Mining Backbone Literals In Incremental SAT
A new kind of incremental data

Alexander Ivrii[1] Vadim Ryvchin[2] Ofer Strichman[3]

[1] IBM Research Lab, Haifa, Israel alexi@il.ibm.com
[2] Design Technology Solutions, Intel Co., Haifa, Israel rvadim@tx.technion.ac.il
[3] Information Systems Eng., IE, Technion, Haifa, Israel ofers@ie.technion.ac.il

Abstract. In incremental SAT solving, information gained from previous similar instances has so far been limited to learned clauses that are still relevant, and heuristic information such as activity weights and scores. In most settings in which incremental satisfiability is applied, many of the instances along the sequence of formulas being solved are unsatisfiable. We show that in such cases, with a P-time analysis of the proof, we can compute a set of literals that are logically implied by the next instance. By adding those literals as assumptions, we accelerate the search.

1 Introduction

Incremental SAT solving is used in numerous applications, including Bounded Model Checking [5], SMT solving [10], unsat core extraction [7], high-level (group) UNSAT core extraction [13], and model-checking via IC3 [6]. In all these applications a sequence of closely related SAT formulas is being solved, and by saying that they are solved incrementally we mean that the SAT solver retains information between subsequent calls, in order to expedite the overall process. Originally there was *clause sharing* [16,17,18], which means that learnt clauses at step i, which are still relevant for step $i+1$, are retained. As of MiniSat [8] most competitive solvers support *assumptions*, which enables them to support a far more general incrementality mechanism, which not only retains relevant learned clauses, but also heuristic information such as literal scores and variable activity. Assuming that consecutive instances are similar, this information saves time. Furthermore, only the added clauses and the assumptions have to be communicated to the solver at each iteration, thus saving time on parsing and reloading the formula to memory.

In this article we suggests a new type of data transfer in incremental solving: implied literals. These can also be understood as *backbone literals*, as we will explain later. More specifically, suppose that we already proved the unsatisfiability of one of the formulas in the sequence and about to solve a formula φ; Then we show how to mine literals from the resolution proof of the latest proof of unsatisfiability in the sequence, which are logically implied by φ; These can safely be added to φ as assumption literals (or units), since they do not change its satisfiability.

Our context is a CDCL solver such as Minisat [8], which is augmented with in-memory proof logging, or is at least capable of producing a proof on demand. There is an overhead associated with maintaining the proof in memory, and generic SAT solvers typically refrain from doing it. In some applications such proof logging is necessary, however, like solvers used in interpolation-based model-checking [11,2], the minimal-core extraction tool HaifaMUC [15,14], and the incremental solver described in [14]. For this work we chose to focus on HaifaMUC, as it is open source. HaifaMUC maintains in memory a *partial* proof, and more specifically the part of the proof that is rooted at clauses that will potentially be removed in future instances, which is all that we need for our technique.

In graph terms, a proof is a directed graph in which nodes represent clauses and edges represent the antecedent relation between them, i.e., the parents of a clause are its antecedents. When the empty clause is reachable from some roots of the graph we call the proof a refutation. The edges maintained by solvers, including HaifaMUC, correspond to *hyper resolution* inference. Such an inference has multiple clauses $c_1 \ldots c_n$ as antecedents, and one clause c as a consequent, if and only if *there exists* a *binary* resolution proof of c from $c_1 \ldots c_n$. For simplicity we will refer to the nodes in the resolution proof simply as the clauses that they represent. For a root clause c, let $cone(c)$ denote the maximal reachable subgraph that is rooted at c. For a set of clauses C, we overload *cone* by defining $cone(C) = \bigcup_{c \in C} cone(c)$.

Our technique exploits the following observation, which was originally made by Nadel in his thesis [12]:

Observation 1 *Let π be a refutation. Then every vertex cut in π represents an inconsistent set of clauses.*

Intuitively this observation is correct because every set of clauses that forms a cut (i.e., separates the roots from the empty clause), implies the empty clause, and hence must be unsatisfiable. A consequent of this observation is

Corollary 1 *Let π be a refutation, and let C be a set of root clauses in its core. Let vc be a vertex cut in $cone(C)$, i.e., vc separates C from the empty clause. Then*

$$\pi \setminus cone(C) \Rightarrow \bigvee_{cl \in vc} \neg cl \; . \tag{1}$$

To see why (1) is correct, observe first that $\pi \setminus cone(C)$ and $cone(C)$ represent a partition of π. Falsely assume, then, that there exists an assignment α such that $\alpha \models \pi \setminus cone(C) \wedge \bigwedge_{cl \in vc} cl$. Take any subset of clauses from $\pi \setminus cone(C)$ that together with vc form a cut in π. Since α satisfies both that subset and vc, then α satisfies a full cut in π, which contradicts Observation 1.

The cut vc is of course not unique in $cone(C)$. Let $Cuts(C)$, then, be the set of *all* cuts in $cone(C)$. We can now generalize (1) to

$$\pi \setminus cone(C) \Rightarrow \bigwedge_{vc \in Cuts(C)} \bigvee_{cl \in vc} \neg cl \; . \tag{2}$$

How is (2) relevant to faster incremental SAT solving? Consider the case that we have an unsatisfiable instance ψ accompanied with a refutation π, and in the next iteration we would like to remove a set of root clauses C and add another set of clauses C'. This means that we now need to check

$$\psi' \equiv C' \wedge (\pi \setminus cone(C)) \ . \tag{3}$$

According to (2) we can check instead the equivalent formula

$$\psi' \wedge \bigwedge_{vc \in Cuts(C)} \bigvee_{cl \in vc} \neg cl \ . \tag{4}$$

For simplicity of the discussion we will ignore from hereon the added clauses C', since they do not affect the correctness argument.

