Hints Revealed

Jonathan Kalechstain,[1,2] Vadim Ryvchin,[1] and Nachum Dershowitz[2]
`kalechstain@gmail.com vadimryv@gmail.com nachumd@tau.ac.il`

[1] Design Technology Solutions Group, Intel Corporation, Haifa, Israel
[2] School of Computer Science, Tel Aviv University, Tel Aviv, Israel

Abstract. We propose a notion of *hints*, clauses that are not necessarily consistent with the input formula. The goal of adding hints is to speed up the SAT solving process. For this purpose, we provide an efficient general mechanism for hint addition and removal. When a hint is determined to be inconsistent, a hint-based partial resolution-graph of an unsatisfiable core is used to reduce the search space. The suggested mechanism is used to boost performance by adding generated hints to the input formula. We describe two specific hint-suggestion methods, one of which increases performance by 30% on satisfiable SAT '13 competition instances and solves 9 instances not solved by the baseline solver.

1 Introduction

Modern backtrack search-based SAT solvers are indispensable in a broad variety of applications [3]. In a classical SAT interface, the solver is given one formula in conjunctive normal form (CNF) and determines whether it is satisfiable or not. Performance of SAT solvers has improved dramatically over the past years [15]. The main advancements came as result of developing new heuristics for existing conflict-driven clause-learning (CDCL) solver techniques, like deletion strategies, decision heuristics, and restart strategies (plus preprocessing and in-processing).

In this work, we propose and investigate a novel method for cutting the search space explored by the SAT solver so as to help it reach a solution faster. The idea is to add *hints*, clauses that are not necessarily "correct", in the sense that they are not necessarily implied by the original input formula.

We call our hint-addition platform HSat (Hint Sat), and present two variants that have been implemented in HaifaMUC [13]. HaifaMUC is an adaptation of MiniSat 2.2 [4], which we will henceforth refer to it as Base.

The addition of hints H to the original formula F creates an extended formula F'. Hints can, of course, affect the satisfiability of the formula. As long as H is implied by F, the extended formula F' will be equi-satisfiable with the original F (either both are satisfiable or neither is). This means that if F is satisfiable but F' is not, then there must be a contradiction between the added hints and the original formula.

In HSat, we try to solve only the extended formula F'. In case it is satisfiable, we are done, and the solver declares that the original formula was likewise satisfiable. Otherwise, the extended formula is unsatisfiable, in which case we need to understand whether the hints are the cause of unsatisfiability, that is, whether any hint is a necessary part of the proof of the empty clause. This is accomplished by an examination of the resolution graph that is built during the run of the solver on F'. In [14], the authors presented an efficient way (their "optimization **A**") of saving a partial resolution with respect to a given subset of input clauses. We use this ability to restrict tracking so that only the effects of hints are recorded in the partial graph. Marking clauses to track their origin is an old idea used in Chaff [8] and later reintroduced in [17], and is well adapted to cases when tracking of clauses is required. When the extended formula is unsatisfiable, we check the cone of the empty clause. If it includes a hint, then the status of the original formula remains unknown and additional operations are required (like deletion of the hints). Otherwise, the original formula is unsatisfiable, and we are done. Handling of inconsistent clauses was done in several other applications, like parallel solving [6,7]; our solution differs, having the ability to track the full effect of the partial resolution tree.

In case the result is unknown and the UNSAT core contains only one hint, an additional optimization can be made by using the UNSAT core of the partial resolution graph. Suppose the UNSAT core contains only hint h, then h must contradict F, and $\neg h$ is implied by F. As $\neg h$ is, in this circumstance, a set (conjunction) of unit clauses, each literal in h can be negated and added as a fact to F, which will increase the number of facts and reduce the search space to be explored. This optimization can be generalized to include all graph dominators in the partial resolution graph. (See Theorem 1 below.)

We introduce two heuristics for hint generation. The first, "Avoiding Failing Branches" (Afb), is a purely deterministic hint-addition method. The main idea behind it is the same idea that drives restarts in modern SAT solvers, namely, the possibility that the solver is spending too much time on "bad branches", branches that do not contain the satisfying assignment to the problem. Our motivation is to prevent the solver from entering branches that have already been explored. In our algorithm, we describe an *explored branch* that is a subset of decision variables. We pick the most conflict-active decisions and add a hint that explicitly precludes choosing that set again. In this approach, we keep a score for each literal. The score is boosted every time a clause containing it participates in a conflict. The literals with the highest scores are added to a hint in their negated form. The hint is then added right after a restart, and the same set of active decision variables will never be chosen unless the hint is removed. This approach leads to significantly improved solver times for satisfiable instances.

A second heuristic, "Randomize Hints" (Rh), draws a given number of random assignments, and tries to create a set of hints that will contradict the instance. When the solver concludes unsatisfiability, all dominators of the partial resolution graph are extracted, and all literals in all dominators are added as facts in their negated form.

We continue in the next section with the formalization and various preliminaries. Section 3 presents the HSat algorithm, and, in Sect. 5, we demonstrate its correctness. The two heuristic hint-generation methods of Sect. 4 are empirically evaluated in Sect. 6. We conclude and discuss future work in Sect. 7.

