from the `atpg`, `plan`, `equiv` and `fpga` families [19, 33, 28]. Of these, `atpg` has seen the most progress in the processing time of its instances. Other families, such as `config` [35], `hbmc` [7], `hverif` [37, 21] and `sverif` [4], represent application areas where SAT was extensively applied over the years. The remaining benchmarks, `netcfg` [29], `termrw` [15], `crypto` [25], and `bioinf` [8, 10], correspond to more recent application domains. The majority of

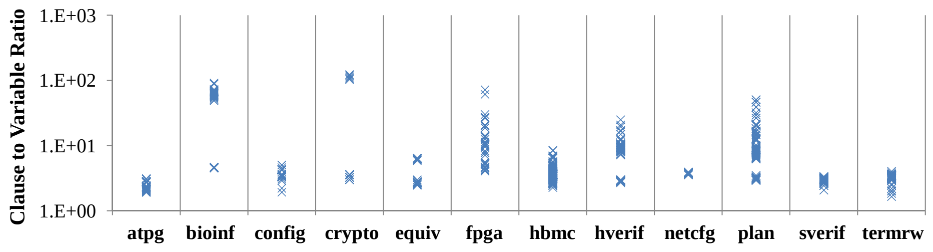
[1] The status of each instance was determined by consulting publicly-available data at various benchmark archives. We were unable to determine the status of 28 instances and tagged them with UNK even though they may be known to be SAT or UNS.



(a) Number of variables



(b) Number of clauses



(c) Clause-to-variable ratio

Fig. 1: Benchmark Statistics.

the instances in our suite have also appeared in SAT competitions. Note that we did not include random benchmarks since a) such benchmarks, especially random 3-SAT, have been studied extensively [26], and b) real-world applications are rarely random.

Figures 1 and 2 provide a variety of statistics for the benchmark families. The benchmarks cover a wide range with the smallest instance (50 variables and 159 clauses) coming from `hbmc` and the largest (2,270,930 variables and 8,901,845) from `netcfg`. For the clause size distributions in Figure 2, we did not include the percentage of 1-literal clauses, since they are eliminated prior to the search.

5 Experimental Evaluation

Our experiments were conducted on a cluster of servers at University College Dublin (UCD) consisting of 3GHz CPUs with 32GB memory and running the

