Improvements to Hybrid Incremental SAT Algorithms

Florian Letombe and Joao Marques-Silva

School of Electronics and Computer Science, University of Southampton, UK
`{fl,jpms}@ecs.soton.ac.uk`

Abstract. Boolean Satisfiability (SAT) solvers have been successfully applied to a wide range of practical applications, including hardware model checking, software model finding, equivalence checking, and planning, among many others. SAT solvers are also the building block of more sophisticated decision procedures, including Satisfiability Modulo Theory (SMT) solvers. The large number of applications of SAT yields ever more challenging problem instances, and motivate the development of more efficient algorithms. Recent work studied hybrid approaches for SAT, which involves integrating incomplete and complete SAT solvers. This paper proposes a number of improvements to hybrid SAT solvers. Experimental results demonstrate that the proposed optimizations are effective. The resulting algorithms in general perform better and, more importantly, are significantly more robust.

1 Introduction

Motivated by significant improvements to Boolean Satisfiability (SAT) solvers over the last decade, SAT has been applied to a large number of areas, including model checking [2], model finding, planning, bioinformatics, and security, among many others. The widespread use of SAT in so many areas generates a large number of challenging problems instances, many of which modern SAT solvers are not capable of solving. This in turn, motivates the development of ever more effective SAT solvers. Nevertheless, recent years have seen a slowdown in improvements made to SAT solvers. As a result, a number of alternatives has been considered, one of which is the use of hybrid incremental SAT solvers [5], that build on existing SAT algorithms that are effective in solving different types of problems. The hybrid incremental SAT solver combines the power of local search (LS) SAT solvers and of conflict-driven clause learning (CDCL) SAT solvers. These more complex algorithms are expected to provide additional performance improvements, in general not as the first choice SAT solver, but as a reliable alternative SAT solver for more complex SAT instances, with the goal of increased robustness in industrial settings.

This paper develops a number of optimizations to the original hybrid incremental SAT algorithm. The proposed optimizations provide better performance and, more importantly, significantly improve the robustness of SAT solvers. A comprehensive experimental evaluation on industrial SAT problem instances provides evidence that the proposed optimized hybrid incremental SAT solver is more robust than other existing SAT solvers.

The paper is organised as follows. The next section provides a necessarily brief perspective on SAT solvers (CDCL and LS SAT solvers), as well as the original hybrid incremental SAT solver [5]. Afterwards, section 3 presents several optimizations to the basic hybrid incremental SAT algorithm. A comprehensive experimental evaluation is summarized in section 4. Additional related work is briefly surveyed in section 5. Finally, the paper concludes in section 6.

2 Boolean Satisfiability Solvers

This section provides a quick overview of Boolean Satisfiability solvers, including Conflict-Driven Clause Learning (CDCL) SAT solvers, Local Search (LS) SAT solvers, and the recent generation of hybrid incremental SAT solvers. CDCL SAT solvers are widely used for solving large industrial problem instances. LS SAT solvers are used in less applied contexts, but have also been used for developing branching heuristics for complete solvers.

Most propositional decision procedures assume the input problem to be in conjunctive normal form (CNF). Moreover, the SAT problem is defined as follows. A formula Σ in CNF is represented as a set of clauses, each clause is a set of literals, and each literal is either a positive or negative propositional variable in V. Moreover, a formula is interpreted as a conjunction of clauses, and a clause is interpreted as a disjunction of literals. For example, $(x_1 \vee \neg x_2 \vee x_3) \wedge (x_4 \vee \neg x_5)$ is represented as $\{\{x_1, \neg x_2, x_3\}, \{x_4, \neg x_5\}\}$. The SAT problem consists in finding an assignment (also called a model) for a subset of V satisfying each clause in a CNF formula or proving that no such assignment exists.

2.1 CDCL and LS SAT Solvers

CDCL SAT solvers follow the organization of the DPLL algorithm [3], but integrate a number of effective techniques, including clause learning [17], lazy data structures [21] and search restarts [10]. CDCL SAT solvers have evolved from the original solvers [17], which essentially proposed clause learning for SAT, to the more recent CDCL SAT solvers, that also integrate lazy data structures and search restarts [21,9,4]. A number of these concepts, used in the following sections, are briefly reviewed below (see [17,21,9,4] for additional detail). A CDCL SAT solver is usually organized into three main engines [17,21,4]: the decision engine, used for branching; the deduction engine, used for unit propagation and identification of unsatisfied clauses (or *conflicts*); and the diagnosis engine, used for clause learning. A *decision level* is associated with each assigned variable. Decision levels measure the depth of the search tree in terms of the number of variables the SAT algorithm has branched on. Variables can be assigned a Boolean value, either resulting from a decision (or branching step), or as the result of unit propagation [3]. Variables assigned as the result of unit propagation are said to be *implied.* With each implied variable the SAT algorithm also associates a *reason* or *antecedent*, representing the clause that explains why the variable is implied. The set of assigned variables and associated reasons implicitly represent the *implication graph* [17]. Finally, the process of *clause learning* consists of traversing the implication graph from a given unsatisfied clause using the reasons of implied variable assignments, and recording unsatisfied literals assigned at decision levels less than the current one. The resulting set of recorded literals is then used to create a new clause, which serves for backtracking non-chronologically, and for preventing the same conflict from occurring again during the search process.

