A General Nogood-Learning Framework for Pseudo-Boolean Multi-Valued SAT

Siddhartha Jain
Brown University, Providence, RI 02912
sj10@cs.brown.edu

Ashish Sabharwal and **Meinolf Sellmann**
IBM Watson Research Center, Yorktown Heights, NY 10598
{ashish.sabharwal,meinolf}@us.ibm.com

Abstract

We formulate a general framework for pseudo-Boolean multi-valued nogood-learning, generalizing conflict analysis performed by modern SAT solvers and its recent extension for disjunctions of multi-valued variables. This framework can handle more general constraints as well as different domain representations, such as interval domains which are commonly used for bounds consistency in constraint programming (CP), and even set variables. Our empirical evaluation shows that our solver, built upon this framework, works robustly across a number of challenging domains.

Introduction

Jain, O'Mahony, and Sellmann (2010) recently introduced a new nogood learning approach for multi-valued satisfaction (MV-SAT) problems. This approach was shown to infer significantly stronger nogoods than those inferred by a mechanism that is based on a Boolean representation of a multi-valued problem. Like earlier methods, the learning approach is based on an implication graph where nodes represent variable domain events and edges represent implications inferred by the clauses or constraints of the given problem. One of the novelties of Jain et al. was the sole focus on variable *inequations* to infer minimal reasons for a failure.

In this paper, we investigate why the particular use of inequations results in stronger nogoods and we formulate a general framework for multi-valued nogood-learning that can handle more general constraints, and also different domain representations, such as interval domains, which are commonly used for bounds consistency in constraint programming (CP). This is an essential step towards an integration of pseudo-Boolean and multi-valued SAT.

SAT vs. CP. Although both are concerned with feasibility problems, state-of-the-art SAT and CP methods differ significantly in style and philosophy. In SAT, we use one fixed, simple input format, allowing only one type of constraints, namely clauses, and variables that can take only two different values. To model a real-world problem, any structure that it may exhibit must be crushed into a number of loosely connected atomic parts: Boolean variables and clauses.

In CP, on the other hand, the input format is not fixed a priori. Practically all CP solvers support variables that can take more than just two values, and many support more involved constraints, such as linear inequalities on two or more numeric variables, element constraints, or even global constraints such as all-different, shortest path, knapsack, or grammar constraints.

Arguably, CP solvers ought to have a significant advantage over SAT solvers, given the additional benefit of being provided with structural information of the problem to be solved. Interestingly, CP solvers that are based on SAT reformulations are highly competitive nevertheless, such as `Sugar` (Tamura, Tanjo, and Banbara 2008), which won the CP global constraints competition in two consecutive years (CSP Competition 2008 2009). With important exceptions, it appears that approaches that exploit structure "just in case" tend to be more expensive while approaches that improve inference "just in time" are often more efficient.

SAT-X Hybrids. Consequently, hybrid approaches have been introduced, such as pseudo-Boolean solvers (e.g., (Dixon, Ginsberg, and Parkes 2004)) supporting inequalities over binary variables, SAT modulo theories or SMT "attaching" additional theories to a SAT solver (Nieuwenhuis, Oliveras, and Tinelli 2006), and the award-winning lazy clause generation approach (Ohrimenko, Stuckey, and Codish 2007). In the adjacent field of integer programming, the SCIP solver (Achterberg 2004) computes nogoods based on an analysis of the linear program at the "conflict" and an analysis of a Boolean-SAT like inference graph. Another related contribution is the recently proposed multi-valued SAT solver `CMVSAT-1` which strengthens nogood learning by directly incorporating the knowledge that each variable must take exactly one out of a number of values (Jain, O'Mahony, and Sellmann 2010).

In this paper, we analyze the nogood learning component of `CMVSAT-1` and generalize the approach by providing a framework for strong multi-valued nogood learning. We identify sufficient conditions for constraint propagators to support nogood learning effectively. Moreover, we show that the generalized framework is capable of handling various domain representations, such as interval domains and set variables, and associated propagators that enforce bounds consistency. We note that, unlike SMT solvers,

this work deals with generalized constraints that are native to the solver, and performs nogood-learning directly on them.

