A Complete Multi-Valued SAT Solver

Siddhartha Jain,[1] Eoin O'Mahony,[2] Meinolf Sellmann[1⋆]
[1] Brown University, Department of Computer Science
P.O. Box 1910, Providence, RI 02912, U.S.A.
sj10,sello@cs.brown.edu
[2] Cork Constraint Computation Centre
University College Cork, Cork, Ireland.
e.omahony@4c.ucc.ie

Abstract. We present a new complete multi-valued SAT solver, based on current state-of-the-art SAT technology. It features watched literal propagation and conflict driven clause learning. We combine this technology with state-of-the-art CP methods for branching and introduce quantitative supports which augment the watched literal scheme with a watched domain size scheme. Most importantly, we adapt SAT nogood learning for the multi-valued case and demonstrate that exploiting the knowledge that each variable must take exactly one out of many values can lead to much stronger nogoods. Experimental results assess the benefits of these contributions and show that solving multi-valued SAT directly often works better than reducing multi-valued constraint problems to SAT.

1 Multi-Valued SAT

One of the very successful solvers for constraint satisfaction problems (CSPs) is Sugar [23]. It is based on the reduction of CSP to the satisfiability problem (SAT). Sugar first encodes the given problem as a SAT formula and then employs MiniSAT [7] to solve the instance. Somewhat surprisingly, Sugar won the ACP global constraint competition in the past two years [21]. Our work is highly motivated by the success of this solver. Our objective is to provide a solver which could replace MiniSAT for reduction-based CSP solvers and work with even better efficiency by taking into account that CSP variables usually have non-boolean domains. To this end, let us begin by formally defining the multi-valued SAT problem.

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*. A variable equation* $X_i = v$ *is called* satisfiable *iff* $v \in D_i$.
- *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* X.

Example 1. Given variables X_1, X_2, X_3 with domains $D_1 = \{1, \ldots, 5\}$, $D_2 = \{\text{true}, \text{false}\}$, and $D_3 = \{\text{red}, \text{green}, \text{blue}\}$,

$$(X_1 = 1 \vee X_1 = 3 \vee X_1 = 4 \vee X_2 = \text{false} \vee X_3 = \text{red}) \tag{1}$$

is a multi-valued clause over X_1, X_2, X_3.

⋆ This work was supported by the National Science Foundation through the Career: Cornflower Project (award number 0644113).

Note that multi-valued clauses have no negation. A classic SAT clause would be written as $(X_1 = \text{false} \vee X_2 = \text{true} \vee X_3 = \text{false})$ in this notation. To save memory, it can be very helpful to encode large sets of allowed values by specifying the disallowed values only. E.g., in the example above we may prefer to write

$$((X_1 \neq 2 \wedge X_1 \neq 5) \vee X_2 = \text{false} \vee X_3 = \text{red}). \tag{2}$$

While the solver that we developed for this paper allows these inputs, they are not part of the problem definition itself to keep the theory free from confusing implementation details.

Definition 3 ((Partial) Assignments and Feasibility). *Given a set of multi-valued variables $X = \{X_1, \ldots, X_n\}$, denote with D the union of all domains $D_1, \ldots, D_n$. Furthermore, denote with S, T 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.
- *An assignment α is called* admissible *iff $\alpha(X_i) \in D_i$ for all $X_i \in S$.*
- *An assignment α of variables in S is called* feasible *with respect to a clause over T iff $T \setminus S \neq \emptyset$ or if there exists a variable equation $X_i = \alpha(X_i)$ in the clause.*
- *Given a clause c, a complete, admissible, and feasible assignment is called a* solution *for c.*

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

As MV-SAT allows boolean variables, the problem is at least as hard as SAT and is thus NP-hard. On the other hand, using standard CP encodings (direct encoding, support encoding, or order encoding [24, 10, 23]), we also know that MV-SAT can be reduced to SAT which makes the problem NP-complete.

Although the problems are equivalent in complexity (as all NP-complete problems are) and also very similar in structure, we consider MV-SAT an interesting generalization of SAT. In [1] it was shown that many problems which are currently being solved as SAT problems do in fact model problems with multi-valued variables. Moreover, many applications, especially in verification, naturally exhibit multi-valued variables. The purpose of this paper is to show that handling multi-valued variables directly rather than encoding them by means of boolean variables can lead to substantial improvements in computational efficiency.

2 Efficient Incremental Clause Filtering

The main inference mechanism in CP and SAT is constraint filtering. That is, for a given constraint we want to identify those variable assignments that cannot be extended to a solution for a given constraint and the current domains. The corresponding domain values are then removed and the process is iterated until no constraint can remove further domain values.

2.1 Unit Propagation in SAT

A classic SAT clause only ever removes a value (true or false) from a variable domain when all other variables in the clause have been assigned a value which does not satisfy the clause. In this case, we speak of a *unit clause* and the process of filtering clauses in this way is called *unit propagation*.

An elegant way of incrementally performing the filtering of SAT clauses was proposed in [18]. Each clause watches only two variable equations. In SAT, these are commonly referred to as *literals*. As long as both watched literals can be satisfied, no filtering can take place. When one watched literal cannot be satisfied anymore, we search for another second literal that can still be satisfied. Only when no second satisfiable literal can be found is the clause unit and we commit the only remaining variable assignment that can still satisfy the clause.

The beauty of this way of performing unit propagation is that it is not necessary to traverse all clauses that involve any variable that has been set to a value. In essence, the two watched literals give us an effective pre-check whether a clause can filter any values. Only when this pre-check fails, i.e. when one of the watched literals is affected, we need to perform any work. Otherwise the cost is not even constant per unaffected constraint, there is in fact no work to do for them at all. This last fact is key for handling problem instances with many constraints and for learning a large number of redundant clauses during search.

