Bit-parallel Learning from Generic Assignments

Dietger van Antwerpen, Menno den Hollander, and Bart de Keijzer

Delft University of Technology

Abstract. We present a technique to learn new clauses from a set of assignments, which is inspired by the learning rules of Stålmarck's proof procedure. Central in the proposed learning method is the concept of generic assignments – a set of 2^n assignments that covers all possible truth values on n Boolean variables. Given a 2^n bit computer, one can store generic assignments of that size efficiently. Further, such assignments can be extended with unit propagation in parallel. We show that various learning rules can be applied in parallel as well.
BitFall is a SAT-solver based on these ideas. It is complete and uses breadth-first search – similar as the HeerHugo solver. We offer some first experimental results of BitFall on hard random 3-SAT formulas. Our algorithm seems to work surprisingly well on unsatisfiable instances.

1 Introduction

Complete satisfiability (SAT) solvers have become very powerful in recent years. All strong complete solvers are based on depth-first (DPLL [1]) search. Important contributions to this success are learning techniques and efficient data-structeres. On the other hand, breadth-first SAT solvers are rare and not competitive. A well-known architecture for these solvers is the *Stålmarck's proof procedure* [2]. Given a formula $\mathcal{F}$, it computes for a variable x two reduced formulae: one by assigning x to true ($\mathcal{F}^+$) and one by assigning x to false ($\mathcal{F}^-$). All clauses that are not in $\mathcal{F}$, but which are present in both $\mathcal{F}^+$ and $\mathcal{F}^-$ can be learnt and added to $\mathcal{F}$. HeerHugo [3] is a breadth-first SAT solver, which applies this procedure systematically.

We propose to improve Stålmarck's proof procedure using an efficient data-structure called *generic multi-bit assignments*: a set of 2^b assignments that cover all possible truth-values to b Boolean variables. Using a 2^b-bit processor, one can efficiently deal with these assignments by performing logical operations in parallel. The concept of multi-bit assignments was introduced in [4,5], where the authors enhance the UnitWalk algorithm [6]) with multi-bit assignments and multi-bit unit-propagation. The resulting solver UnitMarch showed the usefulness of these concepts.

In this paper, we continue to investigate the possibilities of multi-bit truth-assignments by devising rules for learning extra clauses that can be found by using generic assignments. The presented learning rules are inspired by HeerHugo, although slightly different. Instead of merging (learning) all clauses in the reduced formulae, we only merge the unit clauses. This restriction should be regarded as a first step towards efficient learning using generic assignments. Yet, regardless this restriction, we can learn more than merely unit clauses. For instance, if all reduced formula x_i equals x_j, we learn binary equivalence $x_i \leftrightarrow x_j$. Further, if in some reduced formulae unit clause $\neg x_i$ is detected, while in all other reduced formula unit clause x_j occurs, we learn $\neg x_i \vee x_j$. Finally, conflict clauses can be learnt from those reduced formulae with the empty clause (a conflict).

This paper is structured as follows. In Section 2, we will explain the ideas and necessary theory that our learning technique is based on: these are multi-bit truth-assignments, Stålmarck's branch/merge-rule, multi-bit unit propagation and generic assignments. Section 3 explains the various learning rules that are used in the solver. In Section 4, we describe our SAT-solving procedure in global, and we explain some last steps in our algorithm that have not been explained in the previous sections. In Section 5, we give experimental results on the performance of our SAT-solver. Finally, Section 6 concludes this text.

2 Generic Assignments

The concept of *generic multi-bit truth-assignments* (in short: *generic assignments* or GA's) was proposed in [7]. This section introduces this concept, followed by how it can be extended. Generic assignments are at the heart of the learning techniques that will be discussed in Section 3.

2.1 The concept

Definition 1 (truth-assignment). *A truth-assignment for a propositional Boolean formula $\mathcal{F}(x_1, \ldots, x_n)$ is a list $\varphi=(s_1, \ldots, s_n)$ with $s_i \in \{\mathtt{0},\mathtt{1},\mathtt{*},\mathtt{!}\}$. In other words, s_i represents the truth-value assigned to variable x_i.*

Intuitively, assigning a variable to `0` stands for making the variable *false*; assigning a variable to `1` stands for making the variable *true*; and assigning a variable to `*` stands for leaving the variable unassigned. The truth-value `!` is non-standard: it stands for what we call the *bit-conflict state*. We will explain the precise meaning of this truth-value later on. For now, it is sufficient to know that if the bit-conflict state is assigned to one or more of the variables in a truth-assignment, then the formula is falsified.