There is no a priory reason to believe that checking (4) is easier than checking ψ' itself, however. First, the set *Cuts* can be exponential in size and generally cannot be computed efficiently; second, even if we have this set or part thereof, adding a disjunction of terms which is, while being logically implied by the formula, not emanating from the search itself, is not likely to accelerate the search. But suppose that instead of considering all cuts, we only consider *singleton* cuts, i.e., those that include a single clause. In such a case (4) amounts to adding the negation of those clauses, or in other words adding constants to the formula, which *are* likely to accelerate the search.

In the case of deletion-based MUC extraction, it is rather easy to find singleton cuts, because it is always the case that $|C| = 1$: each iteration corresponds to removing a single candidate clause and checking the satisfiability of the remaining formula. This property is used in a technique called *path strengthening* (PS) [14], which helps finding minimal unsatisfiable cores faster. It is illustrated in Fig. 1.

To see how PS finds those cuts, let c be the removed clause, i.e., $C = c$. PS focuses on the subgraph of $cone(c)$ that leads to the empty clause:

Definition 1 (Rhombus). *Let c be a root clause in the core of a refutation π. Then the* rhombus *of c, denoted $\diamondsuit_\pi(c)$, is the clauses in $cone(c)$ that are on some path from c to the empty clause $\bot$.*

We overload $\diamondsuit_\pi$ to a set of clauses in the natural way: $\diamondsuit_\pi(C) = \bigcup_{c \in C} \diamondsuit_\pi(c)$.

PS finds in linear time a maximal 'chain' of clauses $c_0, c_1, \ldots, c_n$ in $\diamondsuit_\pi(c_0)$ such that $c_0 \equiv c$, for $0 \le i < n$, c_i is a parent of c_{i+1}, and, finally, c_n is the only clause in this chain that has multiple children in $\diamondsuit_\pi(c_0)$. Note that this means that each of them is a *dominator* in $\diamondsuit_\pi(c_0)$ with respect to the empty clause. It then checks $\pi \setminus cone(c)$ under the assumptions $\{\neg l \mid \exists i \in [1..n].\ l \in c_i\}$. These extra assumptions, empirically, accelerate the search [14]. PS can be seen as an extension of *redundancy removal*, which was used in the MUC-finder MUSER2 [3], that only uses $\neg c_0$ as assumptions.

Whereas PS is based on a sequence of clauses $c_0, \ldots, c_n$ as explained above, in this article we show that there are many more literals that can be easily found and assumed in the next instance. In particular, we observe that

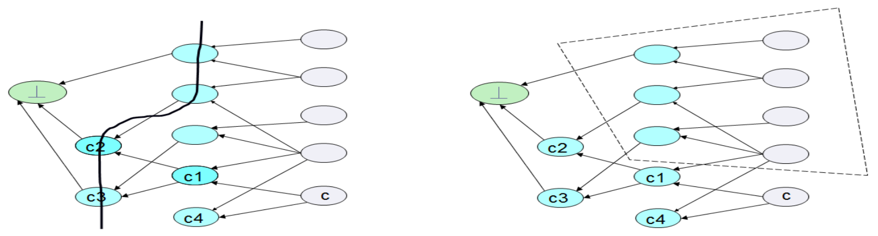


Fig. 1. Demonstrating path strengthening (PS) [14]. Left: given a refutation π, every vertex cut corresponds to a set of clauses that must be contradictory, since together they imply the empty clause. Right: consequently, when removing a clause and its cone ($cone(c) = \{c, c1..c4, \perp\}$), the rest of the formula, which here is marked with a dashed polygon, cannot be satisfied with a model that also satisfies a vertex cut in $\diamondsuit_\pi(c) = \{c, c1..c3, \perp\}$. Each of c and $c1$ constitute a (singleton) cut in $\diamondsuit_\pi(c)$. PS exploits this property, and adds their negation as assumptions when solving $\psi' \equiv \pi \setminus cone(c)$. Identifying such a chain of clauses takes linear time.

Observation 2 *The negation of every literal that appears in every path from* C *to the empty clause is implied by* ψ'.

To see why, consider such a literal l and the set of clauses $S = (l \vee A_1), (l \vee A_2), \ldots$ in $\diamondsuit_\pi(C)$ that it appears in, where $A_1, A_2, \ldots$ are disjunctions of literals. By definition, $S \in Cuts(C)$. Then according to (2), we have

$$\psi' \Rightarrow \bigvee_{cl \in S} \neg cl \tag{5}$$

or equivalently

$$\psi' \Rightarrow (\neg l \wedge \neg A_1) \vee (\neg l \wedge \neg A_2) \vee \ldots \ , \tag{6}$$

from which we can conclude

$$\psi' \Rightarrow \neg l \wedge (\neg A_1 \vee \neg A_2 \vee \ldots) \tag{7}$$

and then

$$\psi' \Rightarrow \neg l \ . \tag{8}$$

Hence, $\neg l$ is implied by ψ', which means that we can add it as an assumption when solving ψ'. We will show in Sect. 2 a P-time algorithm for detecting such literals, and analyze its complexity. We call this extended technique *Mining Backbone Literals*, or MBL for short. The term *backbone literals* of a formula appeared in the past in multiple contexts such as phase transition, maxSAT and optimization problems (see a recent survey in [9]). It is defined as literals that are satisfied in all models of the formula. Generally deciding whether a literal has this property is NP-complete [9], which underlines the benefit we gain from analyzing the proof of ψ. Note that in contrast to all the prior works mentioned in the above survey, we are not interested in finding the complete set of such literals.