CMVSAT-1

We begin by briefly reviewing the nogood learning component of *CMVSAT-1*, in the context of multi-valued SAT.

Definition 1 (Multi-Valued Variables). A *multi-valued variable* X_i is a variable that takes values in a finite set D_i called the *domain* of X_i.

Definition 2 (Multi-Valued Clauses). Given a multi-valued variable X_i and a value v, we call the constraint $X_i = v$ a *variable equation* on X_i; the variable equation is *satisfiable* iff $v \in D_i$. Further, given a set of multi-valued variables $X = \{X_1, \ldots, X_n\}$ and a set $T \subseteq X$, a *clause over* T is a disjunction of variable equations on variables in T.

Definition 3 ((Partial) Assignments and Feasibility). Given a set of multi-valued variables $X = \{X_1, \ldots, X_n\}$, let D be the union of all their domains $D_1, \ldots, D_n$. Furthermore, let S, T be subsets of X.

- A function $\alpha : S \to D$ is called an *assignment* of variables in S. α is called *partial* iff $|S| < n$, and *complete* otherwise. If $|S| = 1$, α is called a *variable assignment*.
- α is called *admissible* iff $\alpha(X_i) \in D_i$ for all $X_i \in S$.
- α is called *feasible* w.r.t. a clause c over T iff $T \setminus S \neq \emptyset$ or if there exists a variable equation $X_i = \alpha(X_i)$ in c.
- Given a clause c, a complete, admissible, and feasible assignment for c is called a *solution* to c.

Definition 4 (Multi-Valued SAT Problem). Given a set X of multi-valued variables and a set C of clauses over subsets of X, the *multi-valued SAT problem (MV-SAT)* is to decide whether there exists an assignment that is a solution to all clauses in C.

Multi-Valued Nogood Learning

Boolean SAT nogood learning is commonly based on a so-called implication graph. Analogously, the conflict analysis proposed by Jain, O'Mahony, and Sellmann (2010) is also based on a graph. Curiously, this graph was set up to only represent the implications that result in variable *inequations*. That is to say, even when propagation could show that some variable *must* take a certain value in its domain, the information added to the graph is that it cannot take any of the remaining values.

Note that this is not merely a matter of choice akin to representing a binary CP constraint as positive list of allowed tuples or as a negative list of disallowed tuples. The reason the proposed approach is based on variable inequations *only* is that this is the *most atomic information* that is communicated between constraints. This way, when a reason for a conflict is sought, it can be traced back to the *minimal domain reductions* that are needed to reduce the domains further. To make the discussion less abstract, let us consider the following example.

Example 1. Consider a multi-valued SAT problem with five variables $X_1, \ldots, X_5$ with domains $D_1 = D_2 = D_4 = \{1,2,3\}, D_3 = \{1, \ldots, 7\}$, and $D_5 = \{1,2\}$. The clauses are

$$(X_1 = 2 \,||\, X_1 = 3 \,||\, X_4 = 2 \,||\, X_4 = 3)$$
$$(X_2 = 1 \,||\, X_2 = 3 \,||\, X_3 = 1 \,||\, X_3 = 3 \ldots 7)$$
$$(X_4 = 1 \,||\, X_4 = 2 \,||\, X_3 = 6,7)$$
$$(X_4 = 1 \,||\, X_4 = 3 \,||\, X_5 = 2)$$
$$(X_5 = 1 \,||\, X_3 = 6,7)$$

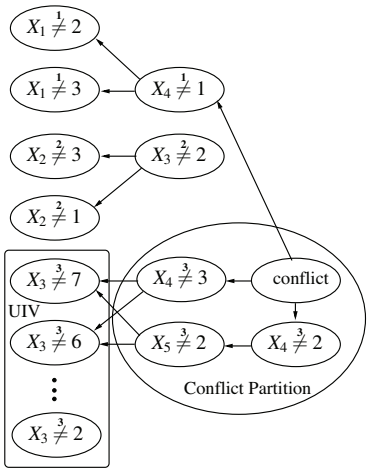

Figure 1: MV-SAT Implication Graph.