Note that we always start the search for a replacement watched literal at the old lost literal and then wrap around if the tail of the clause did not contain a valid support. This prevents us from looking at the same lost supports over and over again which would happen if we always started the search at the first literal in the clause. The method guarantees that each clause literal will be looked at most thrice on any path from the root to any leaf in the search tree. Note further that this scheme is very backtrack friendly as all non-unit constraints do not need to update their watched literals upon backtracking as their current supports are obviously still valid for any ancestor node in the search tree.

2.2 Watched Variable Equations

For multi-valued clauses we can use the very same approach. Note however that multi-valued clauses can trigger filtering earlier. As soon as all variable equations that can still be satisfied regard just one variable, all domain values of this variable that do not satisfy the clause can be removed from its domain. For example, consider the clause from earlier: $(X_1 = 1 \vee X_1 = 3 \vee X_1 = 4 \vee X_2 = \mathrm{false} \vee X_3 = \mathrm{red})$. As soon as the domains of X_2, X_3 do not contain the values false and red anymore, respectively, values 2 and 5 can be removed from D_1.

To accommodate this fact we only need to ensure is that the two variable equations that are watched cannot regard the same variable. This is automatically guaranteed in SAT when we remove tautologies upfront. Although as we have seen earlier; multi-valued clauses can have a number of variable equations in the same variable.

The two watched variable equations approach therefore watches two variable equations that regard two different variables. To this end, in each clause we group the variable equations according to their respective variables, and we fix an ordering of the variables and their allowed values in the clause. When a watched variable equality is affected, we search for a replacement starting at the old variable and its old value. If that

does not lead to a replacement support, we continue with the next variable, whereby we skip over the variable that is used for the second variable equality.

2.3 Quantitative Supports

In multi-valued SAT we can enhance this scheme by allowing a different kind of support that is based on the size of the current domain of a variable. For a given clause c, let us denote with i_c the number of variable equations that regard X_i in c. Furthermore, let us denote with d_i the size of the initial domain of X_i. The observation is that there must exist at least one satisfiable variable equality for a given variable as long as the size of the current domain is still bigger than the number of values that are disallowed by this clause when all variable equations regarding other variables in this clause cannot be satisfied. Formally: $|D_i| > d_i - i_c \quad \Rightarrow \quad \exists v \in D_i : \ X_i = v \in c$.

Example 2. Given are two variables X_1, X_2 with initial domains $D_1 = D_2 = \{1, \ldots, 6\}$, and a clause $c = (X_1 = 1 \vee X_1 = 2 \vee X_1 = 3 \vee X_1 = 4 \vee X_1 = 5 \vee X_2 = 1)$. The constraint just says that $X_2 = 1$ or $X_1 \neq 6$. Now assume that the current domain of X_1 is $D_1 = \{2, 3, 6\}$. The clause only disallows one value for X_1, but three values are still allowed: $|D_1| = 3 > 1 = 6 - 5 = d_1 - 1_c$. Consequently, there must still exist a variable equality on X_1 in c that can still be satisfied, in our case e.g. $X_1 = 2$.

The point of these quantitative supports is that we do not need to place a bet on any particular value for a given variable. Instead, a clause only needs to be looked at when the domain size of a watched variable falls below or equal the threshold $d_i - i_c$, no matter which particular values are lost until this happens. Especially when few values are disallowed for a variable, watching quantitative supports can save us a lot of work. Note that this type of support is not actually new in constraint programming. Solvers like IBM CP Solver have long associated the filtering of constraints with certain events, such as the event that a variable is bound (i.e. when its corresponding domain has shrunk to size 1). The only difference here is that we allow multi-valued clauses to set their own specific variable domain threshold which triggers when they need to be queued for filtering again.

3 Learning Nogoods for Multi-Valued SAT

We have seen in the previous section that multi-valued clauses offer the potential for filtering even before the number of satisfiable variable equations is 1, as long as the ones that remain satisfiable all constrain the same variable. Depending on the SAT encoding, a boolean SAT solver that learns redundant constraints can improve its filtering effectiveness to achieve the same, but the fact that this filtering is guaranteed is already an advantage of modeling a problem as multi-valued SAT over classic SAT.

A second and probably more important advantage of multi-valued SAT is that we can learn better implied constraints by exploiting our knowledge that each variable must take exactly one value. Before we explain how this can be achieved, let us begin by reviewing conflict driven clause learning in SAT.

3.1 Conflict Analysis

Unit propagation is an incomplete inference mechanism in so far as it does not guarantee that all variable domains have size 1 at the end and that it is not guaranteed that a

variable domain will be empty even when the given formula has no solution. The only facts that we know for sure are that variables that are unit must be set to the corresponding value in any solution and, consequently, that the formula has no solution if unit propagation finds a variable that can neither be set to true nor to false.

Due to this incompleteness of our filtering method we need to conduct a search for a solution. We assign variables one after the other to values in their domain, whereby after each assignment we perform unit propagation to simplify the formula or to detect that our current partial assignment cannot be extended to a solution. In the latter case, we speak of a *failure* and the variable that can neither be true nor false is called a *conflict variable*.

At this point, we could just undo the last variable assignment and try a different one. We can do much better, though. The power of modern SAT solvers lies in their ability to analyze the cause of the failure and to construct a redundant constraint, a so-called *nogood*, that will help prevent us from making the same error again. Crucial for this conflict analysis is our ability to give the exact reasons why a constraint filters a value. SAT clauses are very well suited for conflict analysis as we can trace exactly which prior domain reductions triggered the filtering.

In particular, modern systematic SAT solvers set up a so-called *implication graph*. It allows us to trace back the reasons why certain domain values have been removed. By collecting all causes for the removal of all domain values of the conflict variable, we can state a new constraint which forbids that a sufficient condition for a failure occurs. The easiest way to explain how the implication graph is set up and used is by means of an example which we will use throughout this section.