Local search is a meta-heuristic for solving computationally hard optimization problems. It can be used on problems that can be formulated as finding a solution maximizing a criterion among a number of candidate solutions. Local search algorithms move from solution to solution in the space of candidate solutions (the search space) until a solution deemed optimal is found or a time bound is reached. In SAT a candidate solution is a truth assignment, and the target is to maximize the number of satisfied clauses by the assignment. In this case, the final solution is of use only if it satisfies all clauses. Local search for solving SAT became notorious with `GSAT` [25] and is often very effective at finding models of satisfiable formulas [13]. LS starts by assigning random values to all the variables. If the assignment satisfies all clauses, the algorithm stops and returns it. Otherwise, the value of a variable is changed, and the process is repeated. The variable to change is the one that minimizes the number of unsatisfied clauses in the new assignment. If no assignment satisfying all clauses has been found after a fixed number of iterations (called cutoff or number of flips), the algorithm starts again with a new random assignment. The algorithm terminates either when a model of the formula has been found or when the number of restarts exceeds a fixed number.

Selman *et al.* have also proposed improvements to `GSAT`, including `WalkSAT`, whose main differences to `GSAT` are the addition of random noise, and the step of selecting variables to be flipped from unsatisfied clauses [24]. Many other new heuristics have been proposed, including among others `HSAT` [8], `Novelty` and `R-Novelty` [19], `Novelty+` and `R-Novelty+`, `Adaptive Novelty+` [12], and `g2wsat` (including `adaptg2wsat+`) [15]. Local search SAT solvers are incomplete, and so cannot prove unsatisfiability. It should be noted that recent work has shown how to use local search for proving unsatisfiability [22], but then the resulting algorithm can no longer prove satisfiability.

2.2 Hybrid Incremental SAT Solvers: The `hbisat` Algorithm

Recent work has addressed hybrid solutions for SAT, where both LS and CDCL SAT solvers cooperate to solve a target problem instance. This section overviews `hbisat` (for HyBrid Incremental SAT solver) [5], given its promising experimental results. The motivation for the `hbisat` algorithm is to combine the power of LS SAT solvers for finding solutions of satisfiable formulas and the power of CDCL SAT solvers for proving formulas to be unsatisfiable. Albeit past work focused on using assignments suggested by LS solvers to help CDCL solvers selecting decision variables and assignments, `hbisat` proposed the opposite: partial models computed by the CDCL solver serve to initialize the LS solver truth assignment. Clauses not satisfied by the LS solver are sent to the CDCL solver, trying to either identify an unsatisfiable sub-formula in the clauses sent to the CDCL SAT solver, or satisfying the clauses in the LS solver. Preliminary experimental results suggest this approach can be effective [5]. The `hbisat` procedure is summarized in the not underlined part of algorithm 1.1. Note that Σ_Γ is initially empty, and α is first randomly initialized.

A proof that `hbisat` algorithm is sound and complete can be found in [5]. Clearly, if the LS procedure LSSolve can find a truth assignment, then the initial formula is satisfiable. Otherwise, some clauses, chosen from those unsatisfied (or *broken*) during local search, are sent to the CDCL solver. Besides these, a few additional clauses can also be sent to the CDCL solver, subject to a number of *criteria* outlined below. In the algorithm description, the clauses sent to the CDCL solver are denoted $\Sigma_\Lambda^{Criteria}$, and computed with getClausesToSend procedure. If the CDCLSolve procedure concludes that the sub-formula is unsatisfiable, then the original problem instance is also unsatisfiable, and the algorithm terminates. Alternatively, if the CDCLSolve procedure concludes that the sub-formula is satisfiable, the computed assignment, obtained with procedure getModel, serves to initialize the next iteration of the LS solver. In the `hbisat` algorithm the following *criteria* are used to decide which clauses are sent to the CDCL solver [5].

1. Unsatisfied clauses, i.e. clauses that the LS solver was not able to satisfy;
2. Clauses containing the most flipped variable during local search;
3. Clauses with two or more of its literals having opposite polarities to literals in broken clauses.

Note that the last two criteria are empirically only applied after the first four calls to function isSatisfiable. Moreover, all remaining clauses are sent to the CDCL solver if one of the two following additional criteria is satisfied [6]:

1. Less than 1% or less than 50 clauses remain to be sent to the CDCL solver;
2. The number of learnt clauses in the CDCL solver is greater than 20% of the total number of clauses.