Figure 1 depicts the implication graph that emerges when we branch by setting or "committing" $X_1 \leftarrow 1$, then $X_2 \leftarrow 2$, and finally $X_3 \leftarrow 1$, which leads to a conflict. □

An important novelty introduced by Jain, O'Mahony, and Sellmann (2010) is the notion of a unit implication variable:

Definition 5 (Unit Implication Variable). Given a multi-valued SAT problem and an implication graph G, a variable X is called a *unit implication variable (UIV)* if the following holds: if all nodes associated with X are removed from G, then there exists no path from the conflict node to any branch node on the last (i.e., deepest) branch level.

In contrast to Boolean SAT, in multi-valued SAT a clause is unit as soon as all remaining variable equations regard the same variable. Consequently, a nogood computed by analyzing the implication graph is no longer defined by a *cutpoint* in the graph, but by a *cutset* of nodes that all regard the same variable. As long as we compute nogoods based on such cutsets, we can be sure that the corresponding nogood is "unit" after backtracking and will cause propagation.

In our example, X_3 is a UIV. Based on its associated cutset, we can compute a conflict partition and set the nogood as the negation of the conjunction of all variable inequations that have a direct parent in the conflict partition. We thus find the multi-valued clause $(X_4 = 1 \,||\, X_3 = 6 \,||\, X_3 = 7)$. After backjumping to the next deepest level after the conflict level in our learned clause (in our case level 1), this clause is unit and prunes the domain of X_3 to $D_3 = \{6,7\}$. That is, equipped with this nogood the solver does not need to branch on $X_3 \leftarrow 2$, $X_3 \leftarrow 3$, $X_3 \leftarrow 4$, and $X_3 \leftarrow 5$ as all of these settings would fail for the very same reason as $X_3 \leftarrow 1$

did. Note that the same strength of inference would not have been achieved had we simply noted $(X_3 = 1)$ as a sufficient reason for inferring $(X_4 \neq 3)$ and $(X_5 \neq 2)$.

A Generalized Framework

We now generalize this approach and identify sufficient conditions under which strong nogoods can be learned effectively. In multi-valued SAT, the only constraints allowed are multi-valued clauses. Moreover, nogood learning requires that branching decisions regard just one variable, and propagation is limited to unit propagation only. By generalizing the nogood learning component recapitulated thus far, we liberate ourselves from these restrictions and are able to handle more general constraints as well as different variable domain representations. This is an essential step for the design of solvers that can handle the pseudo-Boolean multi-valued SAT problem.

A Formal View of Constraint Programming

A canonical view in CP is to think of the set of constraints as being split in two classes. The first class of constraints is that of *primitive constraints*. Originally primitive constraints were introduced as constraints that can be directly expressed by the respective domain representation. However, from a formal point of view it is actually often convenient to use primitive constraints in the opposite way, namely to express what domain representation is used.

For example, let us consider all unary constraints as primitive. Obviously, any unary constraint can be expressed as a truth table. Consequently, the constraint can be directly encoded in the domain of the variable if the latter is represented as a list of allowed values. Alternatively, we can represent the current domain of a variable by such a primitive constraint. To give another example, assume that we consider unary at-most and at-least constraints as primitive (whereby implicitly we assume a given ordering on the values). This set of primitive constraints is equivalent to a domain representation that maintains only an interval of values for each variable, as is usually the case for approaches that maintain bounds consistency.

The other class of constraints is usually referred to as *secondary constraints*. It consists of all constraints that are supported but are not primitive. Now, the *propagation* of a secondary constraint C_s can be viewed as the process of entailment of new primitive constraints C_p from the conjunction of the existing primitive constraints and the secondary constraint that is propagated: $(\bigwedge_{C_i \in \text{PrimStore}} C_i) \wedge C_s \vdash C_p$. Note that, unlike learned clauses, these inferred constraints need to be removed from the constraint store upon backtracking.

Nogoods as Disjunctions of Negated Primitives

Let us revisit the nogood learning approach discussed in the previous section in the light of the above formal view of CP. All *unary inequations* are primitive constraints, and multi-valued clauses, i.e., *disjunctions of variable equations*, are the only secondary constraints.