This paper contains several contributions. An efficient generic mechanism is introduced to add hints, the goal of which is to speed up the solver. It is based on the ability to remove clauses and all the facts derived from them. In HSAT, we use the partial resolution graph of BASE to remove the hints and their effect in case of an unsatisfiable conclusion. In [9,11,10] and later in [14], it was shown that the alternative, using selector variables for clause removal [5,12], is inferior to the use of the resolution graph. We extend the path-strengthening technique published in [10]. Instead of using only immediate children of the removed clauses, we use all dominators in the partial resolution graph provided in BASE. We introduce two algorithms for hint generation, one of them (AFB) increasing performance for satisfiable instances by 19–30%.

2 Preliminaries

We presume some basic knowledge of the Boolean Satisfiability Problem and CDCL SAT solvers [3]. Let φ be a CNF formula $c_1 \wedge c_2 \cdots \wedge c_m$. We write $c_i \in \varphi$ if $\varphi = c_1 \wedge \cdots \wedge c_i \wedge \cdots \wedge c_m$. Each clause $c = \ell_1 \vee \ell_2 \vee \cdots \vee \ell_k$ is a disjunction of literals, and each literal ℓ_i is either a variable v or its negation $\neg v$. We write $\ell_j \in c_i$ if $c_i = \ell_1 \vee \cdots \vee \ell_j \vee \cdots \vee \ell_k$. In what follows, V denotes the set of variables occurring in φ, and $n = |V|$.

For two clauses $c_i = v \vee c$ and $c_j = \neg v \vee c'$, both involving the same variable $v \in V$, their binary *resolvent* is

$$Resol(c_i, c_j) \triangleq c \vee c' \, .$$

A conflict occurs when several solver decisions and subsequent implications result with a clause being unsatisfiable. In CDCL SAT solvers, a clause preventing the last conflicting set of decisions is created and added; it is referred to as a *conflict clause*. In [8], it was shown that the best clause is the one created by finding the cut in the implication graph that includes the Unique-Implication-Point (UIP) closest to the conflict. That cut corresponds to a number of binary resolutions performed on clauses that are inside the cut or intersect it. For example, Fig. 1 illustrates the cut and the clauses c_4, c_5, c_6 that participated in the resolutions that derived the conflict.

If φ is a formula and H is a set of hint clauses, then by $\varphi \wedge H$ we mean their conjunction: $\varphi \wedge \bigwedge_{h \in H} h$, which we will call a *hint-extended formula*.

In HSAT, we use a *resolution graph* to determine why $\varphi \wedge H$ is unsatisfiable, when it is, by extracting the *UNSAT core*.

Definition 1 (Hyper-Resolution). *Let $c_1, c_2, \ldots, c_i$ be all the clauses (from the implication graph) that participated in the binary resolutions that created the first UIP conflict clause U. The Hyper-Resolution function*

$$Hyper(c_1, c_2, \ldots, c_i) \triangleq U$$

yields that resulting conflict clause U.

$$c_1 = (\sim v_1 \lor v_2)$$
$$c_2 = (\sim v_1 \lor v_3 \lor v_9)$$
$$c_3 = (\sim v_2 \lor \sim v_3 \lor v_4)$$
$$c_4 = (\sim v_4 \lor v_5 \lor v_{10})$$
$$c_5 = (\sim v_4 \lor v_6 \lor v_{11})$$
$$c_6 = (\sim v_5 \lor \sim v_6)$$
$$c_7 = (v_1 \lor v_7 \lor \sim v_{12})$$
$$c_8 = (v_1 \lor v_8)$$
$$c_9 = (\sim v_7 \lor \sim v_8 \lor \sim v_{13})$$

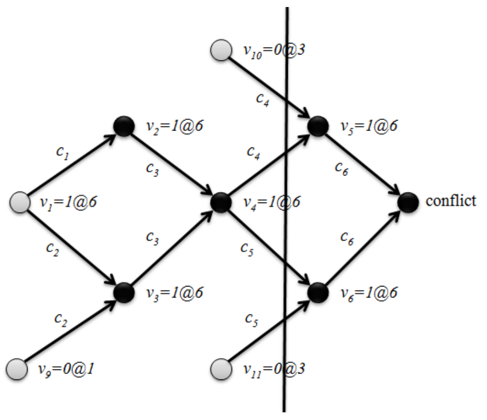

Fig. 1. Conflict analysis graph. The grey nodes represent decision variables while the black nodes represent propagated values. The vertical line is the first UIP cut.
Writing $v_i = a@b$ means that v_i was assigned to a at decision level b.

As mentioned in Sect. 1, we use a partial resolution graph to generate hints. This graph is used to determine whether there exists a directed path from H to an empty clause.

Definition 2 (Resolution Graph). *The Resolution Graph* $G = (V, E)$ *is defined recursively as follows:*

$$\begin{aligned} V &:= \varphi \cup H \cup \{Hyper(c_1, \ldots, c_m) \mid c_1, \ldots, c_m \in V\} \\ E &:= \{(c_i, Hyper(c_1, \ldots, c_i, \ldots, c_m)) \mid c_1, \ldots c_m \in V\} \,. \end{aligned}$$

In words, the vertices are the initial clauses and hints closed under hyper-resolution and the edges point from participating clauses to their hyper-resolvent.