Definition 2 (multi-bit truth-assignment). *A multi-bit truth-assignment is a list of truth-assignments. Formally, given a propositional Boolean formula $\mathcal{F}(x_1, \ldots, x_n)$, ϕ is a multi-bit assignment for $\mathcal{F}$ iff $\phi = (\varphi_1, \ldots, \varphi_b)$ with φ_i as a truth-assignment for $\mathcal{F}$. For ease of discussion, if $\phi = (\varphi_1, \ldots, \varphi_b)$ is a multi-bit truth-assignment, we define $s_{i,j}$ to be the assignment of x_j in φ_i.*

Definition 3 (generic multi-bit truth-assignment). *A generic multi-bit truth-assignment is a multi-bit truth-assignment of* 2^b *assignments to* b *Boolean variables such that all possible truth-assignments to those variables are covered.*

From this point onward we will call a propositional Boolean formula simply a *formula*. Also, we will use the word *assignment* instead of "truth-assignment".

Given a multi-bit assignment $\phi = (\varphi_1, \ldots, \varphi_b)$, we can represent any $s_{i,j}$ using two bits on a physical computer (because there are 4 possible truth values). We encode `*` as the bit-pair $(0, 0)$, we encode `1` as $(1, 0)$, we encode `0` as $(0, 1)$, and we encode `!` as $(1, 1)$.[1]

The name "multi-bit" comes from the following fact: we can exploit bit-parallelism if we store for any j all of the values $(s_{1,j}, \ldots, s_{b,j})$ into two bit words. The first of the bitwords contains all first bits of the two-bit encodings of $(s_{1,j}, \ldots, s_{b,j})$, and the second word contains all second bits. On a machine with a b'-bit architecture it makes most sense to choose b to be as large as b'. As an example, suppose that we have a variable x_1 occurring in some formula, and suppose that we have a multi-bit assignment $\phi = (\varphi_1, \ldots, \varphi_8)$ in which $(s_{1,1}, \ldots, s_{8,1})$ is equal to $(0, !, 1, *, *, 0, 1, *)$. We will from now on use a more convenient notation: we simply say $\phi(x_1) =$ `0!1**01*`.

$\phi(x_1)$	00000000000000000000000000000000111111111111111111111111111111
$\phi(x_2)$	00000000000000001111111111111111000000000000000011111111111111
$\phi(x_3)$	00000000111111110000000011111111000000001111111100000000111111
$\phi(x_4)$	00001111000011110000111100001111000011110000111100001111000011
$\phi(x_5)$	00110011001100110011001100110011001100110011001100110011001100
$\phi(x_6)$	01010101010101010101010101010101010101010101010101010101010101

Fig. 1. Generic assignment $\phi = (\varphi_1, \ldots, \varphi_{64})$ for 6 variables on a 64-bit CPU.

In a computer, we represent this assignment of x_1 as the pair of bit words $(\phi^+_-[x_1], \phi^+_-[\neg x_1])$, with

$$\phi^+_-[x_1] = 01100010 \qquad \phi^+_-[\neg x_1] = 11000100$$

Why is this representation convenient? In this way it is possible to check several properties of a variable's assignment relatively easy by using bitwise Boolean operations (which computers can do efficiently in general). We can for example find out very efficiently in which simple assignments of the multi-bit assignment, x_1 is set to `*`. We do this by performing a bitwise OR-operation on both bit words of x_1, and a bitwise NOT-operation on the result. We then have a bit mask as a result with `1`s on the positions corresponding to simple assignments where x_1 is assigned the truth-value `*`. In a similar way we can check in which simple assignments x_1 is set to true or set to false.

[1] Note that we use the symbols `0` and `1` both to represent bits and truth-values. We believe that throughout this text it is always clear which of the two is meant, and no confusion will arise.

The encoding is not arbitrary. One should interpret the first of the two bit words as an indicator for whether the variable is set to true, and one should interpret the second of the two bit words as an indicator whether the variable is set to false. According to this interpretation, the configuration $(\mathtt{0},\mathtt{0})$ means that a variable is neither set to true nor false; the configuration $(\mathtt{1},\mathtt{0})$ means that a variable is set to true, and is not set to false; the configuration $(\mathtt{0},\mathtt{1})$ means that the variable is not set to true, but is set to false; finally, the configuration $(\mathtt{1},\mathtt{1})$ would mean that the variable is set to both false and true. This is not a valid truth-value and thus indicates that something is wrong. We make use of this *bit-conflict* state in the multi-bit unit propagation algorithm that we use in our solver. In Section 2.2 we will come back to this.

2.2 Multi-bit Unit Propagation

Our SAT-solver uses *multi-bit unit propagation* to extend truth-assignments. In order to explain unit propagation, we will first define what a *unit-clause* is.