Example 3. Consider a constraint satisfaction problem (CSP) with five variables and five constraints. The variables have domains $D_1 = D_2 = D_4 = \{1, 2, 3\}, D_3 = \{1, \ldots, 6\}, D_5 = \{1, 2\}$. The constraints are $X_1 \neq X_4$, $X_2 \neq X_3$, $X_3 + 9 \geq 5X_4$, $X_3 + 4 \geq 5X_5$, and $X_4 \neq X_5 + 1$.

Say we model this problem by introducing boolean variables x_{ij} which are true iff $X_i = j$. To ensure that each CSP variable takes exactly one value we introduce clauses $(\vee_j x_{ij} = \text{true})$ for all i, and $(x_{ij} = \text{false} \vee x_{ik} = \text{false})$ for all i and $j < k$. In particular, we have a clause

$$(x_{31} = \text{false} \vee x_{36} = \text{false}). \tag{3}$$

We need to add multiple clauses for each CSP constraint. Among others, for constraints $X_1 \neq X_4$, $X_2 \neq X_3$, and $X_4 \neq X_5 + 1$ we add

$$(x_{11} = \text{false} \vee x_{41} = \text{false}) \wedge (x_{22} = \text{false} \vee x_{32} = \text{false}) \wedge (x_{42} = \text{false} \vee x_{51} = \text{false}). \tag{4}$$

Constraints $X_3 + 9 \geq 5X_4$ and $X_3 + 4 \geq 5X_5$ are enforced by the clauses

$$(x_{41} = \text{true} \vee x_{42} = \text{true} \vee x_{36} = \text{true}) \wedge (x_{51} = \text{true} \vee x_{36} = \text{true}). \tag{5}$$

The resulting SAT formula has no unit clauses, and so we begin our search. Assume that we first commit $x_{11} \leftarrow \text{true}$. To record this setting, we add a node $(x_{11} \neq \text{false})$ to our implication graph,[1] and we note that this node is added at search depth level 1 (compare with Figure 1 and ignore the dashed nodes and arcs). Among the clauses that

[1] Note that our notation is not standard in SAT. We use it here because it will naturally generalize to multi-valued SAT later.

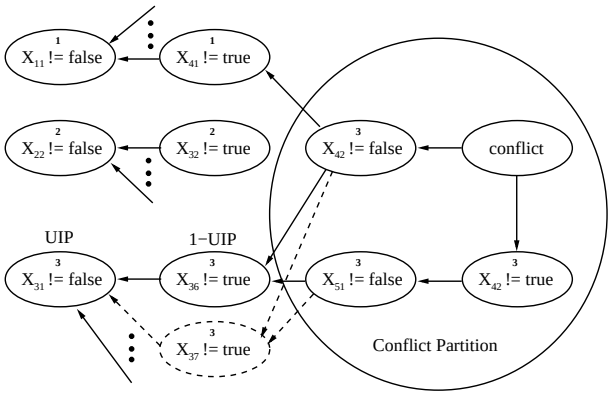

Fig. 1. SAT implication graph for Example 3. The solid nodes and arcs depict the relevant part of the implication graph when $D_3 = \{1, \ldots, 6\}$. The dashed nodes and arcs are part of the implication graph when $D_3 = \{1, \ldots, 7\}$.

we consider for this example, only the first clause in (4) becomes unit, and we infer ($x_{41} \neq$ true). We add also this node to the graph, mark it with depth level 1, and draw an arrow from the second to the first node to record that the first variable inequality implies the second.

After unit propagation is complete, we may next set $x_{22} \leftarrow$ true. Again, we add a node ($x_{22} \neq$ false) to our graph and note that it was added on depth level 2. By (4) we infer ($x_{32} \neq$ true), add this node with depth mark 2, and add an arc to ($x_{22} \neq$ false).

Again after all unit propagation is complete, let us assume that we now commit $x_{31} \leftarrow$ true. Again, we add ($x_{31} \neq$ false), and by (3) we infer ($x_{36} \neq$ true) (both at depth level 3, just like all nodes that follow). The first clause in (5) is now unit and we add a new node ($x_{42} \neq$ false). We add arcs to node ($x_{41} \neq$ true) and to node ($x_{36} \neq$ true), as both variable inequalities are needed to make this implication. The second clause in (5) is also unit, and we infer ($x_{51} \neq$ false) which is implied by ($x_{36} \neq$ true). The last clause in (4) is now unit and we add node ($x_{42} \neq$ true). Together with the earlier implied ($x_{42} \neq$ false) we have now reached a conflict. In Figure 1 we show the implication graph at this point (please ignore the dashed elements).

This graph is used to compute nogoods: Any set of nodes that represents a cut between the conflict node on one side and all branch nodes (i.e. nodes without outgoing arcs) on the other defines a valid constraint. For example, all paths from the conflict node to any branch node must visit either ($x_{42} \neq$ false) or ($x_{36} \neq$ true). Consequently, it is sound to enforce that not both variable inequalities hold at the same time or, equivalently, that $(x_{42} = \text{false} \vee x_{36} = \text{true})$. Conveniently, this constraint is again a clause! In terms of our CSP variables means that $X_4 = 2$ implies $X_3 = 6$.

Another cut set is the set of all reachable branch nodes, in our case $(x_{11} \neq \text{false})$ and $(x_{31} \neq \text{false})$. This cut results in the clause $(x_{11} = \text{false})$ and $(x_{31} = \text{false})$, or $X_1 = 1$ implies $X_3 \neq 1$. This latter fact is interesting as it obviously means that after our first search decision we could already have inferred $x_{31} \neq \text{true}$. Another way to put this is to say that the second branching decision was irrelevant for the conflict encountered, a fact that we can exploit to undo multiple search decisions, a process which is commonly referred to as *back-jumping* or *non-chronological backtracking*.

In order to achieve immediate additional propagation by the newly learned clause after back-jumping, we need to make sure that the clause contains only *one* node from the last depth level. Then and only then the newly learned clause will be unit and thus trigger more filtering after back-jumping. To find nodes that can be extended to a full cut using only nodes from lower depth levels, we consider only paths between the conflict node and the last branching decision. In Figure 1, this happens to be the subgraph induced by the nodes marked with depth level 3 (in general it would be a subset of the nodes on the lowest level). In this subgraph, we search for cut points, i.e., nodes that all paths from conflict to branch node must visit. In SAT these are commonly referred to as *unit implication points (UIP)*.

