New Lower Bounds of Constraint Violations for Over-Constrained Problems

Jean-Charles Régin[1], Thierry Petit[1/2], Christian Bessière[2]
and Jean-François Puget[3]
{puget, regin, tpetit}@ilog.fr, {bessiere, tpetit}@lirmm.fr

[1]ILOG, 1681, route des Dolines, 06560 Valbonne, FRANCE
[2]LIRMM (UMR 5506 CNRS), 161, rue Ada, 34392 Montpellier Cedex 5, FRANCE
[3]ILOG, 9, rue de Verdun, BP 85, 94253 Gentilly Cedex, FRANCE

Abstract. In recent years, many works have been carried out to solve over-constrained problems, and more specifically the Maximal Constraint Satisfaction Problem (Max-CSP), where the goal is to minimize the number of constraint violations. Some lower bounds on this number of violations have been proposed in the literature.
In this paper, we characterize the constraints that are ignored by the existing results, we propose new lower bounds which takes into account some of these ignored constraints and we show how these new bounds can be integrated into existing ones in order to improve the previous results.
Our work also generalize the previous studies by dealing with any kind of constraints, as non binary constraints, or constraints with specific filtering algorithms. Furthermore, in order to integrate these algorithms into any constraint solver, we suggest to represent a Max-CSP as a single global constraint. This constraint can be itself included into any set of constraint. In this way, an over-constrained part of a problem can be isolated from constraints that must be necessarily satisfied.

1 Introduction

A constraint network (CN) consists of a set of variables, each of them associated with a domain of possible values, and a set of constraints linking the variables and defining the set of allowed combinations of values. The search for an assignment of values to all variables that satisfies all the constraints is called the Constraint Satisfaction Problem (CSP). Such an assignment is a solution of the CSP.

Unfortunately, the CSP is an NP-Hard problem. Thus, many works have been carried out in order to try to reduce the time needed to solve a CSP. Some of the suggested methods turn the original CSP into a new one, which has the same set of solutions, but which is easier to solve. The modifications are done through filtering algorithms, that remove from domains values which cannot belong to any solution of the current CSP. If the cost of such an algorithm is less than the time required by the backtrack algorithm to discover many times the same inconsistency, then the solving will be accelerated.

It often happens that a CSP has no solution. In this case we say that the problem is over-constrained, and often the goal is then to find a good compromise. One of the most usual theoretical frameworks is called the Maximal Constraint Satisfaction Problem (Max-CSP). A solution of a Max-CSP is a total assignment that minimizes the number of constraint violations.

Most of existing algorithms for solving Max-CSPs are related to binary constraints and based on a branch and bound schema [2, 8, 4]. They perform successive assignments of values to variables through a depth-first traversal of the search tree, where internal nodes represent incomplete assignments and leaf nodes stand for complete ones. For any given node, the variables which have been instantiated are called past variables, whereas the other variables are called future variables (F). The *distance* of a node is the number of constraints violated by its assignment, UB is the distance of the best solution found so far, and LB is an underestimation of any leaf node descendant from the current one. When $LB \geq UB$, the current best solution cannot be improved below the current node. Thus it is not necessary to traverse the subtree rooted by the current node.

When filtering, the existing approaches combine generally LB with lower bounds local to each value, in order to remove values that cannot belong to a solution. These lower bounds are based on *direct violations of constraints by values.* A value a of a variable x directly violates a constraint C if C has no solution when $x = a$. In other words, (x, a) directly violates C if (x, a) is not consistent with C.

In the PFC-MRDAC algorithm [4], which can be considered as the best reference in the literature[1], two local lower bounds of violations are defined for every a in $D(x)$: $ic(x, a)$ which is related to constraints such that the other involved variable is a past variable, and $dac(x, a)$ which is related to constraints involving only future variables.