Nodes in the implication graph represent primitive constraints, the outgoing arcs link a node to a set of primitive constraints that, when conjoined with some secondary constraint, allow the entailment of the corresponding primitive constraint. A conflict is reached when a subset of nodes in the graph represents a set of primitive constraints that contradict one another. Since the only primitive constraints are variable inequations, conflict sets are necessarily sets of unary constraints regarding the same variable that together exclude all potential values from its domain.

When such a set of conflicting nodes is found, any cutset whose removal from the graph disconnects the conflict nodes and the nodes directly introduced through branching defines a valid redundant constraint. Namely, for the primitive constraints in the cutset, it has been found that they contradict each other in the given constraint system as they entail the conflict. Consequently, we conclude that the conjunction of these primitive constraints must be false or, equivalently, that the *disjunction of the negation of these primitive constraints must be true*. Therefore, in this framework, nogoods take the form of disjunctions of negated primitive constraints.

Valid Cutsets in the Implication Graph. While any cutset results in a constraint that is redundant to the given system of constraints, in order to make progress we search for cutsets with a desirable property. Namely, we are interested in those redundant constraints *that allow us to entail new primitive constraints efficiently upon backtracking*. For traditional SAT and also in multi-valued SAT, this means that we try to learn a clause that will be *unit* upon backtracking so that unit propagation can be effective.

To generalize this notion, we require that the constraint reasoning system provides an efficient approach for finding cutsets with the property that the disjunction of negated primitive constraints that is implied after backtracking can be expressed efficiently as a conjunction of primitive constraints. We call such cutsets *valid*. In particular, we require that primitive constraints added upon branching are themselves valid cutsets as well.

Efficient Nogood Learning: Sufficient Conditions

To summarize, assume that the following conditions are met by our constraint reasoning system:

(1) The system distinguishes between primitive and secondary constraints. Propagation of secondary constraints is the entailment of new primitive constraints.

(2) Apart from adding new primitive constraints, secondary constraint propagators can efficiently provide a set of primitive constraints that is sufficient for entailing the new constraints.

(3) Conflicting sets of primitive constraints can be detected efficiently. Disjunctions of negations of primitive constraints are supported as secondary constraints.

(4) Negated primitive constraints can be expressed as a set (i.e., conjunction) of primitive constraints. Certain disjunctions of negated primitive constraints (namely, those arising from valid cutsets upon backtracking) can be succinctly represented as a set of primitive constraints.

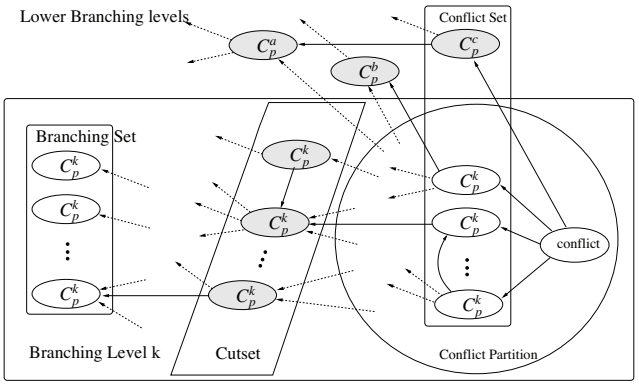

Figure 2: Abstract Implication Graph. We omit those parts of the graph that cannot be reached from the conflict node.

(5) Branching is executed as the addition of one or more primitive constraints. If a conjunction of primitive constraints is added, then the negation of this conjunction is a valid cutset in the sense of condition (4).

If these conditions are satisfied, we can efficiently learn strong nogoods in the following way (see Figure 2): We maintain an implication graph where nodes represent primitive constraints added by branching (5) or entailed by propagators (1). Nodes are associated with the branching level at which they were added or entailed. We add arcs from entailed primitive constraints to those primitive constraints that are needed for the entailment (2). When a conflict is reached (3), we compute a valid cutset on the last branching level that separates the conflict nodes (3) from the branching nodes (5). A nogood is inferred as the negation of the conjunction of these cutset nodes and nodes on lower branching levels that are adjacent to nodes in the conflict partition induced by the cutset. Note that the validity of the cutsets allows us, upon backtracking, to propagate the remaining disjunction of negated primitive constraints by expressing it as a set of primitive constraints (4).