Determining whether a path exists from H to the empty clause is possible by saving only the part relevant to hints. The partial resolution graph will consist only of hints or conflict clauses that were derived by some hint. To do so, we start just with the hints and define the relevant hint-based *Partial Resolution Graph* as follows:

Definition 3 (Partial Resolution Graph). *The Partial Resolution Graph* $G_P = (V_P, E_P)$ *is defined recursively as follows:*

$$\begin{aligned} V_P &:= H \cup \{Hyper(c_1, \ldots, c_i, \ldots, c_m) \mid c_i \in V_P,\ c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_m \in V\} \\ E_P &:= \{(c_i, Hyper(c_1, \ldots, c_i, \ldots, c_m)) \mid c_i \in V_P,\ c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_m \in V\} \,. \end{aligned}$$

In words, the vertices are the hints closed under hyper-resolution and the edges point from participating clauses to their hyper-resolvent. Figure 2 contains an

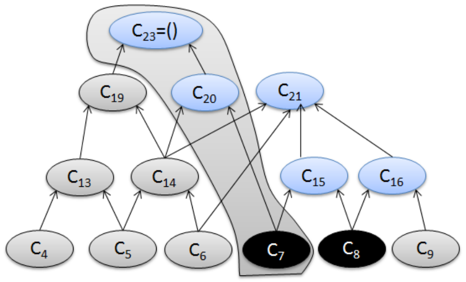

Fig. 2. Resolution graph. Black nodes are the set H. Blue nodes are the hyper resolvents of V_P. The grey nodes are the nodes in $V \setminus V_P$.

example illustrating Definitions 2,3. The nodes c_4, c_5, c_6, c_9 form the set φ, while c_7, c_8 are the hints in H. The entire graph represents G while the black and blue nodes and the edges between them are the restricted graph G_P.

Having defined the partial resolution graph, we are interested in isolating the proof of unsatisfiability. To do so, we define the UNSAT core (UC) of a resolution graph.

Definition 4 (UNSAT Core). *The UNSAT core is a subset UC of $\varphi \cup H$ that is backward reachable from the empty clause in G.*

We are interested in finding that part of UC that is relevant to the hints:

Definition 5 (Relevant UNSAT Core). *The Relevant UnsatCore is the intersection $RC = H \cap UC$.*

We refer to the set of all dominator points (a vertex that lies along every path) between RC and the empty clause in an UNSAT proof as $Dominator_{RC}$.

The negation $\neg h$ of a clause $h = \ell_1 \vee \ell_2 \cdots \vee \ell_k$ is the set (conjunction) of its negated literals $\neg\ell_1 \wedge \neg\ell_2 \cdots \wedge \neg\ell_k$, viewed as unit facts.

3 Hint Addition

We proffer a general platform for adding clauses without worrying that they might be inconsistent with the input formula. These "hint" clauses can be created using prior knowledge about the formula's origins or from information garnered during SAT solving, as explained in Sect. 4. Our solution enjoys several benefits:

1. No additional literals are added.
2. We delay the effect of hints by using techniques from HaifaMUC as described in [14].
3. "Bad" hints, hints participating in the empty clause derivation, are used for search space reduction.

We use a resolution-graph-based solution to avoid the need for extra literals and to enable further optimizations in case the extended formula is unsatisfiable on account of the hints. In addition, we want to prevent any aggressive intervention of hints in the SAT solver's solution process, by using hints only when necessary, which is achieved by delaying their use. We discriminate in favor of the use of ordinary clauses because conflicts derived from hints are not necessarily consistent with the formula. The same motivation underlies modern Minimal Unsatisfiable Set (MUS) and Group Minimal Unsatisfiable Set (GMUS) solvers, which prefer to use clauses already known to be in the minimal core, to keep that as small as possible. Because of the similarity between hints and core clauses in MUS and GMUS solvers, we base our solver on HaifaMUC and use the optimization techniques described in [14]. These techniques allow us to prioritize ordinary clauses over hints and therefore reduce the run-time effect of hints.

The optimizations relevant to hints are the following:

1. Maintain only the partial resolution proof of clauses derived from the added hints. This prevents the keeping of the whole resolution proof in the memory and significantly reduces the memory footprint of the solver.
2. Selective clause minimization. Clause minimization [2,16] is a technique for shrinking conflict clauses. If the learned clause is not derived from the hints, then during shrinking we prevent the use of hints in the minimization. The result is that no additional dependencies on hints are added even at the expense of longer learned clauses.
3. Postponed propagation over hints. This optimization is performed by changing the order of BCP (Binary Constraint Propagation). BCP first runs over ordinary (non-hint) clauses, and only if no conflict is found does it run over hints. The motivation is to prefer conflicts caused by ordinary clauses.
4. Selective learning of hints and selective backtracking. Both optimizations change the learning scheme by reducing the number of clauses effected by hints in case an ordinary clause can be learned.

We denote these optimization techniques collectively as HMucOpt.