For each execution of `hbisat` algorithm, the operation $\Sigma_\Gamma \leftarrow \Sigma_\Gamma \cup \Sigma_\Lambda^{Criteria}$ denotes that the clauses identified by the above criteria and added to the clauses in the CDCL SAT solver. Finally, the algorithm allows for three hundred iterations, i.e. recursive calls to procedure isSatisfiable. Afterwards, all remaining clauses are sent to the CDCL SAT solver to solve the problem.

The `hbisat` algorithm is illustrated in figure 1 (top). A set of clauses is associated with each solver, and the solid black squared portions represent clauses identified by the above criteria. These are the clauses sent to the CDCL solver.

3 New Hybrid Incremental SAT Algorithms

This section proposes several optimizations to the `hbisat` algorithm. All optimizations are included in a new hybrid incremental SAT solver, `hinotos`[1], that can be configured to also implement the original `hbisat` algorithm.

[1] `hinotos` denotes Hybrid Incremental SAT for NOTOS, a LTL model checker.

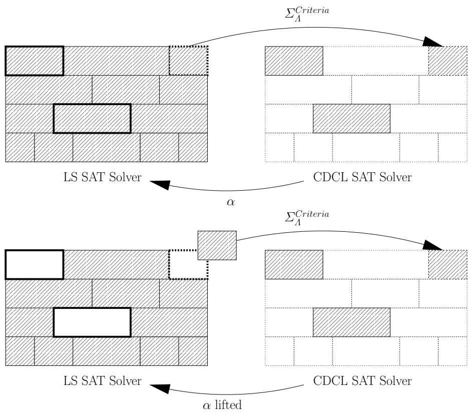


Fig. 1. Solver interaction in `hbisat` (top) and in `hinotos` (bottom)

3.1 Variable Lifting and Blocking Clauses

Modern SAT solvers identify complete assignments. The main reason for this is motivated by the use of lazy data structures, which prevent having knowledge of clause state [21]. Most often, a reasonable number of branching decision made by SAT solvers are irrelevant for satisfying all clauses of a problem instance. *Variable lifting* is the process of removing literals or, equivalently, variables from a satisfying assignment such that for each valuation of the lifted variables, the formula is still satisfied [23].

Example 1 (Variable lifting). Define $V = \{x_1, x_2, x_3, x_4, x_5, x_6\}$ and the following propositional formula Σ over V:

$$(x_1 \vee \neg x_2 \vee x_6) \wedge (x_1 \vee \neg x_2 \vee \neg x_4) \wedge (x_1 \vee \neg x_2 \vee x_4) \wedge (x_3 \vee x_4 \vee x_5) \wedge (x_1 \vee x_2 \vee x_6).$$

Whereas the lifted assignment $\{x_1, x_3\}$ is enough to satisfy the formula, a CDCL solver will identify a complete assignment, e.g. $\{x_1, \neg x_2, x_3, \neg x_4, \neg x_5, x_6\}$. Σ is satisfied, independently of assignments to the other variables $\{x_2, x_4, x_5, x_6\}$.

In `hbisat` complete assignments are sent from the CDCL SAT solver to the LS SAT solver at each step. This can be ineffective, since the LS solver will be unable to identify the assignments that are *relevant* from the ones that are *not relevant*. As a result, variable lifting can be used effectively in hybrid incremental algorithms as shown in the next section. For the experimental results presented in this paper, the variable lifting procedure consists of simply scanning the watched literals in every clause and selecting one of the watched literals as the one that satisfies the clause. As a result, variables not used for satisfying clauses can be lifted.

Another effective technique used in model checking is the use of *blocking clauses* [20] for quantification (or equivalently for model enumeration). Blocking clauses prevent previously computed satisfying partial assignments from being recomputed again during the search, and are created after variable lifting is applied. In hybrid incremental SAT algorithms, the use of blocking clauses guarantees that no part of the search tree is visited more than once.

3.2 Optimized Interaction between the LS and CDCL Solvers

A detailed analysis of the original `hbisat` algorithm reveals that in several situations the same search space can be re-visited, and that the number of times this can happen can be arbitrarily large, up to the limit imposed by the number of times the LS solver is called. The LS solver can repeatedly re-visit the same complete assignments, independently of the information provided by the CDCL solver. In contrast, the CDCL solver may have to redo parts of the search space, motivated by the fact that the LS solver may force the CDCL solver to reconsider branching decisions already made. These are the main sources of inefficiency in the `hbisat` algorithm.

One optimization that addresses these two problems is to guarantee that the clauses in the two solvers are distinct. Each clause sent by the LS solver to the CDCL solver is *removed* from the LS solver. Hence, at any stage, the original clauses are divided into two sets, one associated with the LS solver and the other with the CDCL solver, and these two sets form a partition of the original set of clauses. As a result, the overhead of the LS solver is *effectively reduced* at each iteration of the algorithm due to the reduced number of clauses. Moreover, assignments communicated by the CDCL solver must be *respected* by the LS solver. As a result, variables assigned by the CDCL solver are said to be *tabu* to the LS solver, and will be untouched by the LS solver. Clearly, this idea can be effective only if variable lifting is applied; otherwise the CDCL solver would assign all variables, and the LS solver would be unable to flip any variable.