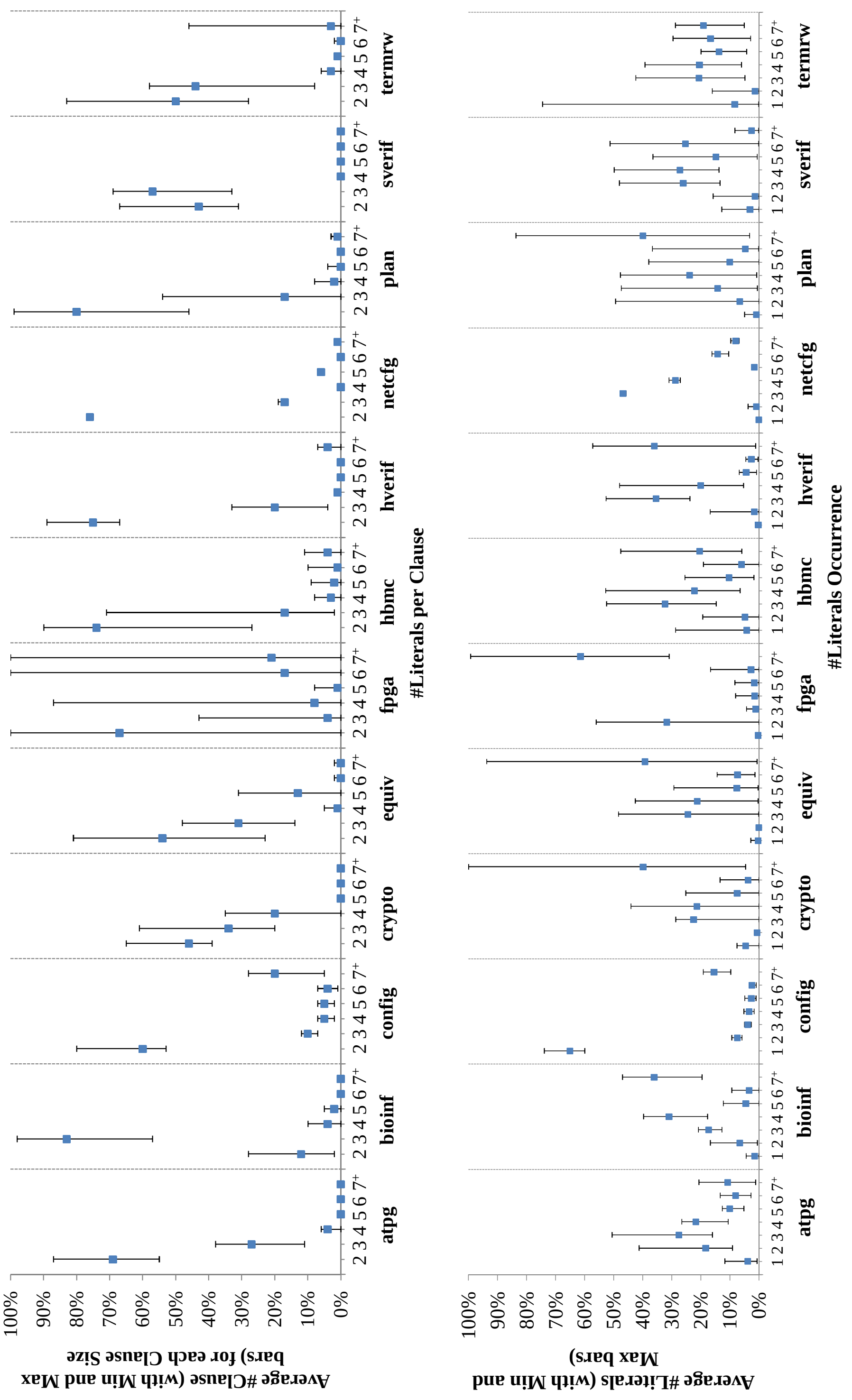


Fig. 2: The distribution of the average #clauses of a given size and #literals of a given occurrence for benchmark families.

Table 2: Number of instances solved by disabling major CDCL features

Family	Runs	¬CL	¬VSIDS	¬2WL	¬RST	CDCL
`atpg`	1000	965	1000	1000	1000	1000
`bioinf`	300	19	34	88	141	150
`config`	500	472	500	500	500	500
`crypto`	300	52	22	113	235	237
`equiv`	300	50	92	187	224	231
`fpga`	500	325	403	444	441	470
`hbmc`	2500	762	1872	2241	2307	2333
`hverif`	2000	1413	1700	1934	1967	1984
`netcfg`	100	0	20	60	74	87
`plan`	800	327	449	559	564	650
`sverif`	1200	336	592	937	754	1006
`termrw`	500	116	248	346	446	420
Total:	10000	4837	6932	8409	8653	9068

64-bit Linux operating system. To obtain meaningful statistical data, we used a script that re-orders the variables and clauses in a CNF instance using a random seed[2] to create ten different versions of each benchmark. We then applied fifteen different configurations of **MiniSAT** to each benchmark version for a total of 150,000 separate runs. Each run was allowed a maximum of 1000 CPU seconds.

5.1 Relative Contribution of Major CDCL Features

Table 2 and Figure 3 summarize the results of the first set of experiments. The goal here was to determine the relative contribution to overall performance, measured by the number of solved instances within the 1000-second time-out, of each of the four CDCL features. This goal was achieved indirectly by disabling the features one at a time as described earlier. Examination of these results leads to the following conclusions:

– The number of instances solved by disabling each of the features suggests the following ordering of their relative importance to solver performance: CL > VSIDS > 2WL > RST. Specifically, disabling clause learning yields the worst performance (finishing on only 4837 instances) followed by disabling VSIDS (6932 instances solved), two-watched-literals (8409 instances solved) and restarts (8653 instances solved). Another way of stating this is to note that the solver configurations that include clause learning (namely, ¬VSIDS, ¬2WL, and ¬RST) dominate the configuration that excludes it. This is not true of the other configurations, i.e., including a feature does not always yield improved performance over excluding that feature. A more direct measure of the relative importance of these features is to compare the configurations

[2] We obtained the reorder.c script and a seed generator from Laurent Simon. The script was originally written by Edward Hirsh and later modified by Simon to handle large benchmarks.