Finally, even more literals can be found efficiently if we consider the solver's *state*. Let Con be the set of *constants* implied by ψ', i.e., literals implied at decision level 0 (those, of course, are backbone literals as well). We say that a path in the resolution graph is *satisfied* by Con if at least one of its clauses is satisfied by Con. More literals can be found based on the following observation, which extends Observation 2:

Observation 3 *The negation of every literal that appears in every path* which is not satisfied by *Con from C to the empty clause is implied by ψ'.*

A special case is when all paths from C to the empty clause are satisfied by Con. Then $\psi' \Rightarrow false$ and we can immediately declare ψ' to be unsatisfiable. We will prove the correctness of this technique, which we call MBL-inline, or MBL_i for short, and discuss variations thereof in Sec. 2.2.

Note that PS as described earlier finds only a subset of what MBL (and MBL_i) finds, because any literal in the clauses $c_0, \ldots, c_n$ that PS finds has the property mentioned in Observation 2: it appears in every path from $C \equiv c_0$ to the empty clause. Furthermore, in contrast to PS which is only relevant when $|C| = 1$, MBL addresses the general case, where C is arbitrary, and hence is relevant for general incremental SAT solving.

A preview of our empirical results. As a case study we experimented with this technique in the context of finding minimal unsatisfiable cores (MUC) based on clause-deletion. Our new technique MBL finds on average five times more assumption literals comparing to PS, and does so in a negligible amount of time.

Having PS as our base-line, MBL reduces the run-time of MUC extraction by 6%-7% on average only, depending on the benchmark set. With additional optimizations targeted on MUC extraction that increase the benefit of finding these extra literals, we reach an improvement of 10%. We will describe those optimizations in Sec. 3. This is, admittedly, a modest improvement. Our analysis of the data reveals two possible reasons for this.

First, our data shows that extra assumptions have diminishing value, since at some point many of them are already implied by previous assumptions and constants (by 'constants' we mean literals that are implied via BCP at decision level 0), and are therefore redundant. The ratio of implied assumptions depends on the order in which they are given to the solver, but since this order is rather arbitrary in our case, we measured the redundancy with the default ordering. In our benchmark sets redundancy varies a lot between instances. In the SAT'11 and SAT02_β benchmark sets (each with hundreds of instances) we measured $\approx$40% and $\approx$22% redundancy on average, respectively. We witnessed a positive correlation between the number of assumptions and the redundancy ratio. The scatter graph in Fig. 2 shows this data for the SAT02$_\beta$ benchmark set. One can observe that when the number of assumptions per SAT call is high, the redundancy also goes up, and approaches 1.

Second, there are particular characteristics of deletion-based MUC extraction, which make it vulnerable, in terms of run-time, to assumptions. We will discuss those in detail in Sect. 3. These characteristics are not present in other known popular domains of incremental satisfiability, such as bounded model checking, so it is reasonable to assume that a larger benefit will be observed in such domains.

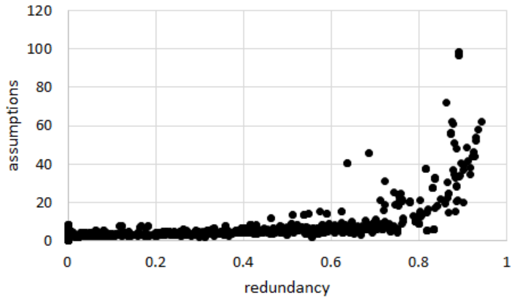


Fig. 2. Assumptions have a diminishing value.

In the next section we present an algorithm for computing MBL and a variation thereof called MBL$_i$. In Sect. 3 we will describe in some detail our case study of MUC extraction: how it is applied and what did we learn from our experiments in this domain.

2 Mining backbone literals

As before let C denote the set of root clauses that are removed when progressing from ψ to ψ'. We assume that ψ is unsatisfiable, and that C is in the core of the refutation π. In our implementation each node n in $\Diamond_\pi(C)$ is a structure with four fields:

- `n.clause`
- `n.LitSet` — a set of literals that are present in all paths from C to `n`. Initialized to $\emptyset$;
- `n.NumOfParents` — the number of parents `n` has in $\Diamond_\pi(C)$. Initialized to 0.
- `n.visited` — whether `n` was visited before in the traversal. Initialized to false.

Alg. 1 presents MBL-Get, which computes the MBL literals — those that are present on each path from the root nodes `C` to the empty clause. In the end of the algorithm, the following relation holds for each node `n` in $\Diamond_\pi(C)$:

$$\texttt{n.LitSet} = \left(\bigcap_{\texttt{p} \in parents(\texttt{n})} \texttt{p.LitSet} \right) \cup \texttt{n.clause}\,. \tag{9}$$

In words, the literals that appear on every path from `C` to `n`, are those that appear on every path to each of its parents, and the literals in the clause of `n` itself.