Note that these UIPs can be computed in time linear in the number of edges of the subgraph. In our case, there are two UIPs, ($x_{36} \neq$ true) and the branch node itself, ($x_{31} \neq$ false). In [25] it was established that it is beneficial to consider the UIP that is closest to the conflict. This is called the *first* UIP (1-UIP). Now, we group all nodes between the 1-UIP and the conflict together (marked by the circle in Figure 1). The cut set is then the set of all nodes that have a direct parent in this set. In our case, these are ($x_{36} \neq$ true) and ($x_{41} \neq$ true), which gives the nogood ($x_{36} =$ true $\vee x_{41} =$ true); or, in terms of the CSP variables, $X_4 \neq 1$ implies $X_3 = 6$.

3.2 Unit Implication Variables

As the example shows, conflict analysis can result in powerful inference. We will now show how we can learn even stronger nogoods in multi-valued SAT.

Consider again Example 3, but this time let us assume that $D_3 = \{1, \ldots, 7\}$. In our SAT model, the clauses enforcing $X_3 + 9 \geq 5X_4$ and $X_3 + 4 \geq 5X_5$ change to $(x_{41} = \text{true} \vee x_{42} = \text{true} \vee x_{36} = \text{true} \vee x_{37} = \text{true})$ and $(x_{51} = \text{true} \vee x_{36} = \text{true} \vee x_{37} = \text{true})$.

If we branch as before, the implication graph at depth level 3 includes the dashed arcs and nodes in Figure 1. We observe that it has only one UIP now, and that is the branch node itself. Consequently, the nogood learned is much weaker, we only infer ($x_{31} =$ false $\vee x_{41} =$ true); or, in terms of the CSP variables, that $X_4 \neq 1$ implies $X_3 \neq 1$.

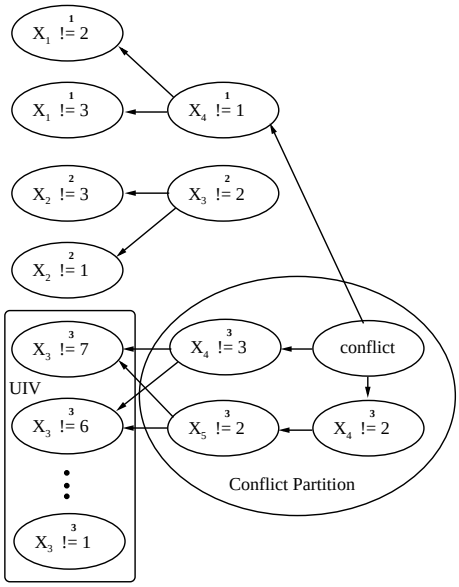

Fig. 2. MV-SAT Implication Graph.

In Figure 2 we show the implication graph for the corresponding multi-valued SAT model. Note that this graph no longer contains a single node that corresponds to the branching assignment. This is rather given as a number of variable inequalities. Observe further that in this graph there does not exist a UIP at all anymore. However, recall from Section 2 that a multi-valued clause can cause filtering even when multiple variable equations are still satisfiable, as long as all of them regard the same variable. For us, this means that we no longer need to find a cut point, but potentially an entire set of nodes:

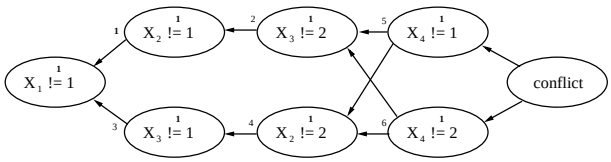

Fig. 3. Implication graph for a problem with four variables with domains $D_1 = D_4 = \{1, 2\}$ and $D_2 = D_3 = \{1, 2, 3\}$.

Definition 5 (Unit Implication Variable). *Given a multi-valued SAT problem and an implication graph G, assume that, if all nodes associated with variable X are removed from G, there exists no path from the conflict node to any branch node on the lowest branch level. Then, we call X a* unit implication variable (UIV).

In our example, X_3 is a UIV. Based on its associated cut set, we can again compute a conflict partition and set the nogood as the negation of the conjunction of all variable inequalities which have a direct parent in the conflict partition. In our example, we thus find the multi-valued clause $(X_4 = 1 \vee X_3 = 6 \vee X_3 = 7)$. After backjumping to the highest depth 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.

The challenge here is to find UIVs efficiently. Unfortunately this task no longer consists of the trivial linear computation of a cut point. We propose the following approach. First, we compute the shortest (in the number of nodes) path from any conflict variable node to any branch node. Only variables associated with nodes on this path can be UIVs. We call this set of variables the *candidate set*. Next we update the costs of the edges in the graph such that visiting a node associated with a variable in the candidate set incurs a cost of one, while all other nodes cost nothing. We compute another shortest path based on this cost function. Again, only variables associated with nodes on this path can be UIVs. We can therefore potentially reduce the candidate set. We repeat this process as long as the candidate set keeps shrinking. Finally, we test each remaining candidate by incurring a cost of one for nodes associated with the candidate variable only. If and only if the cost of the shortest path is greater than zero, then this implies that every path from a conflict node to a branch node must pass through a node associated with the candidate variable, which is therefore a UIV. It follows:

Lemma 1. *The set of all unit implication variables can be computed in time $O(mn)$, where m is the number of edges and n is the number of nodes in the subgraph of paths between a conflict and a branch nodes.*

Proof. Apart from the first and one other which establishes that the candidate set does not shrink anymore, each shortest path computation reduces the candidate set by at least one candidate. As there are at most n candidates, we require at most $n + 2$ shortest path computations. The implication graph is a directed acyclic graph, and therefore each shortest path computation takes time $\Theta(m)$. □

While it is reassuring that UIVs can be computed in polynomial time, a time bound of $O(mn)$ is certainly not very appealing seeing that we need to compute a nogood at

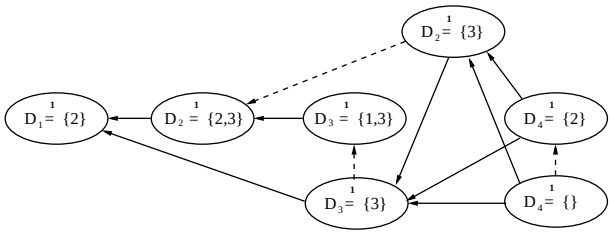

Fig. 4. CAMA implication graph for the same example as in Figure 3.

every failure. In practice we can of course hope that way fewer than n shortest path computations will be necessary. Fortunately, as our experiments will show, this hope is very well justified.