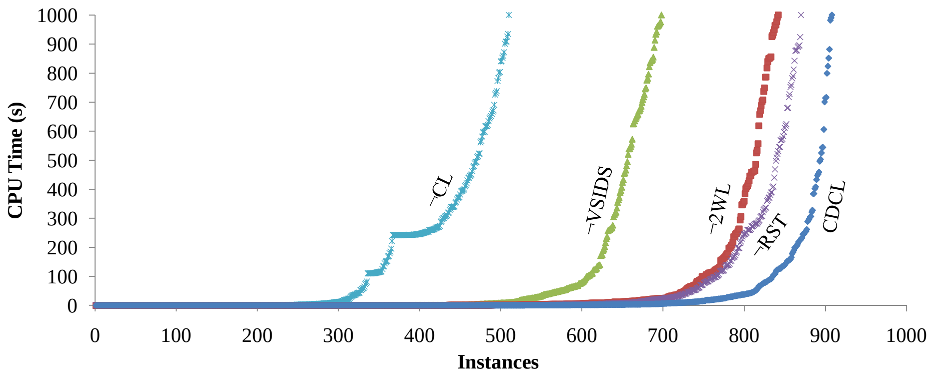


Fig. 3: The run time distribution of the four major CDCL features (data points for timed-out runs are not shown to reduce clutter). These run times are averages over 10 runs per benchmark, and account for time-outs using *maximum likelihood estimation* (MLE) [32]. With a 90% confidence level, 71% of those averages are accurate to within 25%. Higher accuracy can always be obtained by increasing the number of runs.

in which they are disabled against the CDCL configuration in which they are all enabled. Using this measure, we see that enabling CL, VSIDS, 2WL, and RST leads, respectively, to the solution of 4231, 2136, 659, and 415 additional instances.

– Configurations ¬VSIDS and CDCL differ only in the branching heuristic and allow a direct comparison between DLIS and VSIDS. The number of instances solved with VSIDS (9068 in configuration CDCL) is significantly higher than the number solved with DLIS (6932 in configuration ¬VSIDS). Two factors contribute to this performance advantage: a) the much lower overhead of VSIDS compared to DLIS since it only updates activities whenever conflicts arise whereas DLIS updates literal counters every time a literal is assigned/unassigned, b) the selection of literals occurring in the most recent conflicts as opposed to literals occurring the most in unresolved clauses.
– Configurations ¬2WL and CDCL differ only in the implementation of BCP and allow a direct comparison between counter-based and two-watched-literal unit propagation. The number of instances solved with 2WL (9068 in configuration CDCL) is higher than the number solved with the counter-based approach (8409 in configuration ¬2WL). This performance improvement is also due to two factors: a) unlike the counter-based approach which requires updating clause status during branching and backtracking, 2WL propagation needs to update clause status only during branching, and b) 2WL propagation only needs to perform status updates when watched literals are assigned to 0.
– Configurations ¬RST and CDCL differ only in whether restarts are disabled or enabled (using the Luby strategy) and show that the impact of restarts, compared with the other major features, is rather modest. Enabling Luby restarts allows 9068 instances to be solved compared to 8653 instances solved when restarts are disabled. To better understand the behavior of random

Table 3: Number of instances solved under different **MiniSAT** options

Family	CDCL	`rnd-freq`				`rnd-`	`ccmin-mode`		`phase-saving`		`no-`
		25	50	75	100	`init`	none	basic	none	limited	`luby`
`atpg`	1000	1000	1000	1000	1000	1000	1000	1000	1000	1000	1000
`bioinf`	150	133	107	72	46	150	139	149	150	150	148
`config`	500	500	500	500	50	500	500	500	500	500	500
`crypto`	237	67	63	49	35	228	214	223	219	234	**243**
`equiv`	231	221	216	181	162	231	220	222	224	**235**	224
`fpga`	470	456	453	444	421	470	**471**	468	454	463	462
`hbmc`	2333	2328	2322	2225	2057	2328	2328	2333	2318	2326	2315
`hverif`	1984	**1989**	**1993**	**1997**	1949	1984	**1993**	**1991**	1971	**1997**	1960
`netcfg`	87	76	75	60	72	80	76	77	74	74	67
`plan`	650	619	593	526	490	647	637	640	606	636	586
`sverif`	1006	915	858	762	302	1004	1003	996	976	967	944
`termrw`	420	416	407	378	291	420	416	417	**426**	**424**	**444**
Total:	9068	8720	8587	8194	7325	9042	8997	9016	8918	9006	8893

restarts, we examined their effect separately on the SAT and UNS instances. Of the 10000 instances, Luby restarts (configuration CDCL) solved 4533 SAT instances and 4535 UNS instances and timed out on the remaining 932. When restarts were disabled, 4230 SAT and 4423 UNS instances were solved and 1347 instances timed out. These results suggest that, surprisingly, restarts do help for both SAT and UNS instances, but that they are more helpful for SAT instances. However, additional analysis shows that the effect of restarts is not always predictable. For instances, only 420 instances (250 SAT and 170 UNS) of the `termrw` family were solved with restarts whereas 446 (252 SAT and 194 UNS) were solved when restarts were disabled.