Framework Application Example

The abstract procedure above now allows us to generalize nogood learning. The framework is general enough to handle constraints other than clauses (provided that condition (2) is met) as well as multi-valued variables. In fact, the system can even handle *set variables* whose domains (subsets of the power set over some universe of elements) can be represented as ranges, for example by using length-lexicographic bounds (Gervet and Van Hentenryck 2006).

Consider, for example, a constraint reasoning system where primitive constraints are unary at-most and at-least constraints on some variables (we call these *range variables*) and inequations for the other variables (we refer to these as *domain variables*). According to condition (3), we need to support disjunctions of at-most and at-least constraints on range variables and equations on domain variables in our system. Let us assume that, on top of these secondary constraints, the system also supports linear inequalities. For

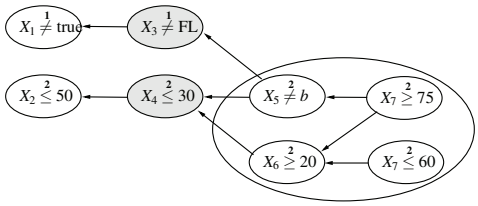

Figure 3: Implication Graph for Inequalities as Primitives.

both types of constraints it is trivial to satisfy condition (2). If we branch by splitting domains for range variables and by adding inequations for domain variables (5), all conditions are satisfied and we can use our nogood learning approach.

To illustrate how nogood learning in this exemplary system works, consider the following constraint satisfaction problem. We have seven variables $X_1 \in \{\text{true}, \text{false}\}$, $X_3 \in \{\text{NY}, \text{TX}, \text{FL}, \text{CA}\}$, $X_5 \in \{r, g, b\}$, and $X_2, X_4, X_6, X_7 \in \{1, \ldots, 100\}$, along with six constraints:

$$(X_1 = \text{true} \;||\; X_3 = \text{NY} \;||\; X_3 = \text{CA})$$
$$(X_2 \geq 60 \;||\; X_4 \leq 30)$$
$$(X_3 = \text{FL} \;||\; X_4 \geq 60 \;||\; X_5 = r \;||\; X_5 = g)$$
$$X_4 + X_6 \geq 50$$
$$(X_5 = b \;||\; X_6 \leq 10 \;||\; X_7 \geq 75)$$
$$X_6 + X_7 \leq 80$$

Suppose we first branch by adding the primitive constraint $X_1 \neq \text{true}$ (see Figure 3). From here we entail by the first secondary constraint that $X_3 \neq \text{FL}$ (and $X_3 \neq \text{TX}$ but we will not use this information). Say we branch on X_2 next and add $X_2 \leq 50$. By the second constraint we entail $X_4 \leq 30$ and, by the third and fourth constraint, we entail $X_5 \neq b$ and $X_6 \geq 20$. The fifth and sixth constraint then entail $X_7 \geq 75$ and $X_7 \leq 60$, two primitive constraints which obviously contradict each other. Analyzing the implication graph, we find that $X_4 \leq 30$ is a non-dominated cutset (in the sense of Jain, O'Mahony, and Sellmann (2010)) on the lowest branching level, and we learn the nogood $(X_3 = \text{FL} \;||\; X_4 \geq 31)$. Important for us is that this cutset is valid in the sense of condition (4), as the negation, $X_4 \leq 30$, inferred upon backtracking can obviously be expressed as a primitive constraint.

We would like to highlight two points. First, note that not any set of primitive constraints that regards the same variable results in a valid cutset. For example, assume that constraints $(X \geq 5)$ and $(X \leq 10)$ are part of a cutset and the disjunction of their negations, $(X \leq 4 \;||\; X \geq 11)$, is implied upon backtracking. This disjunction cannot be expressed as a conjunction of primitive constraints and therefore the underlying cutset is not valid. Second, in general, a valid cutset need not consist of primitive constraints that regard the same variable. For example, it is conceivable that a constraint reasoning system treats inequalities over two variables $(X \leq Y)$ as primitive constraints. A disjunction of negated primitive constraints would be perfectly fine here if it can be expressed as conjunction of primitive constraints over multiple variables. Consequently, the fact that a cutset consists of primitive constraints over the same variable is, in general, by itself neither necessary nor sufficient for validity.