MBL-Get begins by calling CountParents, which simply updates `n.NumOfParents` with the number of parents `n` has inside the $\Diamond_\pi(C)$. The function children(p), which is called in line 6, returns the set of children `p` has in $\Diamond_\pi(C)$.

MBL-GET maintains a queue of nodes, initialized in line 14 to the clauses in C. A node `n` enters this queue only after its `n.LitSet` was fully computed (in the case of a root node, its literal set is the clause itself, as can be seen in line 11). From (9) it is clear that we can compute this set only after such sets were computed for all of `n`'s parents. This is why we need `n.NumOfParents`: we use it to guarantee that a node enters the queue only after all its parents were processed — see lines 24 and 26. We refer to the currently processed node as the parent, and denote it by `p`, as can be seen in line 16. We iterate through `p`'s children, intersect their current literal sets with `p.LitSet` in line 23, and if `p` is the last parent of a child node `n`, then we add in line 25 `n.clause` (see right element of (9)) and enqueue `n`. The last node to be processed — see line 17 — is the empty clause. Its literal-set is the result of this procedure. The negation of each of these literals can be added to ψ', the next formula to be solved, without changing its satisfiability.

Complexity. Let N, E be the number of nodes and edges in $\diamondsuit_\pi(C)$, respectively, and let K be the size of the largest clause in $\diamondsuit_\pi(C)$. The maximal size of the literal set is the number of different literals in $\diamondsuit_\pi(C)$, which we will denote by L. Each of the clauses have to be sorted for the union and intersection operations (lines 23 and 25), which takes $O(N(K \log K))$. Intersection takes not more than $O(L)$, and there are $O(E)$ intersections. The overall complexity is thus $E \cdot L + N \cdot K \log K$.

Example 1. Fig. 3 shows a possible $\diamondsuit_\pi(C)$ for C having a single clause $c = $ (1 2 -3) on the left, and the corresponding `n.LitSet` computed for each node `n` by MBL-GET on the right. Note that $\diamondsuit_\pi(C)$ is part of a hyper-resolution proof, hence the internal nodes may have additional incoming edges from outside of $\diamondsuit_\pi(C)$, which are not shown in the figure. The algorithm returns `n.LitSet` for `n` being the empty clause in line 17, namely the literal set (1 2 -3 5) in this case. These are the literals that appear in all paths from C to the empty clause. In case of $|C| = 1$, as it is in this example, the output always includes the literals of the root clause itself. Here the prefix of clauses under c that has a single parent in the cone of c is just c itself. Hence with PS we would return in this case (1 2 -3) only, while missing the literal 5.

2.1 Optimizations and implementation details

Some additional details about how we implemented MBL-GET in practice follow:

- **Optimization I.** In case $|C| = 1$, we collect the literals in the initial chain of clauses (the same chain that was used in PS and was described in Sect. 1) — these are known to be part of the end result. Then, rather than propagating them down the resolution graph, we just save them and add them to the set of literals that is eventually returned by MBL-GET. In the example resolution graph of Fig. 3, this optimization amounts to *not* propagating (1

Algorithm 1 CountParents updates `n.NumOfParents` for each node `n` with the number of parents in $\diamondsuit_\pi(C)$. MBL-Get computes the set of literals that are on every path from C to the empty clause. Those are the literals that are present in each clause along some vertex cut of $\diamondsuit_\pi(C)$.

```
1: function CountParents(nodeSet C)
2:     NodeQueue Q;                                          ▷ A queue of nodes
3:     Q.enqueue(C);
4:     while Q is not empty do
5:         p = Q.dequeue();
6:         for each node n in children(p) do
7:             n.NumOfParents++;
8:             if n.NumOfParents = 1 then Q.enqueue(n); ▷ Ensures n enters Q once

9: function MBL-Get(nodeSet C)
10:    CountParents($C$);
11:    for each c in C do
12:        c.LitSet = c.Clause;
13:    NodeQueue Q;                                          ▷ A queue of nodes
14:    Q.enqueue(C);
15:    while Q is not empty do
16:        p = Q.dequeue();
17:        if p.clause.isEmpty() then return p.LitSet;
18:        for each node n in children(p) do
19:            n.NumOfParents--;
20:            if !n.visited then
21:                n.LitSet = p.LitSet;
22:                n.visited = true;
23:            else n.LitSet.intersect(p.LitSet);
24:            if n.NumOfParents = 0 then                    ▷ All parents handled
25:                n.LitSet.union(n.clause);
26:                Q.enqueue(n);
```

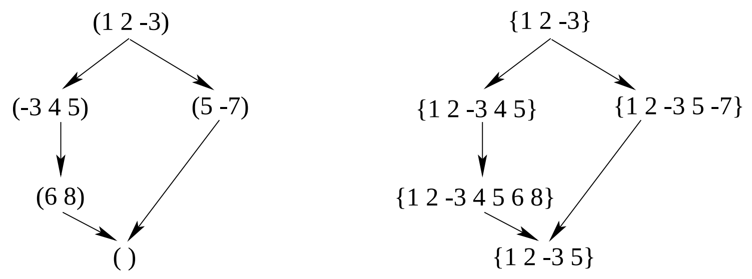


Fig. 3. Left: a partial (hyper) resolution graph corresponding to $\diamondsuit_\pi$(1 2 -3). Right: At each node `n`, showing `n.LitSet` at the end of MBL-Get.