Definition 4 (unit-clause). *Given an assignment $\varphi = (s_1, \ldots, s_n)$ for formula $\mathcal{F}$, clause $C \in \mathcal{F}$ is a unit-clause for φ iff C has one unassigned literal* $(\mathtt{*})$*, while all other literals are assigned to false* $(\mathtt{0})$*.*

Unit propagation works by detecting unit-clauses and setting the unassigned variable of that unit-clause to the truth-value $\in \{\mathtt{0},\mathtt{1}\}$ that makes the clause true. During this process of satisfying unit clauses, there are other clauses that may become unit. This process continues until (1) all unit-clauses are satisfied or (2) two complementary unit-clauses x_i and $\neg x_i$ are detected. In the latter case, x_i must be assigned to both `0` and `1`, so the bit-conflict state `!` gets assigned to x_i.

The extension to *multi-bit* unit propagation (i.e. unit propagation on multi-bit assignments) is easily made: we simply execute unit propagation on all assignments that constitute the multi-bit assignment. This can be done efficiently by making clever use of bitwise operations. In fact, just like conventional unit propagation, multi-bit unit propagation can be done in $O(n)$ time. In UnitMarch, this linear time algorithm for multi-bit unit propagation has already been implemented (see [4,5]). Our solver, called BitFall, makes use of it.

Example 1. We will illustrate extending a generic assignment using unit propagation by using the formula below:

$$\begin{aligned}\mathcal{F}_{\text{example}} = {} & (\neg x_1 \vee \neg x_5) \wedge (x_1 \vee \neg x_7) \wedge (\neg x_2 \vee x_4) \wedge (x_2 \vee x_6 \vee \neg x_7) \wedge \\ & (x_2 \vee x_7) \wedge (\neg x_3 \vee \neg x_4) \wedge (x_3 \vee x_4 \vee x_5) \wedge (\neg x_3 \vee \neg x_5 \vee x_7) \wedge \\ & (x_3 \vee x_5 \vee x_7) \wedge (\neg x_3 \vee \neg x_6 \vee \neg x_7) \wedge (x_3 \vee x_6 \vee \neg x_7)\end{aligned}$$

Say, we assign a generic assignment to the variables x_5, x_6 and x_7:

$$\begin{aligned}\phi(x_1) &= \texttt{********} & \phi(x_5) &= \texttt{01010101} \\ \phi(x_2) &= \texttt{********} & \phi(x_6) &= \texttt{00110011} \\ \phi(x_3) &= \texttt{********} & \phi(x_7) &= \texttt{00001111} \\ \phi(x_4) &= \texttt{********}\end{aligned}$$

Observe that there are several unit-clauses. Consider the first two clauses:

$$\begin{aligned}(\neg x_1 = \texttt{********} \vee \neg x_5 = \texttt{10101010}) \quad &\Rightarrow \quad \phi(x_1) = \texttt{*0*0*0*0} \\ (x_1 = \texttt{*0*0*0*0} \vee \neg x_7 = \texttt{11110000}) \quad &\Rightarrow \quad \phi(x_1) = \texttt{*0*01!1!}\end{aligned}$$

After unit propagation the generic assignment is extended to:

$$\begin{aligned}\phi(x_1) &= \texttt{*0*01!1!} & \phi(x_5) &= \texttt{01010101} \\ \phi(x_2) &= \texttt{1111!!**} & \phi(x_6) &= \texttt{00110011} \\ \phi(x_3) &= \texttt{10101100} & \phi(x_7) &= \texttt{00001111} \\ \phi(x_4) &= \texttt{!1!1001*}\end{aligned}$$

Notice that in the extended assignment, the simple assignments φ_1, φ_3, φ_5, φ_6, and φ_8 falsify $\mathcal{F}_{\text{example}}$, while φ_2, φ_4, φ_7 satisfy $\mathcal{F}_{\text{example}}$. In other words, although the formula consists of seven Boolean variables, one can cover the whole search space using a generic assignment to only three variables.

Some of the simple assignments have, after unit propagation, one or more variables that are in the conflict state. For these simple assignments it holds that the values assigned to the non-conflicting variables falsify the formula. During unit propagation, we can (using bitwise Boolean operations again) efficiently maintain a bit mask that contains a `1` on positions where the corresponding simple assignments falsify the formula. We will call this bit mask the *conflict mask*. We use this bit mask for various learning rules that we will introduce later. During unit propagation, we also use the conflict mask to check whether each of the simple assignments contains a variable in the conflict state. In that case, unit propagation terminates.

There are some reasons for why generic assignments are useful. One of them is in the following theorem, of which we omit the proof.