Post-Processing the Implication Graph

As in Boolean SAT, we can strengthen the learned nogoods by computing a minimal (not necessarily minimum) conjunction of negated literals in our nogood (note that these negated literals are primitive constraints!) for which propagation still results in a conflict (Een and Sörensson 2005). However, we can do even more when we assume that secondary constraints provide the corresponding functionality.

Consider the previous example. The final conflict is given by the conjunction of $X_7 \geq 75$ and $X_7 \leq 60$. It would have been sufficient for a conflict had we just inferred $X_7 \leq 74$, so let us replace the node $X_7 \leq 60$ with $X_7 \leq 74$. When the node $X_7 \leq 60$ was added to the implication graph, this happened as a result of propagating the sixth constraint $X_6 + X_7 \leq 80$. To achieve $X_7 \leq 74$ by propagation, it would have therefore sufficed to have the primitive constraint $X_6 \geq 6$. Suppose the implementation of the linear constraint can provide this minimal primitive constraint needed for inferring $X_7 \leq 74$. Then, we need to check what other implications the node $X_6 \geq 20$ was needed for. We observe that, to infer $X_7 \geq 75$ by the fifth constraint, we needed at least that $X_6 \geq 11$. When replacing the node $X_6 \geq 20$ we must add the strongest precondition for all constraints that have added incoming arcs to this node. We therefore replace the node with $X_6 \geq 11$.

Working our way through the graph this way, we find next that $X_4 \leq 30$ can be replaced by the stronger of the nodes $X_4 \leq 59$ and $X_4 \leq 39$. As we have now reached a node that is part of the nogood, we stop further minimization that would result from the weakening of $X_4 \leq 30$ to $X_4 \leq 39$. We finally learn the *strengthened nogood* $(X_3 = \mathrm{FL} \,||\, X_4 \geq 40)$.

Note that this post-processing of the implication graph has the potential to remove certain nodes entirely, for example when the initial domain bound of a range variable is enough to infer a bound on another that is sufficient to entail the conflict. Therefore, post-processing can also create new cutsets that dominate the ones in the non-processed implication graph. Consequently, post-processing should always take place *before* the cutset is computed and not (as we just did) after the nogood is learned.

Empirical Evaluation

We devised a constraint system, `CMVSAT-2`, based on the results in the previous section. The system provides for range and domain variables, whereby primitive constraints for range variables consists in at-most and at-least constraints. For domain variables, primitive constraints are inequations. Secondary constraints that we provide so far are linear inequalities and disjunctions of equations and range constraints, such as $(X_1 \in [5 \ldots 60] \,||\, X_2 \in [37 \ldots 74] \,||\, X_5 = b \,||\, X_{10} \leq 15)$. Note that the latter constraints include disjunctions of negated primitive constraints.

We consider problems from three domains, each of which combines challenging combinatorial constraints with linear inequality constraints: 'quasigroup with holes' problem with costs (qwh-c), market split problem (msp), and a weighted version of N-queens problem (nqueens-w). We compare our solver `CMVSAT-2` with the pure SAT solver `MiniSat 2.2.0` (Eén and Sörensson 2004), the CSP solver `Mistral` (Hebrard 2008), and the mixed integer programming (MIP) solver `SCIP 2.0.1` (Achterberg 2004). In order to generate problem instances in the various formats required by the above solvers, we used the tool `Numberjack 0.1.10-11-24` (Hebrard, O'Mahony, and O'Sullivan 2010) to create a generic model. This model was solved directly by `Mistral` as the primary built-in solver for `Numberjack`, and was translated using `Numberjack` into ".cnf" and ".lp" formats for `MiniSat` and `SCIP`, respectively. All experiments were conducted on Intel Xeon CPU E5410 machines, 2.33GHz with 8 cores and 32GB of memory, running Ubuntu. We performed one run per machine at a time, with a 10 minute limit.