More precisely, $ic(x, a)$ is simply defined as the number of constraints involving x and a past variable that are directly violated if $x = a$. The definition of $dac(x, a)$ assumes that the constraint graph[2] is oriented. $dac(x, a)$ is equal to the number of constraints out-going x and involving only future variables that are violated if $x = a$. With these definitions LB is defined by:

(1) $LB = distance + \sum_{x \in F} inc(x)$,
where $inc(x) = min_{a \in D(x)}(ic(x, a) + dac(x, a))$.

When filtering future domains, PFC-MRDAC selects for each value the sum $ic(x, a) + dac(x, a)$. This sum is associated with LB, by removing $inc(x)$ from LB in order to guarantee that no violation of any constraint involving x is counted twice. Thus, a value a can be removed from $D(x)$ if:

(2) $ic(x, a) + dac(x, a) + LB - inc(x) \geq UB$.

[1] A variation of this algorithm has been suggested [5], based on a partitioning of the variable set.

[2] The vertex set of the constraint graph is the variable set and there is an edge between two vertices when there is a constraint involving these two variables.

Hence, the quality of the filtering algorithms depends on the value of *inc* counters, which depends on the value of *dac* counters.

However, some constraints which lead to inconsistencies are not taken into account in PFC-MRDAC. In particular, Equations (1) and (2) do not take into account inconsistencies involving constraints defined on variables such that the *inc* counter of each of them is equal to 0.

This drawback is quite important because the probability for having such constraints is huge in real-world applications especially when only few variables have been instantiated. It can be emphasized by the following example (Figure 1):

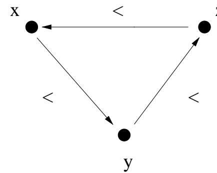

Figure 1

The problem involves three variables x, y, z with domains equal to $\{1, 2, 3\}$ and three constraints: $x < y$, $y < z$, and $z < x$. Value 2 of each variable does not directly violate any constraint. Thus, $inc(x) = inc(y) = inc(z) = 0$ whereas it is clear that any assignment of x, y, and z will lead to the violation of at least one constraint.

In this paper, we propose an original method for identifying constraints that are implicitly ignored by Equations (1) and (2). Then, we present a new way for computing a lower bound of the number of violations related to some of these constraints.

This lower bound can be computed by searching for disjoint conflicting sets of constraints. A *conflict set* is a set of constraints that cannot be simultaneously all satisfied. For instance, $\{x < y, y < z, z < x\}$ is a conflict set.

When a conflict set has been identified then in any solution at least one constraint of this conflict set will be violated. Thus, by finding disjoint conflict sets a non trivial lower bound of the number of violations holds. And this bound can be integrated to Equations (1) and (2).

The paper is organized as follows: first, we recall some notions about CNs. Then, we propose a new framework based on the representation of a Max-CSP as only one constraint called the Satisfiability Sum Constraint (ssc). Section 4 presents a generalization of the results proposed for binary Max-CSPs to the non binary case. Section 5 presents an original approach for computing a lower bound of the number of violations. This result is exploited in section 6 and leads to properties that improve the results of the previous studies. At last we recapitulate our results and conclude.

2 Background

A *constraint network* $\mathcal{N}$ is defined as a set of n *variables* $X = \{x_1, \ldots, x_n\}$, a set of *domains* $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ where $D(x_i)$ is the finite set of possible *values* for variable x_i, and a set $\mathcal{C}$ of *constraints* between variables. A *constraint* C on the ordered set of variables $X(C) = (x_{i_1}, \ldots, x_{i_r})$ is a subset $T(C)$ of the Cartesian product $D(x_{i_1}) \times \cdots \times D(x_{i_r})$ that specifies the *allowed* combinations of values for the variables $x_{i_1}, \ldots, x_{i_r}$. An element of $D(x_{i_1}) \times \cdots \times D(x_{i_r})$ is called a *tuple on* $X(C)$. $|X(C)|$ is the *arity* of C. A value a for a variable x is often denoted by (x, a). A tuple τ on $X(C)$ is *valid* if $\forall (x, a) \in \tau, a \in D(x)$. C is *consistent* iff there exists a tuple τ of $T(C)$ which is valid. A value $a \in D(x)$ is *consistent with* C iff $x \notin X(C)$ or there exists a valid tuple τ of $T(C)$ in which a is the value assigned to x. An Arc Consistency algorithm is an algorithm which guarantees that $\forall x \in X$, $\forall a \in D(x)$, $\forall C \in \mathcal{C}$, a is consistent with C. Given $\mathcal{K} \subseteq \mathcal{C}$, the subset of variables involved in constraints $\mathcal{K}$ is denoted by $X(\mathcal{K})$.