Theorem 1. *The following two properties about satisfiability and bit-conflicts in unit propagation hold:*

1. *If every bit is set to* `1` *in the conflict mask that is obtained after having executed unit propagation on a generic assignment of a set of variables in a formula, then that formula is unsatisfiable.*
2. *Moreover, if a simple assignment in a multi-bit assignment falsifies the formula, then it is always the case that after executing unit propagation, one of the variables in that assignment is conflicting.*

This yields an easy method to detect unsatisfiability. After unit propagation, our solver simply checks whether the conflict mask consists of only `1`s. If so, the solver exits and gives the answer that the formula is unsatisfiable. Otherwise, (if it is also the case that among the simple assignments there is no satisfying assignment) the solver continues by running the learning rules: Apart from detecting whether the formula is satisfied or falsified, one can analyze the outcome of the unit propagation procedure and try to learn extra clauses from this. We will see how to do this in Section 3.

2.3 Branching and Merging

Inspiration for our solver has been drawn from the formula checker HeerHugo [3], and Stålmarck's theorem prover [8,2]. The most important idea of Stålmarck's prover is the *dilemma rule*. In a propositional proof, it is possible to split the proof into two branches, where in both branches we assume a hypothesis that is the complement of the hypothesis in the other branch. Then, we can try to prove or disprove both branches.

The interesting part of the dilemma rule comes from the idea of *merging* two branches back into one branch: after no more derivations are possible in both of the branches, then we can analyze and compare the results of both branches and draw conclusions that can be introduced in the merged branch. Stålmarck's prover uses various derivation rules and merging rules.

Our approach of using generic assignments for SAT-solving corresponds to the branching/merging idea of Stålmarck: in our case, the simple assignments that constitute the generic assignment are the branches. For this reason, in the remainder we will often refer to a simple assignment as a "branch". Our derivation rules consist of the multi-bit unit propagation algorithm, and we use various rules to add extra clauses to our formula. These can be seen as the merging rules.

HeerHugo is a propositional formula checker that is based on the dilemma rule that Stålmarck uses. We borrow some of the merging rules used in HeerHugo, which we will discuss in Section 3.

3 Learning Rules

Our solver tries to learn new clauses by applying learning rules to the assignment that results after executing multi-bit unit propagation on a generic assignment.

We drew our inspiration for the leaning rules from HeerHugo [3]. We have four major learning steps: (1.) learning unit-clauses, (2.) learning binary equivalences, (3.) learning binary clauses, and (4.) learning conflict clauses. We will discuss them in that order.

As we will see, the first three learning rules are only applicable because we use generic assignments. The last rule in principle does not require generic assignments, but is tractable because our assignments are generic.

For the sections that follow, we will use the following notions and notational conventions: With a "non-conflicting branch", we mean a branch where none of the variables is assigned to the bit-conflict state. We will denote the conflict mask by m_{conflict}. For the bitwise OR and AND operations, we use `OR` and `AND` respectively. The first and second bitwords of the representation of a multi-bit assignment for a variable x_i will be denoted by respectively $\phi^+_-[x_i]$ and $\phi^+_-[\neg x_i]$. Moreover, often we want to disregard those branches for which m_{conflict} is set to `1`. Therefore we define $\phi^*[x_i]$ as m_{conflict} `OR` $\phi^+_-[x_i]$. $\phi^*[\neg x_i]$ is defined analogously. Lastly, let $\mathbf{1}$ denote the bit word consisting of only ones.

3.1 Learning Unit-clauses

The *unit-clause* learning rule is the most simple of our learning rules: we simply check for every variable x_i whether $\phi^*[x_i] = \mathbf{1}$ or whether $\phi^*[\neg x_i] = \mathbf{1}$. In the former case, we can add (x_i) to the formula, and in the latter case, we can add $(\neg x_i)$. Alternatively, we can also choose to remove the appropriate clauses and literals. That is how it is done in BitFall.

Detecting unit clauses is done in $O(n2^b)$ time: we simply check for all n variables if it is set to `1` (`0`) in all 2^b branches. Assuming that all 2^b branches can be checked concurrently, the complexity reduces to $O(n)$.[2]

Example 2. Assume that we have the following assignment ϕ after executing multi-bit unit propagation on a formula $\mathcal{F}(x_1, \ldots, x_7)$, where x_5, x_6, and x_7 are the variables that have been selected for the generic assignment:

$$\begin{array}{ll} \phi(x_1) = \texttt{*0!0!11!} & \phi(x_5) = \texttt{01010101} \\ \phi(x_2) = \texttt{11!1!0*!} & \phi(x_6) = \texttt{00110011} \\ \phi(x_3) = \texttt{11*1*001} & \phi(x_7) = \texttt{00001111} \\ \phi(x_4) = \texttt{!101*11*} & m_{\mathsf{conflict}} = \texttt{10101001} \end{array}$$

Variable x_4 has values `1`, `1`, `1` and `1` on the branches where $m_{\mathsf{conflict}} = \texttt{0}$. Because all four of these values are 1, we add the clause (x_4) to $\mathcal{F}$, remove all other clauses containing $\neg x_4$ from $\mathcal{F}$, and remove all occurrences of literal x_4 from $\mathcal{F}$.