2 -3) down the graph, rather adding these literals only at the very end, hence saving some of the cost of intersection in line 23. Since the overall algorithm is linear in the size of the graph, the importance of this optimization is admittedly marginal.

– **Optimization II.** Cutoff values: we collected statistics that show that when the width of $\diamondsuit_\pi(C)$ is large, only few MBL literals, if at all, are detected; furthermore, the wider $\diamondsuit_\pi(C)$ is, the longer it takes to compute the literals[1]. We therefore terminate early the computation if either the number of children of C in $\diamondsuit_\pi(C)$ crosses a certain threshold Th_1, or the width of $\diamondsuit_\pi(C)$ crosses another threshold Th_2 (i.e., the size of the queue in line 4). Our implementation currently sets these values by default to $Th_1 = 400$ and $Th_2 = 500$, based on (limited) experiments. If one of the thresholds is crossed we revert to PS (we can do it in our case because, recall, for MUC extraction it is always the case that $|C| = 1$. In the general case the fallback solution can be to not report assumptions to the next instance).
– **Handling deleted clauses.** The description of MBL-Get so far ignored the fact that not all clauses are available along the resolution graph due to clause deletion. Our solver maintains the graph itself (the arcs), but the clauses themselves are possibly deleted from memory. Our solution to this problem is to skip line 25 when encountering such a clause. This means that the end result may be weakened, since some of the literals in that clause could have been included in that set.

2.2 Constants can increase the number of assumptions

Constants — variables that are forced to a particular Boolean value at decision level 0 — are prevalent in the solving process. In our experiments with industrial benchmarks we witnessed hundreds of constants already after the initial propagation at level 0, and then many more learned during the search. Let σ be the partial assignment corresponding to the constants right after the propagation at decision level 0 (note that we are not restricting ourselves to the initial propagation; the solver may backtrack to level 0 many times during the search). By definition of constants, any assignment that satisfies the formula, if such an assignment exists, is an extension of σ. This fact can be used to increase the set of MBL literals as hinted in the introduction (see Observation 3).

In the traversal of $\diamondsuit_\pi(C)$ by MBL-Get, we ignore the set of clauses $\{c' \mid \sigma \models c'\}$, i.e., the set of clauses that are satisfied by the constants. This effectively 'cuts' paths to the empty clause, and hence potentially increases the set of MBL literals, because we intersect the literal sets in line 23 of less clauses. To apply this optimization, right after line 18 we check if `n.clause` is satisfied by σ, and if yes then we `continue`, i.e., jump to the next iteration of the `for` loop.

[1] It was shown in [4] that wide proofs are necessarily long, although there the reference was to resolution and not hyper resolution. If we consider the rank of each clause in the hyper-resolution proof to be bounded, then this result applies here as well.

Theorem 1. *Let l denote a literal in* `n.LitSet`*, where* `n` *is the empty clause, in the revised* MBL-Get *described above. Then $\psi' \Rightarrow \neg l$.*

Proof. Observe that the set of clauses S in $\diamondsuit_\pi(C)$ that contain l, and the set S' of nodes that were ignored by the revised procedure as described above, form a cut in $\diamondsuit_\pi(C)$. Falsely assume the existence of an assignment α such that $\alpha \models \psi' \wedge l$. Then $\alpha \models \bigwedge_{cl \in S \cup S'} cl$, because it satisfies l and it coincides with σ on the constants. Take any subset of nodes in ψ' that together with $S \cup S'$ form a cut in π. Since that subset is also satisfied by α, then we have a cut in π that is satisfied by α, which contradicts Observation 1. $\square$

As an illustration, consider again Example 1, and suppose that -7 is a constant implied by the formula. Then the clause (5 -7) is satisfied and hence ignored, and then the literal set reaching the empty clause is larger: (1 2 -3 4 5 6 8).

Note that the MBL literals may imply other constants, hence this idea can be applied iteratively, until convergence. Empirically it happens quite often that with this process all paths to the empty clause are cut by constants, a case in which we immediately return UNSAT, since every assignment that satisfies ψ' also satisfies a cut in $\diamondsuit_\pi(C)$.

We implemented several variations of this basic idea, and let the user control them via flags. One flag controls whether it is indeed applied iteratively until convergence, or only once. Another flag determines whether to apply this in the beginning of the search after propagation at level 0, or, at the other extreme, every time the solver backtracks to level 0 with a new learned constant. In the experiments presented in Sec. 3.2, we refer to this technique with the first flag turned off and the second turned on, as *MBL-inline*, or MBL$_i$ for short.

3 A case study: using MBL in the context of MUC extraction

We implemented the ideas described in the previous section in HaifaMUC [15,14], a minimal unsat core extractor, which is based on resolution and hence fits our needs. In the following subsection we will describe the core algorithm of Haifa-MUC, which will help us later, in sect. 3.2, to explain the experimental results.