One of the benefits of using a resolution-graph method is the availability of clause relation information, which can be used in case the extended formula is unsatisfiable on account of hints. In [10], a *path strengthening* technique was presented in relation to the MUS problem solution. It uses the partial resolution graph and is used to check whether a clause c is part of the MUS. Checking whether $c \in$ MUS can be done by checking if the formula is unsatisfiable without using c. If it is, then c cannot be part of the minimal core. To speed up the SAT solver run, the negation of the clause is added to the SAT Solver as assumptions. Path strengthening extends this set of assumptions by analyzing the resolution graph. If c has only one derived clause in the cone of the empty clause, then the literals of this clause are added as assumptions as well. This operation is performed recursively until a clause with more than one child is reached. In HSat, we extend this by using all dominators between the hint clause and the empty clause in the partial resolution graph.

Algorithm 1 HSat– Solves an extended formula, negates dominators and cleans hints' effects.

```
Input:  instance – Boolean formula in CNF form
        H – Initial set of Hints (in our case $\emptyset$)
Output: SAT or UNSAT (ignore TIMEOUT)
 1: while TRUE do
 2:     $model := Solve(instance \wedge H)$        ▷ New $h \in H$ can be added in $Solve()$
 3:     if $model \neq$ NULL then
 4:         return SAT                              ▷ We have the model
 5:     else
 6:         $RC := GetRC()$
 7:         if $RC.Size() = 0$ then
 8:             return UNSAT
 9:         else
10:             $Dominator_{RC} := GetDominators(RC)$
11:             for each $D \in Dominator_{RC}$ do
12:                 $instance := instance \wedge Negate(D)$
13:             for each $c_i \in$ RC do
14:                 $RemoveClauses(c_i)$
```

Algorithm 1 introduces the general workflow of HSat. Operation *Solve*() is a modification of a generic SAT solver with several additions. First, it allows the addition of new hints and produces a partial resolution in case those hints are added. In addition, *Solve*() contains an implementation of HMucOpt. Operation *Solve*() can return satisfiable or unsatisfiable. In the satisfiable case, we are done, as the solver found a satisfying assignment to the formula. In case the result is unsatisfiable, we check the *RC* (UNSAT core of hints) created by the hints. The extraction of *RC* is performed using *GetRC*(). If *RC* is empty, then the solver found a proof of the empty clause without relying on hints, so the original formula is unsatisfiable. Otherwise, we find all dominators of the *RC* using *GetDominators*(). (See Alg. 2 and the next paragraph.) For each dominator, we add its negation via *Negate*() to the input formula and create a new *instance*, which goes back to *Solve*(). Before the next call to *Solve*(), we clean the effect of hints in G_P by means of *RemoveClauses*(). The correctness of Alg. 1 is justified by the observations of Sect. 5. As mentioned already, for *Solve*() we use a modification of Base, so all the optimizations HMucOpt are used, as was introduced in [14]. This way, we ensure an increased chance of finding the solution without hints if such a solution is easy to find.

The operation *GetDominators*() gets all nodes $v \in V_P$ such that all paths from H to the empty clause go through v. At first, we save all nodes from *RC* in a list called *workList*. The algorithm iterates until *workList* is empty. We get from the list some $v \in V_P$ that has all parent marked using *GetAllParentsMarked*(*workList*). Note that since *RC* has no parents, all members of *RC* have all parents marked. If the size of *workList* is 1, then v is a dominator and we push it into $Dominator_{RC}$. We mark v using *Mark*(v) and

Algorithm 2 *GetDominators*() – Gets all dominators in G_P. This set will be negated in Alg. 1

```
Input:   $G_P$ – The Partial Resolution Graph
Input:   $workList$ – list of vertices. Initially set to $RC$
Output:  $Dominator_{RC}$ – The dominators with respect to $G_P$
1:  while $workList.Size() > 0$ do
2:      $v := GetAllParentsMarked(workList)$
3:      if $workList.Size() = 1$ then
4:          $Dominator_{RC}.Push(v)$                    ▷ A dominator
5:      $Mark(v)$                                       ▷ $v$ now marked
6:      for each $u \in Children(v)$ do
7:          if $\neg IsMarked(u)$ then
8:              $workList.Push(u)$
9:      $workList.Remove(v)$
10: return $Dominator_{RC}$
```

push all its unmarked children into *workList*. Note that the empty clause is a child too.

4 Hint Creation Algorithms

Heuristics for hint generation can vary from completely random selection to a purely deterministic selection algorithm.

4.1 Avoiding Failing Branches

In this section, we present a deterministic heuristic for hint creation based on the restart strategy and conflicts. We call this heuristic *Avoiding Failing Branches* (Afb). The idea is to track the most conflict-active decisions in the explored branch and add a hint that explicitly prevents choosing that set again. If a restart took place, it is reasonable to assume heuristically that the last explored branch is less likely to contain the satisfying assignment.

In Afb, we keep an array of variable activity to determine the most conflict-active decisions. When the solver encounters a conflict, we update the scores of all variables responsible for the conflict. We will explain what “responsible” means shortly.

The decision to add hints is taken upon backtracking. If the backtracking is actually a restart, then the most active literals are chosen to participate in a hint, which is added right after the restart. Because the literals are added in their negated form, all explored branches containing the set of literals in the hint will not be re-explored.