Some important results presented in the paper are based on the following definition:

Definition 1 *Let x be a variable, a be a value of $D(x)$, $\mathcal{C}$ be a set of constraints, $\#inc((x,a), \mathcal{C}) = |\{C \in \mathcal{C}$ s.t. (x,a) is not consistent with $C\}|$.*

3 Satisfiability Sum Constraint

Let $\mathcal{N} = (X, \mathcal{D}, \mathcal{C})$ be a constraint network. We suggest to integrate $\mathcal{C}$ into a single constraint, called the Satisfiability Sum Constraint (ssc):

Definition 2 *Let $\mathcal{C} = \{C_i, i \in \{1, \ldots, m\}\}$ be a set of constraints, and $S[\mathcal{C}] = \{s_i, i \in \{1, \ldots, m\}\}$ be a set of variables and unsat be a variable, such that a one-to-one mapping is defined between $\mathcal{C}$ and $S[\mathcal{C}]$. A* **Satisfiability Sum Constraint** *is the constraint $ssc(\mathcal{C}, S[\mathcal{C}], unsat)$ defined by:*

$$[unsat = \sum_{s_i=1}^{m} s_i] \wedge \bigwedge_{i=1}^{m} [(C_i \wedge (s_i = 0)) \vee (\neg C_i \wedge (s_i = 1))]$$

Notation 1 *Given a $ssc(\mathcal{C}, S[\mathcal{C}], unsat)$, a variable x, a value $a \in D(x)$ and $\mathcal{K} \subseteq \mathcal{C}$:*

- *$max(D(unsat))$ is the highest value of current domain of unsat;*
- *$min(D(unsat))$ is the lowest value of current domain of unsat;*
- *$minUnsat(\mathcal{C}, S[\mathcal{C}])$ is the minimum value of unsat consistent with $ssc(\mathcal{C}, S[\mathcal{C}], unsat)$;*
- *$minUnsat((x,a), \mathcal{C}, S[\mathcal{C}])$ is equal to $minUnsat(\mathcal{C}, S[\mathcal{C}])$ when $x = a$;*
- *$S[\mathcal{K}]$ is the subset of $S[\mathcal{C}]$ equals to the projection of variables $S[\mathcal{C}]$ on $\mathcal{K}$;*
- *$X(\mathcal{C})$ is the union of $X(C_i), C_i \in \mathcal{C}$.*

The variables $S[\mathcal{C}]$ are used in order to express which constraints of $\mathcal{C}$ must be violated or satisfied: value 0 assigned to $s \in S[\mathcal{C}]$ expresses that its corresponding constraint C is satisfied, whereas 1 expresses that C is violated[3]. Variable *unsat*

[3] An extension of the model can be performed [6], in order to deal with Valued CSPs [1]. Basically it consists of defining larger domains for variables in $S[\mathcal{C}]$.

represents the objective, that is, the number of violations in $\mathcal{C}$, equal to the number of variables of $S[\mathcal{C}]$ whose value is 1.

Throughout this formulation, a solution of a Max-CSP is an assignment that satisfies the *ssc* with the minimal possible value of *unsat*. A lower bound of the objective of a Max-CSP corresponds to a necessary consistency condition of the ssc. The different domain reduction algorithms established for Max-CSP correspond to specific filtering algorithms associated with the ssc.

This point of view has some advantages in regards to the previous studies:
1. Any search algorithm can be used. Since we propose to define a constraint we can easily integrate our framework into existing solvers. This constraint can be associated with other ones, in order to separate soft constraints from hard ones.
2. No hypothesis is made on the arity of constraints $\mathcal{C}$.
3. If a value is assigned to $s_i \in S[\mathcal{C}]$, then a filtering algorithm associated with $C_i \in \mathcal{C}$ (resp. $\neg C_i$) can be used in a way similar to classical CSPs.