[2] In the remainder of this text, we will for this reason omit the 2^b factor when doing time-complexity analysis.

3.2 Learning Binary Equivalences

Through generic assignments and multi-bit unit propagation, it is possible to learn about equivalences between variables. Equivalences between variables are easy to detect.

For each pair of variables $(x_i, x_j), i \neq j$ that are both not unassigned on any of the non-conflicting branches, we check whether $\phi^*[x_i] = \phi^*[x_j]$. If that is true, we know that $x_i \leftrightarrow x_j$, so we can add this information to the formula. Also, we can check whether $\phi^*[x_i] = \phi^*[\neg x_j]$, from which we infer $x_i \leftrightarrow \neg x_j$. In BitFall we do not add any clauses, but we make substitutions instead.

Naively, checking this rule for all pairs of variables is done in $O(n^2)$ time. By sorting the array of variables according to their assignment, and by performing a binary search for each variable, this binary equivalence check can be improved to $O(n \log n)$ time.

In Example 2, from the application of this rule we would learn the equivalences $x_1 \leftrightarrow \neg x_3 \leftrightarrow x_7$.

3.3 Learning Binary Clauses

Binary clauses can be learned by checking for each pair of two variables (x_i, x_j) with $i \neq j$ if it is true that $\phi^*[x_i]$ `OR` $\phi^*[x_j] = \mathbf{1}$. In that case we can conclude that $(x_i \vee x_j)$ must be true. Three analogous learning rules can be applied for learning $(\neg x_i \vee x_j)$, $(x_i \vee \neg x_j)$ and $(\neg x_i \vee \neg x_j)$.

We need to apply these four rules on every pair of variables, so this method for detecting binary clauses runs in $O(n^2)$ time.

To illustrate this learning rule, let us take a look at Example 2 again. We see that for variables x_2 and x_3, x_3 is assigned `0` on all non-conflicting branches where x_2 is assigned `0`, regardless of whether we fill in a `1` or a `0` on places where x_2 is assigned a `*`. Therefore we can conclude $(\neg x_2 \vee x_3)$. Likewise, we can infer from this learning rule the clauses $(x_2 \vee x_1)$, $(\neg x_1 \vee x_5)$, $(x_7 \vee x_3)$, $(\neg x_3 \vee x_5)$, along with some other binary clauses that follow directly from applying the two other learning rules that we discussed above.

3.4 Learning Conflict Clauses

Not only is it possible to learn from non-conflicting branches, but also from conflicting branches. If we execute multi-bit unit propagation on a generic assignment, and if the result is that a certain branch φ_i the assignment falsifies the formula, then clearly the formula is not satisfiable on the assignment in branch φ_i that we gave to the generically assigned variables. If we pick n variables for the generic assignment, then for any conflicting branch, we can learn (and add to the formula) a clause of n literals. Often, it is also possible to learn clauses of less variables: this can be done if certain groups of branches are conflicting.

Take a look at Example 2 again. Branches φ_1, φ_3, φ_5, and φ_8 are conflicting. Branches φ_1 and φ_3 are the only branches for which variables x_5 and x_7 are both assigned a `0`. Because both branches are conflicting and because our assignment

to variables $x_5, \ldots, x_7$ is generic, it must be that in a satisfying assignment, x_5 and x_7 can not both be `0`. Hence, we can add $(x_5 \vee x_7)$ to our formula. Likewise, by branches φ_1 and φ_5 we can add $(x_5 \vee x_6)$; by branch φ_1 we have clause $(x_5 \vee x_6 \vee x_7)$; by branch φ_3 we have $(x_5 \vee \neg x_6 \vee x_7)$; branch φ_5 gives us $(x_5 \vee x_6 \vee \neg x_7)$; and finally branch φ_8 gives us $(\neg x_5 \vee \neg x_6 \vee \neg x_7)$. As can be seen, some of the clauses that can be learned from this rule are a subset of other clauses that can be learned from this rule. Of course, one should always make sure to only add the smallest possible clauses because they are more general. Smaller clauses can be obtained by looking at larger sets of conflicting branches.