3.1 How HaifaMUC extracts Minimal Unsatisfiable Cores.

HaifaMUC is a deletion-based minimal unsatisfiable core extractor that maintains (parts of) the proof in memory. Algorithm 2 describes in pseudo-code a simplified version of its main loop (see [14] for the full version), still based on PS. For MBL simply replace the called function in line 19 with MBL-Get. The implementation of MBL$_i$ is more complicated as it is intertwined in the search engine of the solver, so we will not present it here in pseudocode. The code is rather self-explanatory so let us only emphasize the role of the Boolean variable *LastProofOK*. This variable is set to false if the last unsatisfiability proof relied on assumptions. In such a case we cannot perform PS, because we do not have a proof of the empty clause (we only have a proof of the negation of the assumptions). An actual proof of the empty clause is necessary because without it we cannot compute a path with the properties explained in Sect. 1. Examining the logs one can frequently see an unsat proof based on assumptions, and then a long sequence of SAT results, none of which can enjoy the benefit of PS. This leads us to:

Observation 4 *Reaching an unsat answer based on assumptions, while generally being faster than without them, has two drawbacks:*

1. *PS (and similarly MBL) cannot be used until another iteration results in a full proof, and*
2. *Only the candidate clause c is removed rather than everything outside the cone (compare line 7 to lines 10–12).*

We see, then, that there is a *negative feedback* process here: more assumptions increase the chances of the next proof using them, and this prevents us from mining more assumptions. This observation helps understanding the results that will be presented next.

3.2 Experiments

Results with the 2011 MUC competition benchmarks We took the 200 benchmarks of the 2011 MUC-extraction track[2] (the mus/ subdirectory in the archive), the only competition ever held in this track, and removed four of them that could not be solved by any of the techniques in 15 minutes. According to the remaining 196 benchmarks — see Fig. 4 (left) — there is 6% improvement in the run-time of MBL comparing to PS, our base line, whereas MBL_i has a 0.5% negative effect. When considering the run-time per benchmark family — see Fig. 4 (right) — it is evident that the degradation in results of MBL_i happens only in one family (the 'abstraction-refinement-intel' set)[3], whereas in the other families it either improves or has marginal effect on the results. Returning to the table in the left of the figure: 'literals' is the number of added assumption literals, 'iterations' is the number of iterations until a MUC is found, and 'lit/iter' is the ratio between them; 'unsat by assump.' is the number of iterations in which the solver returned unsat based on the assumptions; 'overhead' is the *total* run time spent on finding the literals (i.e., for all iterations); 'longest call' is the *longest* SAT call (in sec.) in every CNF instance (not including the initial run) of Alg. 2, and finally '% implied assump.' is the percentage of literals that their value was already implied by previous literals and constants in the formula by the time we tried to apply them.

The number of assumption literals that we add per SAT invocation is 4.1X larger with MBL comparing to with PS. Interestingly it is only 3.5X for MBL_i, despite the fact that MBL_i searches more aggressively for such literals. The

[2] Officially that track was called MUS, for minimal unsatisfiable set.
[3] In fact it happens because of one benchmark in this family – an extreme outlier.

Algorithm 2 The basic main algorithm of HaifaMUC with PS. In the first iteration the condition in line 5 is true and in line 6 it is false.

```
Input:  Unsat formula $\psi$.
Output: A MUC of $\psi$.

1:  $assumptions = \emptyset$;
2:  Mark all $\psi$'s clauses as 'unknown';
3:  while true do
4:      $\langle res, \pi \rangle := SAT(\psi, assumptions)$;
5:      if res = UNSAT then
6:          if $assumptions$ used in proof then
7:              Mark $c$ as 'not in MUC';
8:              $LastProofOK$ = false;
9:          else
10:             for each root outside of $core(\pi)$ do
11:                 Mark root as 'not in MUC';
12:                 Remove $cone(root)$ from $\psi$;
13:             $LastProofOK$ = true;
14:     else                                                  ▷ SAT
15:         Mark $c$ as 'MUC';                                ▷ $c$ is now in core
16:         Add $TmpRemoved$ back to $\psi$;
17:     if no clause is marked as 'unknown' then return clauses marked as 'MUC';
18:     Select $c$ from roots marked as 'unknown';
19:     if $LastProofOK$ then $assumptions$ = PathStrengthening(c);
20:     else $assumptions = \neg c$;
21:     $TmpRemoved = cone(c)$;
22:     $\psi = \psi \setminus TmpRemoved$;
```

Measure	PS	MBL	MBL$_i$
success	196	196	196
time	21.5	**20.3**	21.7
literals	2770.8	11377.2	9886.1
iterations	1180.6	1188.5	1203.7
lit / iter	2.3	9.6	8.2
unsat by assump.	226.5	232.4	236.2
decisions (M)	30.7	30.6	47.3
overhead	0.0	0.3	0.3
longest call	0.5	0.4	0.4
% implied assump.	0.4	0.5	0.5

Bench. Family	PS	MBL	MBL$_i$
fdmus-v100	16.7	16.8	16.7
abs-ref-intel	26.3	**22.3**	36.7
atpg	0.0	0.0	0.0
bmc-aerielogic	4.6	4.4	4.4
bmc-default	6.3	6.3	6.3
design-debugging	1.1	1.1	**1.0**
equivalence-checking	59.9	55.3	**43.3**
fpga-routing	58.2	**50.3**	50.8
hardware-verification	48.4	47.7	**44.9**
product-configuration	0.0	0.0	0.0
software-verification	63.1	**57.6**	64.9
Total	21.5	20.3	21.7

Fig. 4. Results with the 2011 MUC track benchmarks. Left: various measures. Right: Run-time (sec.) by family;

reason for this phenomenon is related to the first item of Observation 4: more assumptions increase the probability that the proof relies on them, which in turn shuts off the search for such assumptions in future iterations, until there is an unsat proof without assumptions. Indeed observe that 'unsat by assump.' is higher for MBL_i. Also observe that the number of iterations is higher with MBL_i, which relates to the second part of Observation 4.