– Of the four features, CL and 2WL showed consistent improvement across all instances when they were enabled. In contrast, the performance of VSIDS and RST was more variable. On reflection, this is to be expected as VSIDS and RST are heuristics whereas CL and 2WL are algorithmic optimizations.

As expected, enabling these four features (the CDCL configuration) yields the best performance and explains why most competitive SAT solvers include them in their implementations.

5.2 The Impact of Additional Options in CDCL Solvers

Table 3 reports the number of instances solved by **MiniSAT** (configuration CDCL) when several of its options deviate from their default settings. Bolded entries in the table indicate option settings that led to better performance than the default. These results show that, overall, **MiniSAT** performs best under the default settings. In some cases, however, changing a default setting yields slightly improved performance. For example, adding some randomness to VSIDS helped solve up to 13 more instances of the `hverif` family. Similarly, relaxing conflict clause minimization helped solve up to 9 more instances of the same family. Relaxing phase saving was modestly helpful for the `equiv`, `hverif` and `termrw` families. Finally, applying a power of 2 rather than the Luby restart strategy helps solve more instances in the `crypto` and `termrw` families. Still, Luby is generally more effective, confirming the earlier results reported by Huang [18].

One surprising anomaly in these experiments is the observation that a completely random branching strategy (option `rnd-freq`=100) solved more instances (7325) than the DLIS heuristic (6932). However, DLIS branching solved 477 instances that random branching failed to process! Such mixed results are hard to explain without further detailed analysis of the specific instances involved and any particular attributes they may have.

Finally, unlike the first set of experiments, it is not possible to draw general conclusions from these results as it seems that the optimal values of such settings need to be determined by trial and error. The options analyzed here are best viewed as refinements added on top of the four major features of CDCL. This is partly justified by noting that, unlike CL, VSIDS, 2WL and RST, the inclusion or exclusion of these refinements has, at best, a modest impact on performance.

6 Conclusions

Much effort has been devoted over the past fifteen years to improve the capacity and performance of SAT solvers that are architected around the CDCL framework. On the other hand, few researchers have explored the interactions among the various algorithmic and heuristic components of a modern CDCL solver to determine their relative importance. And while such solvers are successful in processing many practical instances, they still fail, unpredictably, on many others. The question of why CDCL works well on certain instances and not so well on others is rarely addressed in the literature. One of the few attempts to provide a theoretical explanation for the success of clause learning is due to Beame et al. [5] who show that, as a proof system, clause learning is more powerful than regular and therefore DP resolution.

This paper should be viewed as a preliminary attempt to understand the impact on performance of the primary and secondary features of a modern CDCL solver. The ultimate goal should be the development of analytical/theoretical models that relate the performance of a CDCL solver to key attributes of its input SAT instances. Such attributes include the symmetries of CNF formulas [2], the cut width of graph representations of CNF instances [9], and the *scale-free* graph structure of industrial instances [3]. This will help spur further algorithmic improvements as well as the development of customized SAT solvers that can take advantage of such structural attributes.

Acknowledgement

This work was partially supported by SFI grant BEACON (09/PI/12618) and by the United States National Science Foundation under Grant No. 0705103.

This paper is partly based on, and further extends, the article "Anatomy and Empirical Evaluation of Modern SAT Solvers," in Bull. of Euro. Assoc. for Theor. Computer Science, vol. 103, pp. 96-121, February 2011.