In Alg. 3, which is implemented within the function *Analyze*() of MiniSat 2.2 [4], we update the score of the variables participating in a conflict. For this purpose, we keep an array of variables (*variableScores*), which is updated for all

Algorithm 3 Update the score of a variable after a conflict. The score is updated for all decision variables in the first UIP, and for all variables in the reason clause for non-decision variables.

```
 1: $Analyze()$ {
 2: $\cdots$
 3: $U := ComputeFirstUip()$
 4: $\cdots$
 5: for each $\ell \in U$ do
 6:     $v := Var(\ell)$                                   ▷ $v$ is the variable of $\ell$
 7:     if $DecisionVariable(v)$ then
 8:         $variableScores[v] := variableScores[v] + 1$
 9:     else
10:         $c_v := Reason(v)$
11:         for each $\ell' \in c_v$ do
12:             $v' := Var(\ell')$
13:             $variableScores[v'] := variableScores[v'] + 1$
14: ...
15: }
```

literals that are in the first UIP (U) computed in *ComputeFirstUip*(). We then iterate all variables $v \in U$. If v is a decision variable (*DecisionVariable*(v)), we increment its score by one. Otherwise, we take the reason for v being assigned ($c_v :=$ *Reason*(v)) and increment the score for all variables in c_v.

In Fig. 1, the first UIP node is $U = v_{10} \vee \neg v_4 \vee v_{11}$. Alg. 3 will first compute U and iterate through all its literals. The scores of decision variables v_{10}, v_{11} are increased in line 8; v_4 is not a decision variable, so its reason, c_3, is computed in line 10. The score of variables v_2, v_3, v_4 is increased in line 13.

The hints are added in the function *CancelUntil*() of MiniSat 2.2 [4]. If a restart is decided upon, we use the information acquired by Alg. 3 to choose the most active literals to participate in the hint. A literal ℓ is chosen to participate if *variableScores*[*Var*(ℓ)] is greater than some threshold θ. The integer *conflict* is the number of conflicts since *Solve*() was called. Three magic numbers, $\alpha \in [0..1]$, $x \in \mathbb{N}$, $y \in \mathbb{N}$, also appear in Alg. 4. They are used in the following fashion:

1. A literal ℓ is added to the hint if $variableScores[Var(\ell)] > \alpha \times conflicts = \theta$.
2. We observed that, as time passes, it's advisable to increase θ.
3. Parameter x was added as a minimal threshold to prevent adding hints too "quickly". The idea is to prevent hints from being used when easy instances are solved.
4. Parameter y is used to ensure that new hints are not too small. Small hints can be too influential in the search procedure.

We maintain a vector of literals, *hint*, to store the clause that might form the future hint. Function *AddClause*() adds the hint to the input instance.

Algorithm 4 AFB hint addition – Adds a hint built of all negated literals with a score exceeding θ.

```
1: $CancelUntil(backtrackLevel)\{$
2: if $backtrackLevel > 0$ then                          ▷ This is not a restart
3:     Performing backtracking until $backtrackLevel \ldots$
4:         Upon freeing variable $v$:
5:             $variableScores[v] := 0$
6:         $\ldots$
7: else                                                  ▷ This is a restart
8:     if $conflicts > x$ then
9:         for each decision variable $v$ with decision $\ell$ do
10:            if $variableScores[v] > \alpha \times conflicts$ then
11:                $hint.Push(\neg l)$
12:    for each variable $v$ with decision $\ell$ do
13:        $variableScores[v] := 0$
14:    Perform backtracking until $backtrackLevel = 0 \ldots$
15:    if $hint.Size() > y$ then
16:        $AddClause(hint)$
17:    $hint.Clear()$
18: ...
19: }
```

4.2 Randomized Hints

We introduce next a completely random selection algorithm for hint creation, based on random assignments and satisfiability checking. We call this heuristic *Randomize Hints* (RH). In this algorithm, we use random assignments to see if we can learn literals that are likely untrue, that is, if chosen, a conflict is reached. We add these literals to form a new hint, that will hopefully lead the solver to an unsatisfiable conclusion. This hint is then negated, and the explored search space is reduced.

The randomized hint is created before HSAT is called. First, k random assignments are drawn, each with uniform distribution over $\{0,1\}^n$. These assignments are then checked on every clause. If some clause is unsatisfied, we bump the grade of all literals in the clause. We keep a vector of grades, $literalsGrades()$, and track the maximal graded literals that will be chosen to participate in the hint. We encourage the solver to pick the literals of the hint as decisions by increasing the activity of the variables involved in MINISAT's $VarBumpActivity(v)$.

The following functions and variables are used in Alg. 5 for random hints:

1. $DrawRandomAssignments(num)$ creates num random assignments.
2. $ClauseSatisfied(c, \sigma)$ returns true iff $\sigma(c) = \text{TRUE}$.
3. $PopMax()$ returns and removes the literal with the highest score.