In order to efficiently implement this conflict learning rule that we just described, we use a precomputed table to tell us which binary, ternary, ..., b-ary clauses we can precisely learn from a given m_{conflict}. Our algorithm simply checks for each entry in the precomputed table if the conflict mask has `1`s in the correct positions.

4 Description of the SAT-solver

We are now ready to describe the workings of BitFall globally. In Algorithm 1, $\mathcal{F}$ is the formula and ϕ is an assignment to the variables of $\mathcal{F}$.

Algorithm 1 BitFall

```
1: loop
2:    φ := GenericAssignment(F)                    {Discussed in Section 4.1}
3:    φ := UnitPropagation(F, φ)                   {Discussed in Section 2.2}
4:    if φ satisfies F on one or more branches/bit positions then
5:       return SATISFIABLE
6:    if φ falsifies F on all branches/bit positions then
7:       return UNSATISFIABLE
8:    F := Learn(F, φ)                             {Discussed in Section 3}
```

Applying the learning rules will result in extra clauses being added to the formula. Adding clauses to the formula is useful, because the multi-bit unit propagation that is executed in subsequent iterations of the algorithm will then be able to propagate more unit-clauses and hence infer more truth-values. When adding a clause to the formula, additionally our solver always applies some rules for simplifying the formula.

In the following two sections, we will explain the variable selection heuristics and the formula simplification rules.

In the remainder of the text, we assume for simplicity that if we run the algorithm on a machine with a 2^b-bit architecture, then we choose b as the number of variables for our generic assignment. Generalizations for lower and higher numbers of variables are easily made.

4.1 Variable Selection

The goal of the solver is to learn as much as possible as quickly as possible, so choosing the right variables is important. Typically we are looking for variables to occur often together in the same clause. We used two variable selection strategies: straightforward uniform random selection and a more advanced strategy using a selection heuristic. We will refer to the latter strategy as "the heuristic method".

For both strategies holds that when the solver has learned that a variable must necessarily be true or false (this happens: see Section 3.1), then selecting this variable will not give any useful information. Therefore, these variables are excluded in the selection process.

The uniform random selection method simply selects at random five different variables. The heuristic method first selects one random variable with probability proportional to the occurrence-frequency of this variable in the formula. This way, variables that occur more often in the formula have higher probability of being selected. We already noted that it is probably better to select variables that occur together in the same clause. Therefore, the other four variables are selected based on how often they occur together with the first variable. The probability that any variable is selected, is proportional to the number of clauses containing both this variable and the first selected variable. The heuristic method will typically select variables that are tighter interrelated than when using random selection. This should increase the chance of learning information about relations between these variables and the formula as a whole.

4.2 Formula Simplification Rules

When clauses are learned, we want to add them to our formula. However, to make sure we add no redundant information, we must first scan the formula to see if any similar clauses occur in it. Based on the clauses we find during this scan, we may undertake various actions.

These rules for deciding what actions to take for which clauses, are captured in two different *formula simplification rules*: *subsumption* and *self-subsumption* (or *ad-hoc resolution*). Our inspiration to use these rules comes from [3], where the rules are described only for clauses of two and three literals. We learn larger clauses in some cases, so we use generalized variants of these rules.

Subsumption If we want to add a clause C, it could be that there is already a clause C' in the formula with $C' \subset C$. In this case it is obviously not necessary to add C to the formula, simply because C' implies C.

The converse of this rule is also implemented: If we want to add a clause C, it could be that there is already a clause C' in the formula with $C \supset C'$. In this case we can remove C' from the formula and add C.

Self-subsumption Resolution is a widely studied inference rule for automated theorem proving. It was first introduced in [9]. We apply resolution to CNF-clauses in an on-the-fly manner. In the literature, this variant of resolution is often referred to as *self-subsumption*.

Self-subsumption works as follows. Suppose that we want to add a new clause C_1 with m literals, and there is already a clause C_2 in the formula with m' literals for which it holds that:

1. there is exactly one literal l in C_2 that is the negation of a literal $\neg l$ in C_1;
2. if $m' \leq m$, all the literals in C_2 except l also occur in C_1;
3. if $m' \geq m$, all the literals in C_1 except $\neg l$ also occur in C_2.

Now for the case $m' = m$, we can simplify the formula by removing C_2 from it, not adding C_1, and adding the clause C_2' which is clause C_2 with the literal l removed from it. In the case that $m' < m$, we do not add C_1, but we instead add C_1': C_1 with literal $\neg l$ removed from it. Finally, in the case that $m' > m$, we do add C_1, but we also remove literal l from C_2.