Figure 1 (bottom) illustrates how `hinotos` implements the interaction between the LS and the CDCL solvers. The clauses sent to the CDCL solver, and removed from the LS solver, are shown in solid black squares.

3.3 Additional Criteria for Moving Clauses

Besides the optimizations outlined in the previous sections, two additional criteria are used for moving clauses to the CDCL SAT solver. The first criterion ensures that a minimal amount of clauses is sent into the CDCL solver. In the original algorithm [5], at least one clause, picked randomly, was ensured to be sent at each step of the algorithm. Since 1% of clauses (or 50 clauses) are considered to be "simple" enough for the CDCL solver, this same amount of clauses is also considered each time clauses are moved to the CDCL solver. As a result, instead of the 300 iterations proposed in the original `hbisat` algorithm, the new criterion implies that at most 100 iterations are executed. The second criterion for moving clauses (from the LS solver to the CDCL solver) is a natural consequence of the organization of the `hinotos` algorithms. If all variables become tabu in the LS solver, then the LS solver becomes irrelevant, and all clauses are sent to the CDCL solver.

Algorithm 1.1. hbisat and hinotos

```
function isSatisfiable
Input: $\Sigma_\Lambda$ and $\Sigma_\Gamma$ two CNF formulas, with $\Sigma = \Sigma_\Lambda \cup \Sigma_\Gamma$ and $\Sigma_\Lambda \cap \Sigma_\Gamma = \emptyset$;
       $\alpha$ an assignment.
Output: True if $\Sigma$ is satisfiable, False otherwise.
begin
    Tabu($\Sigma_\Lambda$, $\alpha$); /* New tabu variables to LS solver */
    if LSSolve($\Sigma_\Lambda$, $\alpha$) = SAT then return True; /* Solution found with LS */
    $\Sigma_\Lambda^{Criteria} \leftarrow$ getClausesToSend($\Sigma_\Lambda$);
    $\Sigma_\Gamma \leftarrow \Sigma_\Gamma \cup \Sigma_\Lambda^{Criteria}$;
    $\Sigma_\Lambda \leftarrow \Sigma_\Lambda \setminus \Sigma_\Lambda^{Criteria}$; /* Sent clauses removed from LS solver database */
    if CDCLSolve($\Sigma_\Gamma$) = SAT then
        if $\Sigma_\Gamma = \Sigma$ then return True;
        /* CDCL solver has found a model to be used by LS solver */
        $\alpha \leftarrow$ getLiftedModel($\Sigma_\Gamma$);
        return isSatisfiable($\Sigma_\Lambda$, $\Sigma_\Gamma$, $\alpha$);
    else return False; /* CDCL solver proved sub-formula to be unsatisfiable */
end
```

3.4 The hinotos Algorithm

The hinotos algorithm implements the original hbisat algorithm (shown in Algorithm 1.1 without the underlined parts) as well as the optimizations proposed in the previous sections, and is shown in Algorithm 1.1. Again, Σ_Γ is initially empty, and α is first randomly initialized. The proposed optimizations do not affect soundness or completeness. The soundness and completeness of hbisat [5] allow establishing the following result.

Theorem 1. hinotos *is complete and sound for the satisfiability problem.*

4 Experimental Evaluation

4.1 Methodology

The empirical results presented in this section were obtained on servers running Red Hat Enterprise Linux WS release 4, with Intel Xeon 5160 Dual Core 3GHz processors and 4GB of RAM. For all experiments, the CPU time limit per instance was set to 1500 seconds.

hinotos[2] is implemented in C++ and can be configured to implement or not the optimizations proposed in Section 3. Hence, one possible configuration for hinotos corresponds to the actual hbisat algorithm [5]. However, the LS and the CDCL solvers used in hinotos are more efficient than the ones used in hbisat [5]. In hinotos the LS solver is adaptg2wsat+ [15] and the CDCL solver is Minisat2, whereas in the original hbisat the LS solver is

[2] hinotos complete documentation and binaries are publicly available on http://satstore.ecs.soton.ac.uk/software/hinotos.

`WalkSAT2004.11.15` [24] and the CDCL solver is `Minisat1.14` [4]). Observe that other alternative CDCL and LS solvers could be considered for `hinotos`.

Moreover, `hinotos` can be configured to run the following configurations (the representative letter for each option is highlighted with parenthesis):

(i)nverse order pure incremental: Represents a purely incremental (and not hybrid) SAT solver. No LS solver is used in this version. Clauses are always sent in the same order, inverse from the order of appearance in the formula, according to the first criterion described in section 3.3;

`(h)bisat` implementation: Represents the original `hbisat` algorithm as described in [5] and presented in section 2.2;

(m)inimum amount `hbisat`-like: Implements option `h`, and in addition integrates the criterion for moving clauses described in section 3.3;

`hi(n)otos` method: Implements option `h` and integrates the optimizations proposed in section 3.2;

(r)emoving+minimum amount `hinotos`-like: Corresponds to the combination of the two previous options (i.e. `m` and `n`).