Algorithm 5 Create randomized hints – draws random assignments and boosts score for all literals in a clause unsatisfied by an assignment. The literals with the highest scores are chosen to participate in hints.

```
Input: $sizeOfHint$ – Size of the hint
Input: $assignments$ – Number of assignments to draw
1: $DrawRandomAssignments(assignments)$
2: for each Assignment $\sigma$ do
3:     for each Clause $c$ do
4:         if $\neg ClauseSatisfied(c, \sigma)$ then
5:             for each literal $\ell \in c$ do
6:                 $literalsGrades[l] := literalsGrades[l] + 1$
7:                 $VarBumpActivity(Var(\ell))$
8: for $i \in [0..sizeOfHint - 1]$ do
9:     $hint[i] := literalsGrades.PopMax()$
10: $AddClause(hint)$
11: $hint.Clear()$
```

5 Theoretical Basis

For completeness, a few observations are in place, which should serve to convince readers that correctness is being maintained.

Proposition 1. *For any formula φ, a set of hints H and assignment $\sigma : V \to \{0,1\}$ of truth values to the variables of $\varphi \wedge H$,*

$$\sigma(\varphi \wedge H) \Rightarrow \sigma(\varphi)\,.$$

Proposition 2. *For any formula φ and set of hints H,*

$$\varphi \wedge H \in UNSAT \Rightarrow \varphi \wedge \neg H \equiv \varphi\,.$$

By $\neg H$, we mean $\bigvee_{h \in H} \neg h$.

Proof. If $\varphi \wedge H \in UNSAT$, then $\neg(\varphi \wedge H)$, which is equivalent to $\varphi \Rightarrow \neg H$. ■

From Proposition 2, we establish the following:

Proposition 3. *Given $\varphi \wedge H \in UNSAT$ and $|H| = 1$ where $h = \ell_1 \vee \ell_2 \cdots \vee \ell_k$*

$$\varphi \wedge \neg\ell_1 \wedge \neg\ell_2 \wedge \cdots \wedge \neg\ell_k \equiv \varphi\,.$$

This observation is critical for HSat. In this case, k new facts are learned, which helps reduce the fraction of the search space that gets explored.

As mentioned earlier, this idea can be generalized to include all dominators.

Theorem 1. *If $\varphi \wedge H$ is unsatisfiable, then $\varphi \equiv \varphi \wedge \neg D$ for every $D \in Dominator_{RC}$.*

Proof. Since $D \in Dominator_{RC}$, it is sufficient to prove $\varphi \wedge D \in UNSAT$. By Proposition 2, $\varphi \equiv \varphi \wedge \neg D$. ■

Table 1. AFB performance results for SAT 2013: satisfiable (left) and unsatisfiable (right) instances. Run-time is in minutes.

SAT	BASE	AFB
Run-time	990	**697**
Unsolved (by one)	9	**4**

UNSAT	BASE	AFB
Run-time	**727**	779
Unsolved (by one)	**2**	3

6 Experimental Results

6.1 AFB Results: SAT 2013

We compare now the performance of HSAT, with and without heuristic AFB. We find that hints have a positive effect for satisfiable instances but cause a moderate degradation for unsatisfiable ones. The positive results for satisfiable instances are in line with our presumption that, if a restart takes place, it is heuristically likelier that the satisfying assignment to the problem lies on another branch.

We ran over 150 *satisfiable* instances from SAT 2013, but the results reported below refer only to the 113 that were fully solved by at least one solver within half an hour. All of the instances are publicly available at [1]. We implemented all the algorithms in BASE [14], which is built on top of MINISAT 2.2 [4]. The code is public and available at [13]. For the experiments, we used machines running Intel® Xeon® processors with 3Ghz CPU frequency and 32GB of memory.

Table 1 displays a 30% improvement in overall runtime for satisfiable instances. Furthermore, there are 9 instances solved by AFB that are not solved by the base solver, compared to 4 instances solved by BASE but not by AFB.

In addition, 130 *unsatisfiable* instances from SAT 2013 were tested; the reported results refer only to the 60 that were fully solved by at least one solver within the 30-minute time limit. Table 1 shows a 7% degradation in overall runtime for unsatisfiable instances.

Figure 3 presents BASE vs. AFB. The diagonal $y = x$ emphasizes the superiority of AFB. Figure 4 presents the time differential between BASE and AFB. On average, AFB solves one of these problem instances $2\frac{1}{2}$ minutes faster than the baseline. The graphs refer to satisfiable instances only.

Figure 5 shows three curves, plotted at one minute intervals. The lower curve (A) is the percentage of instances solved by BASE and AFB both; the middle (B) is the percentage of instances solved by BASE the upper (C) is the percentage solved by either one. The gap $B - A$ represents the percentage of instances solved by BASE but not by AFB; $C - B$ represents the percentage solved by AFB but not by BASE. Notably, $C - B$ is consistently larger than $B - A$.

We observe that the positive effect of AFB is due to successful branch cutting by hints and not because of HSAT's ability to negate dominators. Most of the hints added did not contradict the instance, so HSAT's UNSAT core abilities were not helpful in AFB.

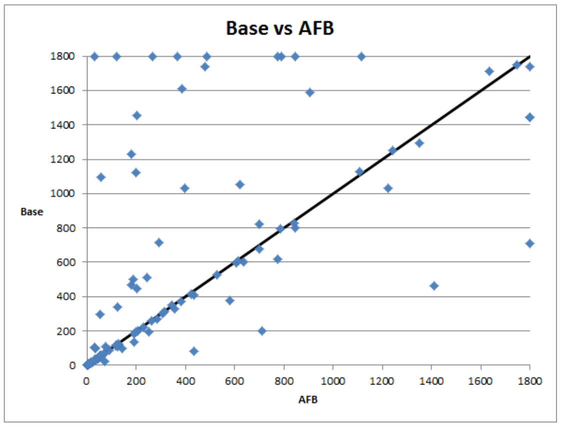

Fig. 3. Comparing AFB to BASE.