Analysis of the run-time of MBL: according to the 'overhead' row, the total amount of time spent on computing the MBL literals is less than half a second per CNF instance. Dividing it by the number of iterations shows that it takes on average around $3 * 10^{-4}$ seconds per SAT call. Recall that we bound the search for MBL literals to prevent spending too much time in case the proof is large (see optimization II in Sect. 2), hence the theoretical exponential upper-bound on the size of the proof is irrelevant: the overhead is always small. Indeed examining the data further reveals that it takes less than 10^{-3} sec. in 95% of the cases and less than $2*10^{-2}$ seconds in all cases. For comparison, we analysed the run time of the solver itself. The average time of a SAT call (not including the initial call) in the case of MBL is 0.012 sec., which is about 40X longer than the time it takes to compute the MBL literals. In these instances the SAT run-time is particularly short[4]. Hence the relative overhead is expected to be smaller on harder instances.

Analysis of implied assumptions. We already mentioned in the introduction that more assumptions typically implies more redundancy, i.e., a larger portion of them are implied by other assumptions. To quantify the connection between these two figures, we computed their *Spearman correlation* ρ. This is a frequently-used measurement for checking *monotonicity* between two arrays, i.e., for two arrays $A1, A2$ of equal size, $\rho(A1, A2) = 1$ if and only if for all i, j, when $A1[i] > A1[j]$ then $A2[i] > A2[j]$; $\rho(A1, A2) = -1$ if the opposite relation holds, and $\rho(A1, A2) = 0$ if there is no relation between the two arrays. With the above benchmark set we computed $\rho = 0.57$, which shows a strong connection between these two figures.

Results with the SAT 2002$_\beta$ competition benchmarks The SAT 2002$_\beta$ competition benchmark set contains over 500 CNF application instances. After removing those that are SAT and benchmarks that cannot be solved with any of our parameters within 15 minutes, we were left with 216 instances. The table in Fig. 5 presents these results. The 5 columns on the right repeat the results for those 52 benchmarks in the set that at least in one of the parameters sets, they impose a SAT call that takes over a second. The table shows a gain of 5% and 6% to MBL and MBL_i in the first set, respectively, and a gain of 3% and 7% to MBL and MBL_i in the second set, respectively. The increased benefit matches our intuition that extra assumptions can help more in hard SAT calls.

[4] This is expected for this benchmark set, because only easy CNF instances were selected for inclusion in this set to begin with.

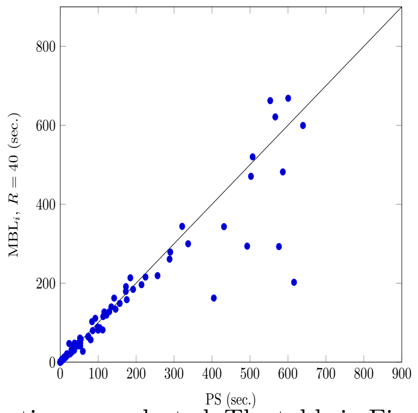
	all			longest > 1		
Measure	PS	MBL	MBL$_i$	PS	MBL	MBL$_i$
success	215	215	215	52	52	52
time	62.3	59.3	**58.9**	264.5	258.1	**245.9**
literals	3321.9	5778.1	5925.3	9484.5	32960.1	31628.3
iterations	1032.4	1037.1	1044.4	3817.0	3909.9	3867.7
lit/iter	3.2	5.6	5.7	2.5	8.4	8.2
unsat by assump.	284	285.9	287	1296.1	1386.0	1331.42
decisions (M)	10.0	10.3	10.6	68.7	69.8	67.7
overhead	0.0	0.1	0.2	0.0	0.8	1.2
longest call	2.0	1.9	1.8	9.8	9.3	8.9
% implied lit.	0.1	0.1	0.1	0.0	0.2	0.2

Fig. 5. Results on the sat02$_\beta$ benchmarks, rounded.

3.3 Mitigating the side-effect of assumptions on MUC search

As described in Observation 4, in MUC extraction extra assumptions can have indirect negative impact on the run-time. We now describe several heuristics that we experimented with, to mitigate this undesired side-effect.

- *Delayed activation.* Activate assumptions (in any one of the three methods) and the search for additional assumptions (with MBL$_i$), only after running for a while without assumptions. If the solver is able to find a solution within this delay, we gain by having a proper proof.
- *Continue execution after detection of UNSAT by assumptions.* When reaching UNSAT based on the assumptions, ignore the fact that we know the result is unsat, and continue the search for a bounded amount of time (defined by a 'budget' R) with the hope that the SAT solver will detect UNSAT without using the assumptions. The value of the assumptions with such a strategy is limited to the SAT case, and as a fallback solution when it takes too much time to find a proof without them.