These five configurations for the `hinotos` solver were compared against `Minisat` versions 1.14 and 2. The main purpose of the first configuration was to evaluate the usefulness of the LS SAT solver for identifying sets of clauses to move to the CDCL SAT solvers. The second configuration allowed benchmarking the implementation of `hbisat` in `hinotos`, thus confirming similar performance on instances for which results for `hbisat` are known. Finally, the last three configurations evaluate whether the proposed optimizations are effective in practice.

4.2 Benchmarks

With the objective of conducting a comprehensive evaluation of the different algorithms, a total of 602 problem instances were selected from the following classes of instances:

IBM Formal Verification Benchmarks. Problem instances from Bounded Model Checking considering different numbers of computation steps [29];

Pimag Problem instances from pipelined-machine-verification problems [16]. All Pimag instances are unsatisfiable;

Formal Verification of Processors (fvp). Formal verification of buggy variants of an out-of-order super-scalar processor [27].

Calysto (csv). Benchmarks generated by the `Calysto` static checker [1] for software verification.

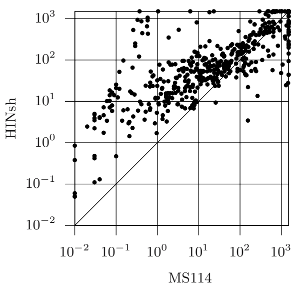
The results presented in this section extend and complete the preliminary experimental evaluation of [5], by considering 602 instances instead of the 24 studied in [5]. In order to reduce bias from too many instances from any of the classes considered, for classes ibm and csv only a subset of the available instances was considered, which was chosen arbitrarily. Nevertheless, for each of these classes a large number of instances was evaluated (respectively 198 and 152).

Table 1. Number of solved instances (and approximate average CPU time in seconds) for each configuration. CPU timeout is fixed to 1500 seconds.

	#Solved(Avg Time)							
CNF	Minisat		hinotos					#s/#t
	1.14	2	si	sh	sm	sn	sr	
c*	95(135)	**98(164)**	**98(194)**	97(189)	**98(202)**	96(188)	96(155)	102/126
f*	20(158)	**21(298)**	19(231)	18(161)	20(255)	**21(293)**	19(232)	21/42
g*	**42(139)**	41(144)	41(120)	38(150)	**42(186)**	41(199)	41(144)	42/42
pmg	157(139)	**160(177)**	158(179)	153(176)	**160(205)**	158(205)	156(161)	165/210
fvps	11(157)	**12(120)**	10(83)	**12(136)**	**12(150)**	10(135)	11(203)	14/20
fvpu	**19(98)**	18(39)	**19(83)**	18(75)	**19(100)**	18(44)	18(88)	20/22
fvp	30(119)	30(72)	29(83)	30(100)	**31(119)**	28(76)	29(131)	34/42
ibms	81(150)	84(192)	**87(185)**	78(234)	85(198)	83(222)	86(252)	93/93
ibmu	86(97)	85(109)	87(112)	83(176)	**88(138)**	85(164)	**88(170)**	88/88
ibm	167(97)	169(108)	**174(112)**	161(176)	173(137)	168(164)	**174(169)**	182/198
csvs	92(142)	95(127)	98(144)	117(197)	99(133)	**121(134)**	116(201)	129/129
csvu	8(85)	8(101)	8(87)	8(116)	8(115)	7(31)	**9(267)**	9/9
csv	100(138)	103(125)	106(140)	125(192)	107(132)	**128(128)**	125(206)	138/152
Alls	184(146)	191(155)	195(159)	207(207)	196(162)	**214(168)**	213(222)	237/242
Allu	270(105)	271(118)	272(125)	262(150)	**275(155)**	268(158)	271(136)	282/329
All	454(122)	462(133)	467(139)	469(175)	471(158)	482(163)	**484(174)**	519/602

4.3 Results

Table 1 summarizes the results obtained by all configurations of `hinotos` and the two versions of `Minisat`. Each configuration of `hinotos` is denoted by `sX`, where `X` is one of the possible configurations: `i`, `h`, `m`, `n`, and `r`. The first column contains the name of the class CNF formulas, where rows c*, f*, g* are problems in the Pimag category, rows fvps and fvpu (respectively ibms and ibmu, csvs and csvu) show the results on satisfiable and unsatisfiable instances of fvp (respectively ibm, csv) categories. The last column shows for each class of instances the total number of solved instances (by any solver) and the total number of instances. For each combination of tool configuration and instance category, the results shown are the number of solved instances followed (in parenthesis) by the average running time on *solved* instances. Analysis of table 1 allows concluding that no configuration, for either `hinotos` or `Minisat`, seems to give the best performance in terms of average run time for solved instances. Compared to `Minisat`, the performance improvements of `hbisat` are not clear, particularly for the first three classes of instances. For all classes of instances, the implementation of `hbisat` solves 7 more instances than `Minisat2`. Regarding the optimizations proposed in this paper, there are reasonable gains in terms of robustness, i.e. the number of instances solved. Indeed, configurations `sn` and `sr` solve significantly more instances than any other configuration, either for `Minisat` or for `hinotos`. A more detailed analysis for each class of problem instances indicates that `Minisat` and configuration `sm` perform better for the Pimag and fvp