References

1. M. L. Alistair, A. Sinclair, and D. Zuckerman. Optimal speedup of las vegas algorithms. *Information Processing Letters*, 47:173–180, 1993.
2. F. Aloul, K. Sakallah, and I. Markov. Efficient symmetry breaking for boolean satisfiability. *Computers, IEEE Transactions on*, 55(5):549 – 558, May 2006.
3. C. Ansótegui, M. L. Bonet, and J. Levy. On the structure of industrial sat instances. In *CP*, pages 127–141, 2009.
4. D. Babic and A. J. Hu. Calysto: scalable and precise extended static checking. In *International Conference on Software Engineering*, pages 211–220, 2008.
5. P. Beame, H. A. Kautz, and A. Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22:319–351, 2004.
6. A. Biere. Adaptive restart strategies for conflict driven SAT solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 28–33, 2008.
7. A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. *Advances in Computers*, chapter Bounded Model Checking. Academic Press, 2003.
8. M. L. Bonet and K. S. John. Efficiently calculating evolutionary tree measures using SAT. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 4–17, 2009.
9. E. Broering and S. V. Lokam. Width-based algorithms for sat and circuit-sat. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 162–171, 2003.
10. F. Corblin, L. Bordeaux, E. Fanchon, Y. Hamadi, and L. Trilling. Connections and integration with SAT solvers: A survey and a case study in computational biology. In *Hybrid Optimization: the 10 years of CPAIOR*. Springer, 2010.
11. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, July 1962.
12. M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, July 1960.
13. N. Eén and N. Sörensson. An extensible SAT solver. In *International Conference on Theory and Applications of Satisfiability Testing*, May 2003.
14. N. Eén and N. Sörensson. MiniSAT (version 2.2.0) *http://minisat.se/downloads/minisat-2.2.0.tar.gz*, 2010.
15. C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. Sat solving for termination analysis with polynomial interpretations. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 340–354, 2007.
16. C. P. Gomes, B. Selman, and N. Crato. Heavy-tailed distributions in combinatorial search. In *International Conference on Principles and Practice of Constraint Programming*, pages 121–135, 1997.
17. C. P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *National Conference on Artificial Intelligence*, pages 431–437, July 1998.

18. J. Huang. The effect of restarts on the efficiency of clause learning. In *Proceedings of the 20th international joint conference on Artifical intelligence*, pages 2318–2323, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc.
19. T. Larrabee. Test pattern generation using Boolean satisfiability. *IEEE Transactions on Computer-Aided Design*, 11(1):4–15, January 1992.
20. I. Lynce and J. Marques-Silva. Building state-of-the-art sat solvers. In *Proceedings of the 15th Eureopean Conference on Artificial Intelligence (ECAI 2002), IOS*, pages 166–170. Press, 2002.
21. P. Manolios and S. K. Srinivasan. A parameterized benchmark suite of hard pipelined-machine-verification problems. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 363–366, 2005.
22. J. Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *Proceedings of the Portuguese Conference on Artificial Intelligence*, pages 62–74, September 1999.
23. J. Marques-Silva and K. A. Sakallah. GRASP: A new search algorithm for satisfiability. In *International Conference on Computer-Aided Design*, pages 220–227, November 1996.
24. J. Marques-Silva and K. A. Sakallah. GRASP-A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.
25. I. Mironov and L. Zhang. Applications of SAT solvers to cryptanalysis of hash functions. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 102–115, 2006.
26. D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of sat problems. In *National Conference on Artificial Intelligence*, pages 459–465, 1992.
27. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Engineering an efficient SAT solver. In *Design Automation Conference*, pages 530–535, June 2001.
28. G.-J. Nam, K. A. Sakallah, and R. A. Rutenbar. Satisfiability-based layout revisited: Detailed routing of complex FPGA s via search-based boolean SAT. In *International Symposium on Field-Programmable Gate Arrays*, February 1999.
29. S. Narain. Network configuration management via model finding. In *Conference on Systems Administration*, pages 155–168, 2005.
30. K. Pipatsrisawat and A. Darwiche. Rsat 1.03: Sat solver description. Technical Report D–152, Automated Reasoning Group, Computer Science Department, UCLA, 2006.
31. K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisfiability solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 294–299, 2007.
32. J. A. Rice. *Mathematical Statistics and Data Analysis*. Duxbury Press, 2006.
33. B. Selman and H. Kautz. Planning as satisfiability. In *European Conference on Artificial Intelligence*, pages 359–363, 1992.
34. C. Sinz and M. Iser. Problem-sensitive restart heuristics for the DPLL procedure. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 356–362, 2009.
35. C. Sinz, A. Kaiser, and W. Küchlin. Formal methods for the validation of automotive product configuration data. *AI EDAM*, 17(1):75–97, 2003.
36. N. Sörensson and A. Biere. Minimizing learned clauses. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 237–243, 2009.
37. M. N. Velev and R. E. Bryant. Effective use of boolean satisfiability procedures in the formal verification of superscalar and vliw microprocessors. *J. Symb. Comput.*, 35(2):73–106, 2003.