Moreover, properties are simplified: there is no longer references about past or future variables; $min(D(unsat))$ and $max(D(unsat))$ respectively correspond to the parameters *distance* and $UB - 1$ of PFC-MRDAC.

4 Related Work

The results presented in this section are a generalization to non binary constraints of the previous works for Max-CSP [2, 9, 4].

4.1 Simple Filtering Algorithm

Domains of variables of $S[\mathcal{C}]$ initially contain two values. Removing one of them amounts to saying that the other one is assigned to the variable. Let $s \in S[\mathcal{C}]$ and $C \in \mathcal{C}$, such that s is linked to C in a *ssc*:

Property 1 *If the value* 0 *(resp.* 1*) is assigned to s then values from domains of variables $X(C)$ which are not consistent with C (resp. $\neg C$) can be removed.*

Property 2 *Let $x_i \in X(C)$. If all values of $D(x_i)$ are not consistent with C (resp. $\neg C$) then $s = 1$ (resp.* 0*).*

4.2 Necessary Condition of Consistency

From the definition of $minUnsat(\mathcal{C}, S[\mathcal{C}])$ we have:

Property 3 *If $minUnsat(\mathcal{C}, S[\mathcal{C}]) > max(D(unsat))$ then $ssc(\mathcal{C}, S[\mathcal{C}], unsat)$ is not consistent.*

A lower bound of $minUnsat(\mathcal{C}, S[\mathcal{C}])$ provides a necessary condition of consistency of a ssc. A possible way for computing it is to perform a sum of independent lower bounds of violations, one per variable. For each variable a lower bound can be defined by:

Definition 3 *Given a variable x and a constraint set $\mathcal{K}$,*
$\#inc(x, \mathcal{K}) = min_{a \in D(x)}(\#inc((x,a), \mathcal{K}))$*.*

The sum of these minima with $\mathcal{K} = \mathcal{C}$ cannot lead to a lower bound of the total number of violations, because some constraints can be taken into account more than once. For instance, given a constraint C and two variables x and y involved in C, C can be counted in $\#inc(x, \mathcal{C})$ and also in $\#inc(y, \mathcal{C})$. In this case, the lower bound can be overestimated, and an inconsistency could be detected while the *ssc* is consistent. Consequently, for each variable, an independent set of constraints must be considered.

In the binary case, the constraint graph has been used in order to guarantee this independence [4]. Each edge is oriented and for each variable x only the constraints out-going x are taken into account.

This idea can be generalized to the non binary case, by associating with each constraint C one and only one variable x involved in the constraint: C is then taken into account only for computing the $\#inc$ counter of x. Therefore, the constraints are *partionned* w.r.t the variables that are associated with:

Definition 4 *Given a set of constraints $\mathcal{C}$, a* **var-partition** *of $\mathcal{C}$ is a partition $\mathcal{P}(\mathcal{C}) = \{P(x_1), ..., P(x_k)\}$ of $\mathcal{C}$ in $|X(\mathcal{C})|$ sets such that $\forall P(x_i) \in \mathcal{P}(\mathcal{C}) : \forall C \in P(x_i), x_i \in X(C)$.*

Given a var partition $\mathcal{P}(\mathcal{C})$, the sum of all $\#inc(x_i, P(x_i))$ is a lower bound of the total number of violations, because all sets belonging to $\mathcal{P}(\mathcal{C})$ are disjoint; thus we have:

Definition 5 $\forall \mathcal{P}(\mathcal{C}) = \{P(x_1), ..., P(x_k)\}$,
$LB(\mathcal{P}(\mathcal{C})) = \sum_{x_i \in X(\mathcal{C})} \#inc(x_i, P(x_i))$*.*