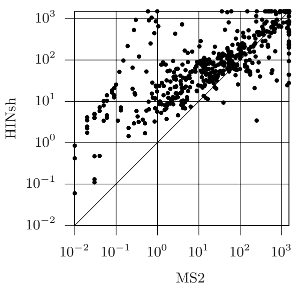
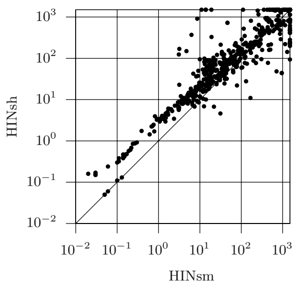


Fig. 2. Scatter plot: `hinotos-sh` vs. `Minisat1.14` (left hand side) and `Minisat2` (right hand side)



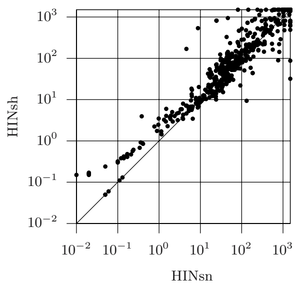
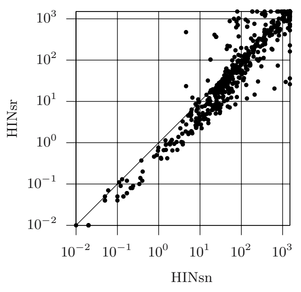


Fig. 3. Scatter plot: `hinotos-sh` vs. `hinotos-sm` (left hand side) and `hinotos-sn` (right hand side)

instances, whereas configurations `sn` and `sr` perform significantly better for the ibm and csv instances.

Figures 2 to 4 show scatter plots comparing different tool configurations. The names used are MS114, MS2, HINsh, HINsm, HINsn and HINsr, respectively for `Minisat1.14`, `Minisat2`, `hinotos-sh`, `hinotos-sm`, `hinotos-sn` and `hinotos-sr`. All run times are in seconds. Given the large number of configurations, only a subset of possible scatter plots is shown. Figure 2 evaluates `hinotos-sh` against `Minisat` versions 1.14 and 2. Albeit `hinotos-sh` aborts fewer instances than either version of `Minisat`, the plots indicate that this configuration is slower than either version of `Minisat` for most problem instances. Figure 3 shows the results for `hinotos-sh` against `hinotos-sm` and `hinotos-sn`. As can be concluded, for most instances, the run times for `hinotos-sm` and `hinotos-sn` are smaller than for `hinotos-sh`. This indicates that the optimizations proposed in this paper are in general effective, allowing smaller run times, besides being more robust. Finally, figure 4 shows scatter plots of `hinotos-sr` against `hinotos-sn` and `Minisat2`. As can be concluded, `hinotos-sr` in general has smaller run times than `hinotos-sn` (which has smaller run times than `hinotos-sh`), besides being more robust. The scatter plot on the right shows that



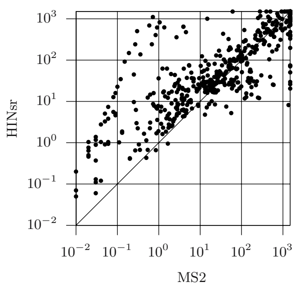


Fig. 4. Scatter plot: `hinotos-sr` vs. `hinotos-sn` (left hand side) and `Minisat2` (right hand side)

for most instances `Minisat2` has smaller run times than `hinotos-sr`. However, for a significant number of instances `hinotos-sr` has smaller run times than `Minisat2`. The plot also shows that several instances that `Minisat2` aborts, are solved by `hinotos-sr` in run times that do not exceed 100 seconds.

4.4 Analysis

The experimental results allow drawing a few conclusions. Existing versions of `Minisat` are less robust than most of the available configurations of `hinotos`. Out of 602 instances, the best version of `Minisat` (i.e. version 2) solves 462 instances. The more robust algorithm of `hinotos` solves 484 instances (almost 5% more instances solved than `Minisat2`). This result is quite significant in an industrial setting, and suggests that some of the configurations of `hinotos` should be considered for some classes of instances. On the other hand, a detailed analysis of the run times also suggests that for most instances, `Minisat2` has the smallest run time, and so should be the preferred solver. As a result, the experimental results given in this section suggest that an effective approach would be the following:

– Run `Minisat2` with a small CPU time limit, e.g. 100 seconds.
– If `Minisat2` does not solve the instance in the allowed CPU time limit, run one of the `hinotos` configurations (e.g. `hinotos-sr`) with a larger CPU time limit (e.g. 900 seconds).