3.3 Non-Dominated UIVs

Recall that in SAT, if there are many UIPs, we choose the one that is closest to the conflict. In multi-valued SAT, we may also have multiple UIVs, whereby the last branching variable is always one of them. The question arises which UIV we should prefer. It is easy to see that there is no longer one unique UIV that dominates every other. For example, see the implication graph in Figure 3.

Definition 6 (Non-Dominated UIV). *We call a UIV* non-dominated *if there exists a path from the conflict to a branch node where a node associated with the UIV is the first node on the path that is associated with a UIV.*

In our algorithm, we can easily compute a non-dominated UIV by testing the remaining candidates in the order in which they appear on the first shortest path that we computed. Our solver learns the nogood that corresponds to this UIV.

An important aspect of our computing non-dominated UIVs is that they give us a good indication of the strength of the nogoods that we compute. In our experiments, we found that on problems where the vast majority of UIVs coincides with the branching variable, learning sophisticated nogoods is often a waste of time. Consequently, when our solver detects that the number of UIVs that are different from the branching variable drops below 5%, we no longer attempt to find improved, non-dominated UIVs and simply use the branching variable to define the next nogood instead.

3.4 Nogood Management

Among others, our solver offers impacts for selecting the branching variable [17]. Impacts measure the reduction in search space size achieved by propagation after each branching step. The concept has been found very effective for selecting branching variables.

As we learn more and more nogoods through the course of the search, an important task for any conflict driven solver is the management of learned constraints. First, due to limited memory it is simply not feasible to store all learned nogoods until an instance is solved. Second, for the efficiency of the solver it is essential that we forget redundant constraints which only cause work but rarely filter anything.

We use impacts to determine nogoods that can be deleted without losing much inference power. Whenever a constraint filters some values during the propagation of the

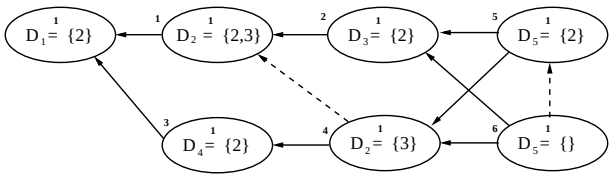

Fig. 5. CAMA implication graph for a problem with five variables with domains $D_1 = D_3 = D_4 = D_5 = \{1, 2\}$ and $D_2 = \{1, 2, 3\}$.

effects of a branching decision, we associate the constraint with the *entire* reduction in search space that all constraints achieve together. The rational behind looking at the entire reduction is that a constraint may not remove many values but very important ones which trigger lots of follow-up propagation.

We keep a running average of these reductions for each learned constraint. When the number of learned clauses reaches a limit (which grows over time), we remove roughly half of the learned clauses by removing all that have an average reduction below the median of all learned clauses. To ensure the completeness of the approach, clauses are protected from removal when they are currently unit.

This scheme works well in principle, but it has one major drawback: constraints which are part of some high impact propagation and which then never become unit again would clog up our system as their average reduction stays high. Therefore, in regular intervals we decrease the expected reduction for each learned constraint by a certain percentage. In our experiments, we decrease the expected reduction by 7% every 100 failures. In this way, constraints that have not been useful in a while get discounted.

4 Related Work

There are many approaches which reduce CSP to SAT and then employ a standard boolean SAT solver. A number of different encodings have been proposed for this purpose [24, 10, 23]. In the award-winning paper from Ohrimenko et al. [15], CSP propagators themselves were encoded lazily as SAT clauses which gave very good results on scheduling problems. The CSP solver Sugar [23], which won the ACP global constraints competition in the past two years, computes very efficient encodings for various global constraints and then employs MiniSAT [7] to solve the resulting SAT problem. Our work is heavily influenced by these studies and have motivated us to provide a back-end SAT solver that could directly exploit the fact that variables must take exactly one out of many values.

In [11, 14], classical CSP nogoods [19] were generalized to accommodate multi-valued variables better. Pioneering work on multi-valued SAT was presented in [2–4]. Here, the then state-of-the-art SAT solvers Chaff [18] and SATZ [12] were augmented with domain-based branching heuristics. Very good speed-ups over the performance over baseline SAT solvers were reported which were solely due to the ability of the multi-valued solver to take domain sizes into account when branching.

An incomplete multi-valued SAT solver was presented in [8]. It is based on an adaptation of the well known local search solver WalkSAT [20]. It was shown that working the knowledge that each variable must take exactly one out of many values into the solver can lead to superior performance on instances from various problem classes with larger variable domain sizes.