Property 4 $\forall \mathcal{P}(\mathcal{C}) = \{P(x_1), ..., P(x_k)\}$,
$LB(\mathcal{P}(\mathcal{C})) \leq minUnsat(\mathcal{C}, S[\mathcal{C}])$*.*

The necessary condition of consistency of a *ssc* is deduced from this property:

Corollary 1 $\forall \mathcal{P}(\mathcal{C}) = \{P(x_1), ..., P(x_k)\}$*, If* $LB(\mathcal{P}(\mathcal{C})) > max(D(unsat))$ *then* $ssc(\mathcal{C}, S[\mathcal{C}], unsat)$ *is not consistent.*

The quality of such a lower bound depends on the var-partition that is choosen. This property corresponds to Equation (1) given in Introduction.

4.3 Filtering Algorithm

From definition of $minUnsat((x,a), \mathcal{C}, S[\mathcal{C}])$ we have the following theorem:

Theorem 1 $\forall x \in X(\mathcal{C}), \forall a \in D(x)$*:*
if $minUnsat((x,a), \mathcal{C}, S[\mathcal{C}]) > max(D(unsat))$ *then* (x,a) *is not consistent with* $ssc(\mathcal{C}, S[\mathcal{C}], unsat)$*.*

Therefore, any lower bound of $minUnsat((x,a), \mathcal{C}, S[\mathcal{C}])$ can be used to check the consistency of (x,a). An obvious lower bound is $\#inc((x,a), \mathcal{C})$:

Property 5 $\#inc((x,a),\mathcal{C}) \leq minUnsat((x,a),\mathcal{C},S[\mathcal{C}])$

From this property and theorem 1, we obtain a first filtering algorithm. This filtering algorithm can be achieved as a generalization of the constructive disjunction [7]: given $C_1 \vee C_2 ... \vee C_n$, the constructive disjunction removes a value from a domain when this value is not consistent with each constraint taken separately. The constructive disjunction corresponds to the particular case where $max(D(unsat)) = 1$.

This filtering algorithm can be improved, by including the lower bound of Property 4. In order to do so, we suggest to split $\mathcal{C}$ into two disjoint sets $P(x)$ and $\mathcal{C} - P(x)$, where $P(x)$ is the subset of constraints associated with x in a var-partition $P(\mathcal{C})$ of $\mathcal{C}$. Consider the following corollary of Theorem 1:

Corollary 2 *Let $P(\mathcal{C})$ be a var-partition of $\mathcal{C}$, x a variable and $a \in D(x)$, if $minUnsat((x,a),P(x),S[P(x)])$
$+minUnsat((x,a),\mathcal{C}-P(x),S[\mathcal{C}-P(x)]) > max(D(unsat))$
then (x,a) is not consistent with $ssc(\mathcal{C},S[\mathcal{C}],unsat)$.*

Proof: $\mathcal{C} - P(x)$ and $P(x)$ are disjoint and included in $\mathcal{C}$. Therefore,
$minUnsat((x,a),P(x),S[P(x)]) + minUnsat((x,a),\mathcal{C}-P(x),S[\mathcal{C}-P(x)])$
$\leq minUnsat((x,a),\mathcal{C},S[\mathcal{C}])$. From theorem 1 the corollary holds.

Note that $minUnsat(\mathcal{C}-P(x),S[P(x)]) \leq minUnsat((x,a),\mathcal{C}-P(x),S[P(x)])$. From this remark and Properties 4 and 5 we deduce the following theorem, which corresponds to Equation (2) given in Introduction:

Theorem 2 *$\forall \mathcal{P}(\mathcal{C})$ a var-partition of $\mathcal{C}, \forall x \in X(\mathcal{C}), \forall a \in D(x)$, if $\#inc((x,a),P(x)) + LB(\mathcal{P}(\mathcal{C}-P(x))) > max(D(unsat))$ then a can be removed from its domain.*

5 Conflict Set Based Lower Bound

5.1 Intuitive Idea

Some inconsistencies are not taken into account by the previous lower bound, because it is based on counters of direct violations of constraints by values. This drawback is pointed out in the example of introduction.