The translation to CNF format for `MiniSat` is similar to the one used by `Sugar` (Tamura, Tanjo, and Banbara 2008). It creates Boolean variables to capture relations of the form $X = v$ and $X \leq v$ for a variable X and a domain value v. The use of propositions for $X \leq v$ eliminates the quadratic blowup usually associated with the direct encoding of a CSP variable into a set of Boolean variables. It also allows implicitly for domain splitting as a branching strategy. Like `Sugar`, the size of CNF encodings is kept under control (where necessary) by using a "compact" encoding that breaks up long inequalities into a number of smaller ones consisting of only three variables each, by introducing auxiliary variables. E.g., $X_1 + X_2 + X_3 + X_4 + X_5 \leq 20$ is broken down into $Y_1 = X_1 + X_2; Y_2 = Y_1 + X_3; Y_2 + X_4 + X_5 \leq 20$ in our compact encoding. The market split problem brings out an interesting trade-off between encoding compactness and propagation efficiency, as we will shortly see.

Our main finding is that *`CMVSAT-2` is a robust general method that works well across a variety of domains*. Each of the other solvers we consider (SAT, CSP, and MIP solvers) may perform somewhat better than `CMVSAT-2` on one domain but they all suffer from a significant disadvantage in other domains. The results, averaged over 100 instances in each domain considered, are summarized in Table 1. In all cases, we used *impact based variable selection* for `CMVSAT-2`. For *value selection*, selecting the value with the least impact worked well in general, although the "disconovo-gogo" heuristic (Sellmann and Ansótegui 2006) was better on nqueens-w. Our implementation includes post-processing of the implication graph as discussed earlier, although on the benchmark set considered, `CMVSAT-2` performed quite well even without post-processing.

The **`qwh-c`** domain adds costs to the classic quasigroup with holes problem (Gomes and Shmoys 2002). We assign uniform random costs $C_{i,j}$ from $\{1..10\}$ to each cell and add one additional constraint $\sum_{1 \leq i,j \leq N} C_{i,j} X_{i,j} \leq (\sum_{1 \leq i,j \leq N} C_{i,j})/2$. For our experiments, we considered quasigroup instances of order 25 with 40% filled entries (i.e., 375 holes). In this regime, the unweighted problem is close to the known feasibility and hardness threshold, but still allows enough flexibility to have solutions with various costs. As we see from the first row of Table 1, `CMVSAT-2` is able to solve all 100 instances, requiring an average of only 0.55 seconds. The CNF translation for these instances, even with the "compact" encoding, was too large to generate or to read using `MiniSat`, as the cost inequality involves

problem domain	**CMVSAT-2** solved	runtime	nodes	MiniSat solved	runtime	nodes	Mistral solved	runtime	nodes	SCIP solved	runtime	nodes
qwh-c	100	0.55s	231	0	MEM	MEM	36	≥403.05s	≥2.9M	99	≥ 58.68s	≥65
msp-3	100	1.03s	10,305	100	19.41s	5,361	100	0.02s	7,527	100	0.54s	4,732
nqueens-w	100	9.08s	2,409	100	0.07s	2,018	92	≥125.51s	≥2.3M	100	106.03s	110
all	300	3.55s	–	200	≥206.49s	–	228	≥176.19s	–	299	≥55.08s	–

Table 1: Summary of results with a 10 minute cutoff, showing the average over 100 instances in each domain of the number of solved instances, the runtime, and the number of search nodes processed by each solver. The last row, marked "all", represents the total number of instances solved and the average runtime.

625 variables with coefficients and variable values ranging in $\{1..10\}$. SCIP can solve 99 instances but needs roughly a minute on average, while Mistral shows extreme variation in runtime and times out on 64 instances.

The **msp-3** domain represents the market split problem which is notoriously hard for constraint solvers. We used the standard Cornuejols-Dawande generator (Cornuejols and Dawande 1998) and created 100 instances of order 3, of which exactly 10 instances were satisfiable. As we see from Table 1, CMVSAT-2 can solve all 100 instances in roughly 1 second on average. Mistral and SCIP perform better on this domain. However, MiniSat is slower than CMVSAT-2 on average by 20x using the non-compact encoding. Interestingly, while MiniSat is in general able to process many more nodes per second than CMVSAT-2, the size of the input files significantly limits its propagation engine here despite techniques such as watched literals. Specifically, MiniSat takes roughly 20x longer to process roughly half as many nodes as CMVSAT-2, because of sheer encoding size. The compact CNF encoding, while significantly reducing the size, apparently hinders effective inference as it caused 57 instances to time out.