QWH	+Q	-Q	BCSP	+Q	-Q	GraphCol	+Q	-Q
qwh-25-40	3.91	5.44	b-25-20-.43	0.70	0.90	fpsol2.i.1	1.14	1.74
qwh-25-42	3.04	4.18	b-25-25-.45	0.92	1.29	inithx.i.1	0.87	1.29
qwh-27-40	3.96	5.90	b-25-30-.47	1.15	1.60	inithx.i.2	0.38	0.62
qwh-27-42	3.21	5.06	b-25-40-.5	1.45	1.93	le450_15a	0.54	0.67
qwh-29-40	4.56	6.67	b-30-20-.37	0.75	0.96	le450_15b	0.36	0.49
qwh-29-42	3.55	4.48	b-30-25-.39	0.94	1.25	le450_25a	0.24	0.40
qwh-31-40	4.19	5.69	b-30-30-.33	1.05	1.36	le450_25b	0.24	0.38
qwh-31-42	3.29	5.07	b-30-40-.35	1.66	2.19	le450_5a	0.45	0.72
qwh-33-40	4.31	6.39	b-35-20-.26	0.82	1.02	le450_5c	1.95	2.70
qwh-33-42	3.71	5.32	b-35-25-.28	1.17	1.46	miles1500	2.23	3.52
qwh-35-40	5.33	6.79	b-35-30-.29	1.48	2.00	queen8_8	0.52	0.53
qwh-35-42	4.21	5.44				queen9_9	1.03	1.03

Table 1. Time [ms] per choice point when using (+Q) and not using (-Q) quantitative supports.

The only "pure" complete multi-valued SAT solver we know of was presented in [13]. It was named CAMA and like our own solver it features propagation based on watched literals (albeit without quantitative supports) and a nogood learning method which exploits the knowledge about multi-valued variables. Like us the authors attempt to learn improved nogoods.

In CAMA, a nogood is not constructed through the analysis of an implication graph but through resolution. An implication graph is used only for the computation of a UIV. The implication graph differs considerably from ours, though. As we will see, due to its structure, CAMA is not able to identify non-dominated UIVs, for two reasons:

First, CAMA does not consider pure variable inequalities for learning nogoods, but the entire domain of each variable after a value has been removed. The current domain, however, reflects *all* domain reductions on the variable and not just the ones that are relevant for the filtering that is triggered. Consequently, CAMA needs to trace back the relevant domain reductions, and since it conservatively assumes that the relevant domain reductions happened earlier during propagation, it may miss a unit implication point. In Figure 4 we show a CAMA implication graph for the same example as depicted in Figure 3 where we also mark the order in which the nodes are added to the graph. CAMA computes, in linear time, the cut point in this graph that is closest to the conflict node. As we can see, due to the dashed edges which are needed to denote the implications by earlier domain reductions on the same variable, the only cut point is the branching node itself. CAMA thus finds variable X_1 as UIV which is dominated both by X_2 and X_3 as we can easily see in Figure 3.

The second reason why CAMA cannot identify non-dominated UIVs is that it is simply unable to identify all UIVs by only considering cut points in its implication graph. In Figure 5 we show a different example. Here, even when we ignore the dashed arcs, the only cut point is the branch node, and CAMA chooses X_1 as UIV which is dominated by X_2. In summary, CAMA's nogood learning method runs in linear time, but therefore the nogoods found are in general not as strong as they could be.

5 Numerical Results

We have introduced quantitative supports and non-dominated UIVs for nogood learning in multi-valued SAT. We will now study these contributions and finally compare our solver with standard SAT technology.

	ND-UIV			Q-UIV			ND-UIV			Q-UIV	
QWH	Paths	Time	Fails	Time	Fails	GraphCol	Paths	Time	Fails	Time	Fails
qwh-25-40	2.37	0.61	90.7	0.68	94.4	fpsol2.i.1	1.09	0.38	35.6	0.39	35.6
qwh-25-42	2.34	0.32	33.7	0.43	44.7	inithx.i.1	1.01	0.49	24.4	0.47	24.4
qwh-27-40	2.43	0.72	85.3	1.10	130	inithx.i.2	0	0.20	0	0.20	0
qwh-27-42	2.39	0.58	62	1.02	117	le450_15a	2.16	3.69	2.93K	41.7	23.2K
qwh-29-40	2.44	2.83	331	3.87	442	le450_15b	2.08	1.64	1.27K	5.27	4.2K
qwh-29-42	2.47	3.14	360	3.06	329	le450_25a	0	0.11	0	0.11	0
qwh-31-40	2.43	2.12	210	2.70	228	le450_25b	0	0.10	0	0.11	0
qwh-31-42	2.40	1.14	97.9	1.62	141	le450_5a	2.04	3.06	4.42K	10.4	13.5K
qwh-33-40	2.44	4.39	394	7.23	658	le450_5c	1.90	0.12	51	0.13	65
qwh-33-42	2.41	2.43	200	4.12	351	miles1500	1.04	0.61	73.8	0.64	73.8
qwh-35-40	2.44	19.2	1.51K	33.9	2.14K	queen8_8	1.90	18.1	24.1K	19.8	27.7K
qwh-35-42	2.44	6.18	482	17.2	1.19K	queen9_9	1.93	108	79.8K	173	96.5K

Table 2. Comparison between non-dominated (ND-UIV) and quick UIVs (Q-UIV). Time in [s].

5.1 Benchmark Sets and Architecture

For our experiments, we use the following four classes of problems: quasi-group with holes, random binary constraint satisfaction problems, n-queens, and graph coloring.

The quasi-group with holes instances were produced by the generator of Carla Gomes. Instances of different sizes and different percentages of holes were used (40% and 42% which is right below and right at the phase transition). Ten instances were generated for each parameter setting of the generator and collected in a set named qwh-[order]-[percent holes].

Random binary constraint satisfaction problems were generated by the generator of Christian Bessiere available at [5]. Instances vary in number of variables, domain size, number of constraints, and constraint tightness. We fix the density of the constraint graph at 0.5 and then derive the value for the critical constraint tightness using the formula given in [16] which is the value for which the BCSP problems are generally hard. We then generate instances with constraint tightness slightly above and below the critical value. Ten instances were generated for each parameter setting of the generator and collected in a set named b-[vars]-[vals]-[tightness].