5 Experiments

Many of the algorithms and heuristics used in our SAT-solver have been implemented only to show that they work (proof-of-concept). Although some effort has been spent on optimizing the SAT-solver for time- and space-efficiency, we think that there is still room for improvement. For example, a significant speed-up could be achieved by optimizing the underlying data structures to better match their usage. However, doing so would require a complete rewrite of the code-base. Furthermore, we have only looked at methods that benefit from the multi-bit concept (i.e. the learning rules discussed above). It is also possible to enhance the solver further with common SAT-solving tricks and extensive preprocessing steps, but that is beyond the scope of our research.

Since our SAT-solver is not fully optimized for speed, we recorded both the duration and the number of variable assignment iterations that were spent per problem instance. The SAT-solver was tested using a selection of SAT-problem instances from SATLIB[3]. The tests were executed on a Core 2 Duo E6600 with 2 GB RAM running Ubuntu Linux 8.04 and each problem instance was given 10 seconds to reach a conclusion.

The first set of tests was intended to reveal how the problem size impacts performance. For this test we enabled all the solver features and used 5 generic assignment variables. We measured how many iterations and how much time was required to solve the problems. The results are given in Figure 2 and represent the minimal number of iterations and minimal duration that was required to solve 25%, 50%, 75% and 100% of the problems. As expected, the results indicate that a linear increase in the problem size leads to an exponential increase of the number of iterations and time required. In this case all the problems are satisfiable, but the same results have been obtained for unsatisfiable problem instances.

We also looked at how learning with different amounts of generic assignment variables influences the ability to solve problems. For this test, we enabled all

[3] SATLIB - The Satisfiability Library `http://www.satlib.org/`

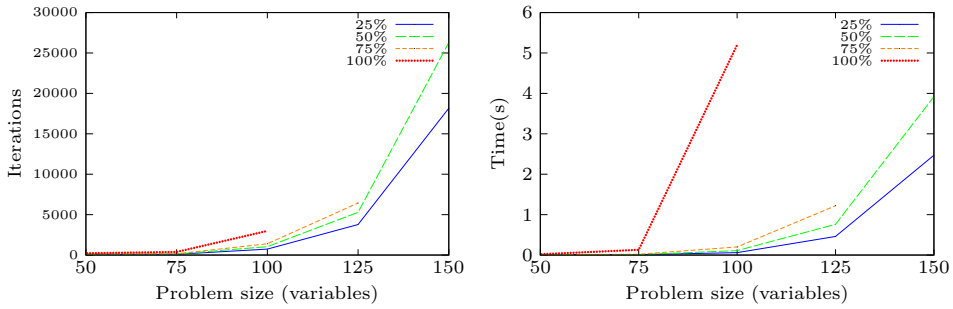

Fig. 2. Impact of problem size on iterations (left) and duration (right). The lines represent the minimal number of iterations and minimal duration that was required to solve 25%, 50%, 75% and 100% of the problems.

the features of the solver. With one variable, the solver was unable to solve problems of 20 and more variables. The results for two to five variables are given in Figure 3. It is interesting to see that the solver is able to solve 90% of the problems with 50 variables, using only 2 variables to make generic assignments.

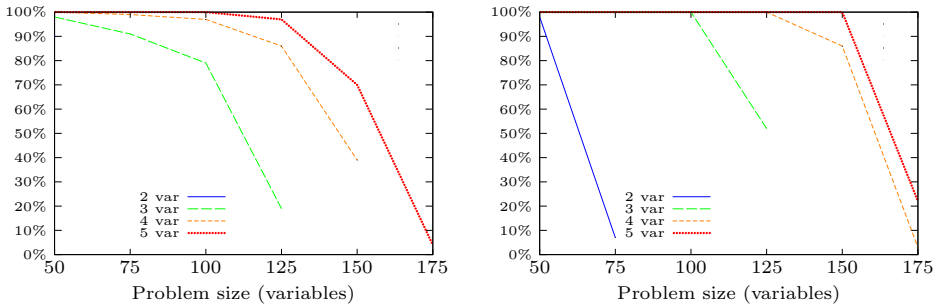

Fig. 3. Influence of generic assignment variables on solver performance for satisfiable (left) and unsatisfiable (right) instances. The percentage of solved instances for given problem size is plotted for the number of generic assignment variables used (2 to 5).

Concerning the performance of the solver on satisfiable instances versus unsatisfiable (uniform random 3SAT) instances, from Figures 3 we can see that the solver is more successful at unsatisfiable instances.

As a next experiment, a solver using the heuristic variable selection method has been compared to a solver that uses the uniform random variable selection method. In this case, all the solver features were set enabled and we used 5 generic assignment variables in all cases. Figure 4 (left) reveals that the heuristic significantly improves the performance of the solver.