The proposed configuration will be as robust as `hinotos-sr` and, for easier instances, run as effectively as `Minisat2`.

Moreover, from the results one might consider running the two solvers in parallel, without interaction between them. We conducted this experiment, and the results indicate that the LS solver is hardly ever useful. Moreover, the results show that running the two solvers in parallel is essentially equivalent to the original CDCL SAT solver. Finally, one should note that the algorithms proposed in `hinotos` complement the original SAT solvers, `Minisat2` and `Minisat1.14`. Together, `Minisat2`, `Minisat1.14` and `hinotos` can solve a significantly larger number of instances than any solver alone: 519 for all solvers compared with 484 for the best standalone solver, `hinotos` with configuration sr. These results suggest using a portfolio of SAT solvers similar to `SATzilla` [28] in industrial problems instances.

5 Related Work

Besides `hbisat` [5], other hybrid approaches have been proposed for SAT. A few are analyzed next. First, Mazure *at al.* [18] propose an approach essentially opposite to the one used in `hbisat` and `hinotos`. In this approach, a LS SAT solver is used to help a CDCL solver select the next decision variable. The motivation is that the use of LS SAT solver to guide the branching strategy can provide significant improvements. However, existing results are not promising [7]. Exploiting variable dependencies has been shown useful in local search algorithms for SAT. The approach of [11] proposes to extend the use of such dependencies by hybridizing `WalkSAT`, and the DPLL procedure `Satz`. At each node reached in the DPLL search tree to a fixed depth, the literal implication graph is constructed. Its strongly connected components are viewed as equivalency classes. Each one is substituted by a unique representative literal to reduce the constructed graph and the input formula. Finally, the implication dependencies are closed under transitivity. The resulting implications and equivalencies are exploited by `WalkSAT` at each node of the DPLL tree. The resulting algorithm [11] is an incomplete LS procedure that helps another LS SAT solver `WalkSAT` making better variable selections. Again, the approach, albeit efficient for some classes of satisfiable problem instances, is fundamentally different from `hbisat` and `hinotos` and unusable for unsatisfiable instances. Finally, Lardeux *at al.* [14] propose the GASAT algorithm, based on evolutionary algorithms and tabu search for SAT. The GASAT algorithm consists of a recombination stage based on a specific crossover and a tabu search stage. GASAT includes a crossover operator which relies on the structure of the clauses and a tabu search with specific mechanisms. The resulting algorithm is incomplete, and so not applicable to unsatisfiable problem instances.

6 Conclusions and Future Work

Hybrid incremental SAT solvers have recently been proposed as a possible approach for improving performance of CDCL SAT solvers [5]. Unfortunately, when an extended set of problem instances is considered, our experience is that the original `hbisat` algorithm is not effective when compared with different versions of `Minisat`. From the results, the conclusion is that for `hbisat` the CPU times increase, and the reduction in the number of aborted instances is negligible.

Motivated by these less positive results, this paper outlines a number of key optimizations to the original hybrid incremental SAT solver, `hbisat`. The experimental results, obtained on a wide range of problem instances, indicate that the proposed optimizations provide relevant performance improvements in terms of problem instances that can be solved, either with respect to `Minisat` (version 2) or `hbisat`. In terms of aborted instances, the best configuration of `hinotos` is significantly more effective than either version of `Minisat`, solving 5% more instances. In an industrial setting this is significant.

Despite the promising results, the experimental evaluation also suggests that no solver is the best option individually, and that three of the `hinotos` configurations should be considered as an alternative to `Minisat2` for some classes of problem instances. The analysis of the results also indicates that `Minisat2` usually performs better when a solution can be found in a reasonably small amount of time (e.g. 100 seconds). Hence, one strategy would be to consider running `Minisat2` as the first option and, in case no solution is found, considering one of the `hinotos` configurations. Given the experimental results, the most promising configurations are `hinotos-sr`, `hinotos-sn`, and `hinotos-sm`, all of which include improvements proposed in this paper. A fairly orthogonal approach, that for some classes of instances yields promising results and so should be considered, is the `hinotos-si` configuration, which is also proposed in this paper.

Future work will address portfolios of configurations, based on the ideas used in `SATzilla` for different SAT algorithms [28]. Moreover, further tuning of the algorithm's components should be considered, e.g. improve variable lifting procedure. Finally, it might be interesting to compare results with different CDCL solvers and different LS algorithms (e.g. based on dynamic clause weighting like SAPS, PAWS or DLM [26]). Another line of work is to automatically select some of the heuristic numbers used by the `hinotos` algorithm.

Acknowledgement. This work is partially supported by EU grants IST/033709 and ICT/217069, and by EPSRC grant EP/E012973/1. The reviewers provided insightful comments to an earlier version of the paper.