The n-queens model consists of the standard four types of all different constraints; two enforcing that queens cannot attack each other on the rows and columns, and two enforcing that the queens cannot attack each other on diagonals. In CMV-SAT-1 the all different constraint is decomposed into a clique of not equal constraints. Not equal constraints are transformed into disjunction of variable assignments. The SAT encoding of all different constraints provided by Sugar is described in [22].

The graph coloring instances are part of the DIMACS standard [9]. The problems were changed into decision problems as opposed to optimization problems by setting the desired number of colors to the best known value. We use the subset of 44 instances which could be solved in under one hour of CPU time.

For each instance, we report the average statistics (runtime, nodes, failures, etc.). For all experiments including the ones on different configurations of CMV-SAT-1 we used ten different seeds per instance and ran on Intel Core 2 Quad Q6600 processors with 3GB of RAM.

	Imp		MinDom			Imp		MinDom			Imp		MinDom	
QWH	Time	Nodes	Time	Nodes	BCSP	Time	Nodes	Time	Nodes	GraphCol	Time	Nodes	Time	Nodes
qwh-25-40	0.61	166	0.81	193	b-25-20-.43	2.04	2.91K	2.51	2.90K	fpsol2.i.1	0.38	362	0.28	375
qwh-25-42	0.32	107	0.45	132	b-25-25-.45	6.35	7.02K	7.66	6.64K	inithx.i.1	0.49	587	0.96	777
qwh-27-40	0.72	187	0.86	229	b-25-30-.47	13.9	12.6K	16.9	11.7K	inithx.i.2	0.20	558	0.22	599
qwh-27-42	0.58	179	0.83	227	b-25-40-.5	54.6	40.4K	69.2	35.5K	le450_15a	3.69	6.84K	23.2	17.6K
qwh-29-40	2.83	585	4.54	742	b-30-20-.37	18.0	24.7K	19.5	20.6K	le450_15b	1.64	3.24K	1.23	4.52K
qwh-29-42	3.14	645	2.02	551	b-30-25-.39	55.4	60.7K	62.5	51.7K	le450_25a	0.11	438	0.10	438
qwh-31-40	2.12	497	2.44	576	b-30-30-.33	1.21	1.10K	1.69	1.33K	le450_25b	0.10	438	0.09	438
qwh-31-42	1.14	350	1.38	389	b-30-40-.35	2.53	1.46K	4.89	2.09K	le450_5a	3.06	5.67K	1.62	7.86K
qwh-33-40	4.39	861	5.22	1.07K	b-35-20-.26	1.40	1.55K	1.70	1.63K	le450_5c	0.12	73.5	0.10	69.1
qwh-33-42	2.43	641	2.75	769	b-35-25-.28	5.08	4.19K	5.13	3.63K	miles1500	0.61	257	0.48	449
qwh-35-40	19.2	2.68K	12.8	2.79K	b-35-30-.29	4.85	3.19K	9.91	4.54K	queen8_8	18.1	27.8K	24.0	25.5K
qwh-35-42	6.18	1.28K	6.43	1.67K						queen9_9	108	92.1K	135	72.5K

Table 3. Comparison between minDomain and impact-based branching. Time in [s].

5.2 Quantitative Supports

In Table 1 we give the average time per choice point when using and when not using quantitative supports. We see clearly that quantitative supports speed up the propagation process considerably and almost independently of the type of problem that is solved. The reduction in time per choice point is roughly 20%-25% on average. Given that propagation does not make up for 100% of the work that has to be done per choice point (there are also impact updates, nogood computations, branching variable selection etc.), this reduction is substantial.

5.3 Non-Dominated UIVs

Next we investigate the impact of computing non-dominated UIVs when learning nogoods. Table 2 shows the results on quasi group with holes and graph coloring instances. We did not conduct this experiment on random binary CSPs as our solver detects quickly that most often the branching variable is the only UIV and then switches the optimization off.

In the table we compare two variants of our solver. The first uses the $O(mn)$ approach presented in Section 3.2 and ensures that non-dominated UIVs are used for computing the nogood. The second approach works in linear time $O(m)$ and uses the branching variable as a basis for computing the nogood.

We see clearly that using non-dominated UIVs has a profound impact on the number of failures which are almost always substantially lower than when potentially dominated nogoods are used. Interestingly, our data shows that the time per choice point is not measurably higher when using the advanced nogood learning scheme. In Table 2 we show the average number of shortest path computations for each failure. As we can see, it is usually very low, somewhere between two and three shortest paths are sufficient on average to find a non-dominated UIV.

5.4 MinDomain vs. Impacts

The work in [2] suggested that augmenting a SAT solver with min domain branching can lead to substantial performance improvements. Since impacts have since become a popular alternative to min domain branching in CP, we investigated which method performs better for multi-valued SAT. As Table 3 shows, on graph coloring both methods perform roughly the same, while on random binary CSPs and QWH impacts work clearly better. We also tested activity-based branching heuristics commonly used in SAT and min domain over weighted degree [6], but both were not competitive (the latter due to the large number of constraints). Our solver therefore uses impact-based branching.

Class	CMV-SAT-1 Time	CMV-SAT-1 Nodes	CMV-SAT-1 Time Outs	MiniSAT Time	MiniSAT Nodes	MiniSAT Time Outs
GraphCol	137	138K	0	178	373K	3
N-Queens	168	12.4K	2	235	106K	0
qwh D=25	0.93	273	0	5.35	19.6K	0
qwh D=27	1.30	366	0	13.2	36.7K	0
qwh D=29	6.00	1.23K	0	48.0	94.4K	0
qwh D=31	3.26	847	0	64.0	119K	1
qwh D=33	6.82	1.5K	0	178	218K	14
qwh D=35	25.2	4K	0	338	318K	54
QWH Total	43.5	8.26K	0	647	806K	69
b D=20	21.4	30K	0	13.8	181K	0
b D=25	66.8	72K	0	64.2	524K	1
b D=30	20.0	17K	0	22.1	165K	0
b D=40	61.1	41K	0	103	423K	0
BCSP Total	169.3	160K	0	203.1	1293K	1

Table 4. CMV-SAT 1 vs. MiniSAT. For the QWH and BCSPs we aggregate all instances in our benchmark set that have the same domain size, for Graph Coloring and N-Queens we aggregate all instances. Time in [s], timeout for the runs is 15 min.