In our implementation both the delay and the budget R are defined in terms of the number of restarts. We experimented with two variation of the second optimization above: either reset R to 0 for every SAT call, or not. The latter makes sense in the context of MUC, because the earlier iterations, when they end with UNSAT, are able to remove large parts of the proof via an analysis of the core, hence we want to spend time for finding a proof without assumptions. Indeed in all the empirically winning configurations the latter option was selected. The table in Fig. 6

shows results of the best configurations that we found for this benchmark set. The best configuration shows 10% improvement over the base-line, and one more solved instance. The scatter graph above on the right shows that most of the benefit is in the hard instances.

The default of HaifaMUC is now set to the above best configuration, and its source code is available in [1]. Spreadsheets with detailed results can be found at the same location.

Measure	PS, R=120	MBL, R=160	MBL$_i$, $R = 80$, Delay=5	MBL$_i$, R=40
success	215	**216**	215	215
time	56.9	**56.5**	57.0	**56.1**
literals	3057.3	11820.9	3162.6	7940.6
iterations	969.9	998.7	979.4	1003.7
lit/iter	3.2	11.8	3.2	7.9
unsat by assump.	211.1	212.6	223.5	245.0
decisions (M)	10.1	10.5	10.0	10.2
overhead	0.00	0.38	0.03	0.21
longest call	1.83	2.15	1.73	1.67
% implied assump.	0.14	0.27	0.15	0.24

Fig. 6. Results on the SAT02$_\beta$ benchmarks, rounded, with various configurations of the heuristics to mitigate the negative side-effect of extra assumptions. A timeout is counted as 900 sec.

4 Conclusions and future work

We presented a technique called Mining Backbone Literals (MBL) and variations thereof, that potentially accelerates incremental satisfiability. It is based on analysing resolution refutations for the purpose of finding literals that are logically implied by the consecutive SAT call. This is a new type of data transfer in incremental satisfiability.

Our case study in the domain of MUC extraction showed only modest improvement ($\approx$10%) in run-time, but as we explained this is mostly related to specific characteristics of the MUC-extraction algorithm, which are not present in other domains in which incremental SAT is used. We hope that future research will a) reveal how to overcome these characteristics in deletion-based MUC extraction, and b) investigate the impact of MBL in other domains. Recently a new format ('IPASIR') for incremental solving was suggested as part of the preparations for the SAT'15 race, and hopefully standard benchmarks will follow soon; then it will be very interesting to try our methods in the domain of general incremental satisfiability.

References

1. HaifaMUC is available from `http://ie.technion.ac.il/~ofers/haifasolvers/`.
2. Omer Bar-Ilan, Oded Fuhrmann, Shlomo Hoory, Ohad Shacham, and Ofer Strichman. Reducing the size of resolution proofs in linear time. *STTT*, 13(3):263–272, 2011.
3. Anton Belov and João Marques-Silva. MUSer2: An efficient MUS extractor. *J. on Satisfiability, Boolean Modeling and Computation (JSAT)*, 8(1/2):123–128, 2012.
4. Eli Ben-Sasson and Avi Wigderson. Short proofs are narrow - resolution made simple. *J. ACM*, 48(2):149–169, 2001.
5. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, LNCS, pages 193–207. Springer-Verlag, 1999.
6. Aaron R. Bradley. Sat-based model checking without unrolling. In Ranjit Jhala and David A. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2011.
7. Nachum Dershowitz, Ziyad Hanna, and Alexander Nadel. A scalable algorithm for minimal unsatisfiable core extraction. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 36–41, 2006.
8. Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing (SAT)*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003.
9. Mikolás Janota, Inês Lynce, and Joao Marques-Silva. Algorithms for computing backbones of propositional formulae. *AI Commun.*, 28(2):161–177, 2015.
10. Daniel Kroening and Ofer Strichman. *Decision procedures – an algorithmic point of view.* Theoretical computer science. Springer-Verlag, Feb. 2008. (to be published).
11. K.L. McMillan. Interpolation and SAT-based model checking. In Jr. Warren A. Hunt and Fabio Somenzi, editors, *Computer Aided Verification (CAV)*, LNCS, Jul 2003.
12. Alexander Nadel. *Understanding and Improving a Modern SAT Solver.* PhD thesis, Tel Aviv University, Tel Aviv, Israel, August 2009.
13. Alexander Nadel. Boosting minimal unsatisfiable core extraction. In Roderick Bloem and Natasha Sharygina, editors, *FMCAD*, 2010.
14. Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. Efficient MUS extraction with resolution. In *FMCAD*, pages 197–200. IEEE, 2013.
15. Vadim Ryvchin and Ofer Strichman. Faster extraction of high-level minimal unsatisfiable cores. In *Theory and Applications of Satisfiability Testing (SAT)*, number 6695 in LNCS, pages 174–187, 2011.
16. Ofer Shtrichman. Sharing information between instances of a propositional satisfiability (SAT) problem, Dec 2000. US provisional patent (60/257,384). Later became patent US2002/0123867 A1.
17. Ofer Shtrichman. Prunning techniques for the SAT-based bounded model checking problem. In *proc. of the 11th Conference on Correct Hardware Design and Verification Methods (CHARME'01)*, Edinburgh, September 2001.
18. Jesse Whittemore, Joonyoung Kim, , and Karem Sakallah. Satire: a new incremental satisfiability engine. In *In IEEE/ACM Design Automation Conference (DAC)*, 2001.