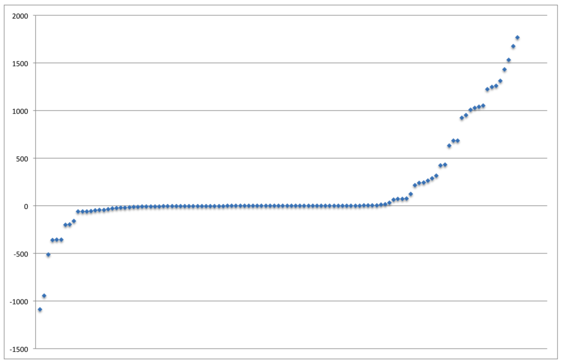

Fig. 4. The time difference (in seconds) between BASE and AFB

For the SAT 2013 benchmark, we also measured the average size of hints (number of literals participating in a hint), the average number of hints per instance, and the number of dominators found in all instances:

	SAT	UNSAT
Hint average size	34	43
Hints per instance	0.84	1.16
Dominators	2	15

Hints were used in 34% of the satisfiable instances and 39% of the unsatisfiable cases.

6.2 AFB Results: SAT 2014

We used the same configuration when testing AFB on satisfiable instances from the SAT 2014 competition. Table 2 shows a 19% improvement in overall runtime

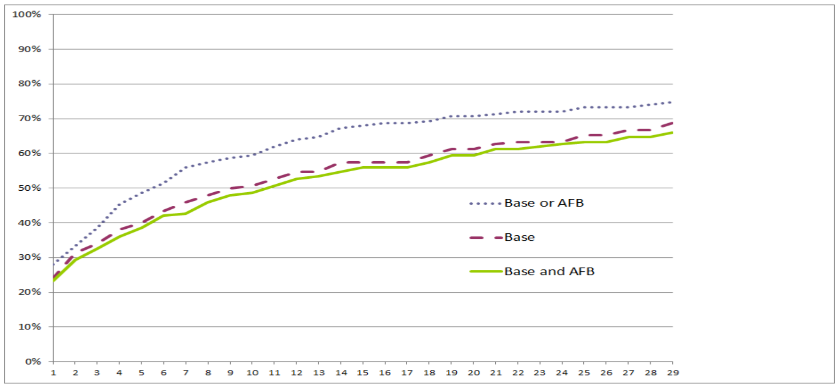

Fig. 5. Comparing percentage of instances solved by BASE and AFB.

Table 2. AFB performance results for SAT 2014: satisfiable instances. Run-time is in minutes.

SAT	BASE	AFB
Run-time	833	**681**
Unsolved (by one)	9	**2**

for satisfiable instances. Furthermore, there were 9 instances solved by AFB that were not solved by the base solver, compared to 2 instances solved by BASE but not by AFB. These results refer only to the 98 instances that were fully solved by at least one solver within 30 minutes.

6.3 RH Results: SAT 2013

The same configuration as in Sect. 6.1 was used for the RH heuristic on satisfiable instances from SAT 2013, and the same instances were tested. The results reported below refer only to the 116 instances that were fully solved by at least one solver within 30 minutes. Table 3 shows a 8% improvement in overall runtime for satisfiable instances.

We were admittedly surprised to see that satisfiable instances were solved faster because of "good" hints, hints that do not contradict the input. We were surprised because we tried to build hints that would contradict the input and have the negation of dominators drive the solution.

In addition, 130 *unsatisfiable* instances from SAT 2013 were tested; the results below refer only to the 60 that were fully solved by at least one solver within the 30-minute time limit. Table 3 shows a 15% degradation in overall runtime for unsatisfiable instances.

Combining the two heuristics, AFB and RH, as though they would run in parallel for half an hour on the SAT 2013 benchmark, we obtain 16 SAT instances

Table 3. Rh performance results: satisfiable (left) and unsatisfiable (right) instances. Run-time is in minutes.

SAT	Base	Rh
Run-time	1080	**988**
Unsolved (by one)	**12**	**12**

UNSAT	Base	Rh
Run-time	**757**	888
Unsolved (by one)	**3**	9

that are solved for which Base times out, versus 2 that only Base solves, and 5 UNSAT instances that Base fails on, versus 3 only by Base.

7 Discussion

We have introduced a new paradigm and platform, called HSat, with which one can speed up SAT solving by means of added clauses. It enables the addition of "hint" clauses that are not necessarily derivable from the original formula but which can nevertheless help the solver reach a solution faster. HSat avoids the addition of new literals, using instead a partial resolution graph to keep track of the effect of hints. We have seen that the Afb hint heuristic, which causes the prover to avoid retaking the most conflict-active decisions, outperforms the (hintless) baseline system and introduces a significant improvement in the solver core. On a benchmark of 280 instances, 150 of which are satisfiable: Afb achieved a 30% runtime improvement over the baseline and solved 9 instances not solved by the baseline prover.