5.5 MV-SAT vs. SAT

In our last experiment, we compare our multi-valued solver with the well-known MiniSAT solver for boolean SAT. In particular, we use Sugar [23] to pre-compile the SAT formulas from the XCSP model of each instance. The MV-SAT instances for our solver are generated as explained before. In our experiment, we compare the pure *solution time* of MiniSAT and CMV-SAT-1 on the resulting SAT and MV-SAT instances. Note that this solution time does not include the time that Sugar needs to compile the SAT formula, nor the time for reading in the input.

Table 4 summarizes our results. We observe that CMV-SAT-1 visits massively fewer choice points than MiniSAT. Depending on the class of inputs the reduction is typically between one and two orders of magnitude. We attribute this reduction in part to impact-based branching, and in part to the use of sophisticated nogoods.

We also observe that our prototype requires about an order of magnitude more time per choice point than MiniSAT. Only to a small extend this is due to the additional time needed for learning high-quality nogoods. When solving random BCSPs, CMV-SAT-1 quickly finds that computing sophisticated nogoods is not worthwhile and then uses the branching variable as UIV as explained in Section 3.3. Still we need about an order of magnitude more time per choice point on these instances. This indicates that our implementation still leaves a lot of room for improvement.

Overall, we find that CMV-SAT-1 performs a little better than MiniSAT on graph coloring and random BCS problems, whereby on both CMV-SAT-1 works more robustly and thus causes fewer timeouts. On n-queens and quasi groups with holes, CMV-SAT-1 clearly outperforms MiniSAT. For all problems, the reduction in the number of choice points is very substantial and overall the multi-valued SAT solver runs upto fifteen times faster than the boolean SAT solver.

6 Conclusion

We have introduced CMV-SAT-1, a new complete multi-valued SAT solver which can serve as a back-end for CSP solvers that are based on decomposition and reformulation as SAT. We contributed the ideas of quantitative supports to augment the well-known watched literal scheme, and a new method for learning multi-valued nogoods. Experiments substantiated the practical benefits of these ideas and showed that multi-valued SAT solving offers great potential for improving classical boolean SAT technology.

References

1. C. Ansótegui. Complete SAT solvers for Many-Valued CNF Formulas. *PhD thesis*, Universitat de Lleida, 2004.
2. C. Ansótegui, J. Larrubia, F. Manyà. Boosting Chaff's Performance by Incorporating CSP Heuristics. *CP*, 96–107, 2003.
3. C. Ansótegui and F. Manyà. Mapping Problems with Finite-Domain Variables to Problems with Boolean Variables. *SAT*, 1–15, 2004.
4. C. Ansótegui, J. Larrubia, C. Liu, F. Manyà. Exploiting multivalued knowledge in variable selection heuristics for SAT solvers. *Ann. Math. Artif. Intell.*, 49(1-4): 191–205, 2007.
5. C. Bessiere. : http://www.lirmm.fr/ bessiere/generator.html.
6. F. Boussemart, F. Lecoutre, C. Sais. Boosting systematic search by weighting constraints. *ECAI*, 146–150, 2004.
7. N. Eén and N. Sörensson. An Extensible SAT-solver. *SAT*, 502–518, 2003.
8. A. Frisch and T. Peugniez. Solving Non-Boolean Satisfiability Problems with Stochastic Local Search. *IJCAI*, 282–288, 2001.
9. Graph coloring instances. http://mat.gsia.cmu.edu/COLOR/instances.html.
10. I. Gent. Arc consistency in SAT. *ECAI*, 121–125, 2002.
11. G. Katsirelos. Nogood Processing in CSPs. *PhD Thesis*, University of Toronto, 2009.
12. C. Li and A. Anbulagan. Heuristics based on unit propagation for satisfiability problems. *IJCAI*, 366–371, 1997.
13. C. Liu, A. Kuehlmann, M. Moskewicz. CAMA: A Multi-Valued Satisfiability Solver. *ICCAD*, 326–333, 2003.
14. D. Mitchell Resolution and Constraint Satisfaction. *CP*, 554–569, 2003.
15. O. Ohrimenko, P. Stuckey, M. Codish. Propagation=Lazy Clause Generation. *CP*, 544–558, 2007.
16. Patrick Prosser. An empirical study of phase transitions in binary constraint satisfaction problems . *Frontiers in Problem Solving: Phase Transitions and Complexity* , 81(1–2):81–109, 1996.
17. P. Refalo. Impact-Based Search Strategies for Constraint Programming. *CP*, 557–571, 2004.
18. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik. Chaff:Engineering an Efficient SAT Solver. *DAC*, 530–535, 2001.
19. T. Schiex and G. Verfaille. Nogood Recording for Static and Dynamic Constraint Satisfaction Problems. *IJAIT*, 48–55, 1994.
20. B. Selman, H. Kautz, B. Cohen. Local Search Strategies for Satisfiability Testing. *DIMACS*, 521–532, 1995.
21. International CSP Competition Result Pages. http://bach.istc.kobe-u.ac.jp/sugar/cpai08.html http://bach.istc.kobe-u.ac.jp/sugar/csc09.html
22. N. Tamura, A. Taga, M. Banbara. System Description of a SAT-based CSP solver Sugar. http://bach.istc.kobe-u.ac.jp/sugar/cpai08-sugar.pdf *CPAI*, 2008.
23. N. Tamura, A. Taga, S. Kitagawa, M. Banbara. Compiling Finite Linear CSP into SAT. *Constraints*, 14:254–272, 2009.
24. T. Walsh. SAT vs CSP. *CP*, 441–456, 2000.
25. L. Zhang, C. Madigan, M. Moskewicz, S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. *ICCAD*, 279–285, 2001.