References

1. Babić, D., Hu, A.J.: Structural Abstraction of Software Verification Conditions. Computer-Aided Verification, 371–383 (2007)
2. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. Tools and Algorithms for the Construction and Analysis of Systems, 193–207 (1999)
3. Davis, M., Putnam, H.: A computing procedure for quantification theory. Journal of the ACM 7, 201–215 (1960)
4. Een, N., Sörensson, N.: An extensible SAT solver. In: International Conference on Theory and Applications of Satisfiability Testing, pp. 502–518 (2003)
5. Fang, L., Hsiao, M.S.: A new hybrid solution to boost SAT solver performance. In: Design, Automation and Testing in Europe Conference, pp. 1307–1313 (2007)
6. Fang, L., Hsiao, M.S.: Private communications (2007)
7. Ferris, B., Froehlich, J.: WalkSAT as an Informed Heuristic to DPLL in SAT Solving. Technical report, CSE 573: Artificial Intelligence (2004)
8. Gent, I.P., Walsh, T.: Towards an understanding of hill-climbing procedures for SAT. In: National Conference on Artificial Intelligence, pp. 28–33 (1993)
9. Goldberg, E., Novikov, Y.: BerkMin: a fast and robust SAT-solver. In: Design, Automation and Testing in Europe Conference, pp. 142–149 (2002)

10. Gomes, C.P., Selman, B., Kautz, H.: Boosting combinatorial search through randomization. In: National Conference on Artificial Intelligence, pp. 431–437 (1998)
11. Habet, D., Li, C.M., Devendeville, L., Vasquez, M.: A hybrid approach for SAT. In: International Conference on Principles and Practice of Constraint Programming, pp. 172–184 (2002)
12. Hoos, H.: An adaptive noise mechanism for WalkSAT. In: National Conference on Artificial Intelligence, pp. 655–660 (2002)
13. Hoos, H., Stützle, T.: Stochastic Local Search: Foundations and Applications. Morgan Kaufmann, San Francisco (2004)
14. Lardeux, F., Saubion, F., Hao, J.-K.: GASAT: A genetic local search algorithm for the satisfiability problem. Evolutionary Computation 14(2), 223–253 (2006)
15. Li, C.M., Huang, W.Q., Zhang, H.: Combining adaptive noise and look-ahead in local search for SAT. In: International Conference on Theory and Applications of Satisfiability Testing, pp. 121–133 (2007)
16. Manolios, P., Srinivasan, S.K.: A parameterized benchmark suite of hard pipelined-machine-verification problems. In: Advanced Research Working Conference on Correct Hardware Design and Verification Methods, pp. 363–366 (2005)
17. Marques-Silva, J., Sakallah, K.: GRASP: A new search algorithm for satisfiability. In: International Conference on Computer-Aided Design, pp. 220–227 (1996)
18. Mazure, B., Sais, L., Grégoire, E.: Boosting complete techniques thanks to local search methods. Annals of Mathematics and Artificial Intelligence 22(3-4), 319–331 (1998)
19. McAllester, D., Selman, B., Kautz, H.: Evidence of invariants in local search. In: National Conference on Artificial Intelligence, pp. 321–326 (1997)
20. McMillan, K.L.: Applying SAT methods in unbounded symbolic model checking. Computer-Aided Verification, 250–264 (2002)
21. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Engineering an efficient SAT solver. In: Design Automation Conference, pp. 530–535 (2001)
22. Prestwich, S., Lynce, I.: Local search for unsatisfiability. In: International Conference on Theory and Applications of Satisfiability Testing, pp. 283–296 (2006)
23. Ravi, K., Somenzi, F.: Minimal assignments for bounded model checking. Tools and Algorithms for the Construction and Analysis of Systems, 31–45 (2004)
24. Selman, B., Kautz, H., Cohen, B.: Noise strategies for improving local search. In: National Conference on Artificial Intelligence, pp. 337–343 (1994)
25. Selman, B., Levesque, H., Mitchell, D.: A new method for solving hard satisfiability problems. In: National Conference on Artificial Intelligence, pp. 440–446 (1992)
26. Thornton, J., Pham, D.N., Bain, S., Ferreira Jr., V.: Additive versus multiplicative clause weighting for SAT. In: National Conference on Artificial Intelligence, pp. 191–196 (2004)
27. Velev, M.N.: Using rewriting rules and positive equality to formally verify wide-issue out-of-order microprocessors with a reorder buffer. In: Design, Automation and Testing in Europe Conference, pp. 28–35 (2002)
28. Xu, L., Hutter, F., Hoos, H., Leyton-Brown, K.: SATzilla-07: The design and analysis of an algorithm portfolio for SAT. In: International Conference on Principles and Practice of Constraint Programming, pp. 712–727 (2007)
29. Zarpas, E.: Benchmarking SAT solvers for bounded model checking. In: International Conference on Theory and Applications of Satisfiability Testing, pp. 340–354 (2005)