Finally, the **nqueens-w** domain adds weights to the classic N-queens problem. We assign uniform random weights $W_{i,j} \in \{1,\ldots,\text{maxweight}\}$ to the cells and augment the problem with the constraint $\sum_{\text{Queen on }(i,j)} W_{i,j} \geq 0.7 \times N \times \text{maxweight}$. Table 1 shows results for $N = 30$ and maxweight $= 10$. CMVSAT-2 can solve all 100 instances within an average of 9 seconds each. MiniSat fares better here, while Mistral times out on 8 instances and SCIP is slower than CMVSAT-2 on average by 11x.

Overall, Table 1 demonstrates that our general framework is capable of good performance across a variety of domains, while other solvers do not generalize as well. The last line of Table 1, for example, shows that CMVSAT-2 was able to solve all 300 instances considered within 4 seconds an average, while MiniSat could solve only 200 in over 206 seconds on average, Mistral could solve only 228 in over 176 seconds on average, and SCIP could solve 299 but needed over 55 seconds on average.

Conclusion

This work presented a generalized framework for performing SAT-style conflict driven nogood-learning in any setting where primitive and secondary constraints (and the corresponding consistency notions) supported by the system satisfy certain sufficient conditions. The framework shows how low-overhead nogood learning can be achieved in solvers that combine the benefits of multi-valued variables and pseudo-Boolean solvers. We believe that this is an important step towards the development of a new class of solvers that favor opportunistic learning over the expensive exploitation of structure even on problems with multi-valued variables and more elaborate constraints. Our empirical results attest to the robustness of this new class of solvers whose potential we are just beginning to explore.

References

Achterberg, T. 2004. SCIP - a framework to integrate constraint and mixed integer programming. Technical Report ZR-04-19, Zuse Institute Berlin.

Cornuejols, G., and Dawande, M. 1998. A class of hard small 0-1 problems. In *IPCO*, 284–293.

CSP Competition. 2008-2009. International CSP competition result pages.

Dixon, H. E.; Ginsberg, M. L.; and Parkes, A. J. 2004. Generalizing Boolean satisfiability I: Background and survey of existing work. *JAIR* 21:193–243.

Eén, N., and Sörensson, N. 2004. An Extensible SAT-solver. In *SAT-2005*, volume 2919 of *LNCS*, 333–336.

Een, N., and Sörensson, N. 2005. MiniSat v1.13 - A SAT Solver with Conflict-Clause Minimization, System description for the SAT competition.

Gervet, C., and Van Hentenryck, P. 2006. Length-lex ordering for set csps. In *AAAI-06*, 48–53.

Gomes, C., and Shmoys, D. 2002. Completing quasigroups or Latin squares: A structured graph coloring problem. In *Computational Symposium on Graph Coloring and Extensions*.

Hebrard, E.; O'Mahony, E.; and O'Sullivan, B. 2010. Constraint Programming and Combinatorial Optimisation in Numberjack. *CPAIOR-2010* 181–185.

Hebrard, E. 2008. Mistral, a constraint satisfaction library. In *3rd International CSP Solver Competition*, 31–40.

Jain, S.; O'Mahony, E.; and Sellmann, M. 2010. A complete multi-valued sat solver. In *CP-2010*, 281–296.

Nieuwenhuis, R.; Oliveras, A.; and Tinelli, C. 2006. Solving sat and sat modulo theories: From an abstract davis-putnam-logemann-loveland procedure to dpll(t). *J. ACM* 53(6):937–977.

Ohrimenko, O.; Stuckey, P. J.; and Codish, M. 2007. Propagation = lazy clause generation. In *CP-07*. Springer-Verlag.

Sellmann, M., and Ansótegui, C. 2006. Disco-Novo-GoGo: Integrating local search and complete search with restarts. In *AAAI-06*.

Tamura, N.; Tanjo, T.; and Banbara, M. 2008. System description of a SAT-based CSP solver Sugar. *Proceedings of the Third International CSP Solver Competition* 71–75.