Lastly, we also tested how much influence different learning techniques have on the performance of the solver. As discussed earlier, we can learn unit clauses,

binary clauses, binary equivalences and conflict clauses. In this experiment we enabled the variable selection heuristic and learned from 5-variable generic assignments. The maximum conflict clause learning length was also set to 5. The results are given in Figure 4 (right). The results show that learning binary equivalences does not have a significant impact on the performance. It is interesting to see that learning implications improves performance significantly when conflict clause learning is disabled. However, due to the computation cost of learning implications, it is not good for the runtime if it is used in combination with conflict clause learning. Conflict clause learning is definitely the technique that improves performance of the solver the most.

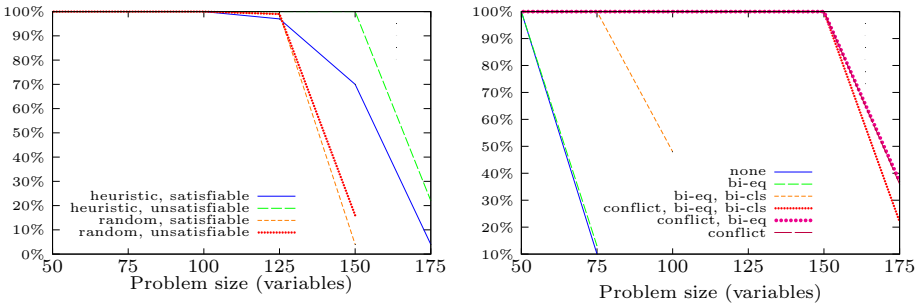

Fig. 4. Impact is measured as the percentage of solved instances as a function of the problem size. (Left) Impact of variable selection method on solver performance. (Right) Impact of learning rules on solver performance.

6 Conclusion

We have explored the idea of combining multi-bit SAT-solving with clause-learning techniques. In short, our approach combines the Stålmarck's proof procedure with generic assignments. Although our current implementation only learns clauses from the unit clauses in the extended generic assignments, many hard uniform random 3-SAT formulae could be solved. It would be interesting to see how our solver BitFall performs on structured instances. We expect, for example, that the binary equivalence learning rule will be much more crucial to success in such cases.

A natural direction for future work lies in improving the solver. First and foremost, we want to study whether full learning (not only from unit clauses) can be implemented efficiently. Another possible direction for future research includes finding good heuristics for picking the variables to use in the generic assignment. Finally, it should be possible to attain a very significant constant-factor speedup by some thorough code optimization. A complete rewrite of the codebase is probably the first thing to do: our current solver is built on UnitMarch, and some of the architecture of UnitMarch turned out to be not that well-suited to implement some of the necessary procedures efficiently.

References

1. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM **5**(7) (1962) 394–397
2. Sheeran, M., Stålmarck, G.: A tutorial on Stålmarck's proof procedure for propositional logic. In Gopalakrishnan, G., Windley, P., eds.: Proceedings 2nd Intl. Conf. on Formal Methods in Computer-Aided Design, FMCAD'98, Palo Alto, CA, USA, 4–6 Nov 1998. Volume 1522. Springer-Verlag, Berlin (1998) 82–99
3. Groote, J.F., Warners, J.P.: The propositional formula checker heerhugo. In: 691. Centrum voor Wiskunde en Informatica (CWI), ISSN 1386-369X (31 1999) 18
4. Heule, M.J.H., van Maaren, H.: From idempotent generalized boolean assignments to multi-bit search. In Marques-Silva, J.P., Sakallah, K.A., eds.: Theory and Applications of Satisfiability Testing - SAT 2007. Volume 4501 of Lecture Notes in Computer Science., Springer (2007) 134–147
5. Heule, M.J.H., van Maaren, H.: Parallel sat solving using bit-level operations. Journal on Satisfiability, Boolean Modeling and Computation **4** (2008) 99–116
6. Hirsch, E.A., Kojevnikov, A.: UnitWalk: a new SAT solver that uses local search guided by unit clause elimination. Technical Report PDMI preprint 9/2001, Steklov Inst. of Math. at St.Petersburg (2001)
7. Heule, M.J.H.: SmArT solving: Tools and techniques for satisfiability solvers. PhD thesis, TU Delft (2008)
8. Stålmarck, G.: A system for determining propositional logic theorems by applying values and rules to triplets that are generated from a formula. Swedish Patent No. 467 076 (approved 1992), U.S. Patent No. 5 276 897 (approved 1994), European Patent No. 0403 454 (approved 1995) (1989)
9. Robinson, J.A.: A machine-oriented logic based on the resolution principle. J. ACM **12**(1) (1965) 23–41