Though these results are very encouraging, we have reason to believe that future work can lead to further improvements. For example, we tried to increment conflict decision variable scores by an amount that is inversely proportional to its depth in the proof tree, so those closer to the root (which have greater impact) get greater weight. This approach did not work for the thresholds we looked at, but might work for others. Another example is that our hint heuristics do not work well for unsatisfiable instances, the main reason being that there are usually no dominator clauses, in which case unsatisfiability does not drive the subsequent search very well. In this case, the incremental running of Alg. 1 just adds overhead. An interesting avenue for research would be to design hints that create multiple dominators or that lead the solver to a contradiction faster.

There are an endless number of ways to create hints, and many places in the process to add them; so far we have only explored a few options. It is likely that there remain even more interesting ways to create good hints for satisfiable instances and, hopefully, for unsatisfiable ones, too.

Acknowledgments. The suggestion to exploit hints originated with Ofer Strichman. The authors would like to thank Alex Nadel and Ofer Strichman for their advice and reading of a draft of this paper.

References

1. SAT competition 2013, http://satcompetition.org/2013/downloads.shtml
2. Beame, P., Kautz, H.A., Sabharwal, A.: Towards understanding and harnessing the potential of clause learning. CoRR abs/1107.0044 (2011), http://arxiv.org/abs/1107.0044
3. Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (February 2009)
4. Eén, N., Sörensson, N.: An extensible SAT-solver [extended version 1.2]. In: Proceedings of Theory and Applications of Satisfiability Testing (SAT). Lecture Notes in Computer Science, vol. 2919, pp. 512–518. Springer (2003)
5. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electr. Notes Theor. Comput. Sci. 89(4), 543–560 (2003), http://dx.doi.org/10.1016/S1571-0661(05)82542-3
6. Hyvärinen, A.E.J., Junttila, T.A., Niemelä, I.: Grid-based SAT solving with iterative partitioning and clause learning. In: Lee, J.H. (ed.) Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6876, pp. 385–399. Springer (2011), http://dx.doi.org/10.1007/978-3-642-23786-7_30
7. Lanti, D., Manthey, N.: Sharing information in parallel search with search space partitioning. In: Nicosia, G., Pardalos, P.M. (eds.) Learning and Intelligent Optimization - 7th International Conference, LION 7, Catania, Italy, January 7-11, 2013, Revised Selected Papers. Lecture Notes in Computer Science, vol. 7997, pp. 52–58. Springer (2013), http://dx.doi.org/10.1007/978-3-642-44973-4_6
8. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of the 38th Design Automation Conference (DAC), Las Vegas, NV. pp. 530–535. ACM (Jun 2001), http://doi.acm.org/10.1145/378239.379017
9. Nadel, A.: Boosting minimal unsatisfiable core extraction. In: Bloem, R., Sharygina, N. (eds.) Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design (FMCAD), Lugano, Switzerland. pp. 221–229. IEEE (Oct 2010), http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5770953
10. Nadel, A., Ryvchin, V., Strichman, O.: Efficient MUS extraction with resolution. In: Formal Methods in Computer-Aided Design (FMCAD), Portland, OR. pp. 197–200. IEEE (Oct 2013), http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6679410
11. Nadel, A., Ryvchin, V., Strichman, O.: Accelerated deletion-based extraction of minimal unsatisfiable cores. JSAT 9, 27–51 (2014), https://satassociation.org/jsat/index.php/jsat/article/view/116
12. Oh, Y., Mneimneh, M.N., Andraus, Z.S., Sakallah, K.A., Markov, I.L.: AMUSE: A minimally-unsatisfiable subformula extractor. In: Malik, S., Fix, L., Kahng, A.B. (eds.) Proceedings of the 41st Design Automation Conference (DAC), San Diego, CA. pp. 518–523. ACM (Jun 2004), http://doi.acm.org/10.1145/996566.996710
13. Ryvchin, V.: HaifaMUC, https://www.dropbox.com/s/uhxeps7atrac82d/Haifa-MUC.7z
14. Ryvchin, V., Strichman, O.: Faster extraction of high-level minimal unsatisfiable cores. In: Proceedings of the 14th International Conference on Theory and Appli-

cation of Satisfiability Testing (SAT), Ann Arbor, MI. pp. 174–187. No. 6695 in Lecture Notes in Computer Science, Springer, Berlin (2011)
15. Sakallah, K.A., Simon, L. (eds.): Proceedings of the 14th International Conference on Theory and Application of Satisfiability Testing (SAT), Ann Arbor, MI. Lecture Notes in Computer Science, Springer, Berlin (2011)
16. Sörensson, N., Biere, A.: Minimizing learned clauses. In: Kullmann, O. (ed.) Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT), Swansea, UK. Lecture Notes in Computer Science, vol. 5584, pp. 237–243. Springer (Jun 2009), http://dx.doi.org/10.1007/978-3-642-02777-2_23
17. Strichman, O.: Accelerating bounded model checking of safety properties. Formal Methods in System Design 24(1), 5–24 (2004), http://dx.doi.org/10.1023/B:FORM.0000004785.67232.f8