In order to take more inconsistencies into account, we propose a new lower bound based on successive computations of disjoint conflict sets.

Definition 6 *A conflict set is a subset $\mathcal{K}$ of $\mathcal{C}$ which satisfies: $minUnsat(\mathcal{K},S[\mathcal{K}]) > 0$.*

We know that a conflict set leads to at least one violation in $\mathcal{C}$. Consequently, if we are able to compute q disjoint conflict sets of $\mathcal{C}$ then q is a lower bound of $minUnsat(\mathcal{C},S[\mathcal{C}])$. They must be disjoint to guarantee that all violations are independent.

For each $C_i \in \mathcal{C}$ such that $D(s_i) = 1$, the set $\{C_i\}$ is a conflict set. Moreover, constraints C_i of $\mathcal{C}$ with $D(s_i) = 0$ are not interesting in the determination of conflict sets. Hence we will focus on the set of constraints C_i of $\mathcal{C}$ with $D(s_i) = \{0,1\}$.

5.2 Computation of Disjoint Conflict Sets

We will denote by isAConflictSet($\mathcal{K}$) the function which returns true if $\mathcal{K}$ is a conflict set and false otherwise.

Determining if a set of constraints $\mathcal{K}$ satisfies the condition of definition 6 is a NP-complete problem. Indeed, it consists of checking the global consistency of the constraint network $\mathcal{N}[\mathcal{K}]$ defined by $\mathcal{K}$ and by the set of variables involved in the constraints of $\mathcal{K}$. However, for our purpose, the identification of some conflict sets is sufficient.

In lack of other algorithms isAConflictSet($\mathcal{K}$) can be defined as follows: it returns true if the achievement of arc consistency on the constraint network $\mathcal{N}[\mathcal{K}]$ leads to a failure (i.e. the domain of one variable has been emptied), and false otherwise. Thus we can consider that we are provided with isAConflictSet($\mathcal{K}$) function.

Let $\mathcal{C}$ be an identified conflict set, we are interested in finding subsets of $\mathcal{C}$ which are themselves conflict sets. Such a conflict set $\mathcal{K} \subseteq \mathcal{C}$ can be easily identified by defining an ordering on $\mathcal{C}$: the principle is to start with an empty set $\mathcal{K}$ and then successively add constraints of $\mathcal{C}$ to $\mathcal{K}$ until isAConflictSet($\mathcal{K}$) returns true. This algorithm can be implemented thanks to OL, a data structure implementing a list of constraints ordered from 1 to *size*. The following basic functions are available:

- OL.$ct[i]$ returns the ith constraint of OL.
- OL.$size$ returns the number of constraints of OL.
- addFirst(C,OL) adds C to OL at first position and shift all the other elements to the right.
- addLast(C,OL) adds C to OL at last position.
- getLast(OL) returns the last constraint in OL.
- removeLast(OL) removes from OL the last constraint in OL and returns it.
- remove(OL, C) removes the constraint C from the OL.

For convenience, given a constraint set $\mathcal{C}$ stored in an OL ol, and $\mathcal{K} \subseteq \mathcal{C}$, $ol - \mathcal{K}$ denotes the OL obtained after calls of function remove(ol,C) for all the constraints C of $\mathcal{K}$.

Given a conflict set $\mathcal{C}$ stored in an OL ol, a subset of $\mathcal{C}$ which is also a conflict set can be computed by calling the function computeConflictSet(ol) which is defined by:

computeConflictSet(OL ol) returns OL
1: $S \leftarrow emptyOL$
2: for $i = 1$ **to** $ol.size$
addLast($ol.ct[i]$,S);
if isAConflictSet(S) **then return** S;
3: return $emptyOL$;

A set of disjoint conflict sets can be easily computed by calling function computeConflictSet(ol) with ol containing all constraint of $\mathcal{C}$ and by iteratively calling it with $ol \leftarrow ol - \mathcal{K}$ each time a conflict set $\mathcal{K}$ is detected in ol.

The lower bound we search for depends on the number of conflict sets, and, since they are disjoint, on the size of the conflict sets.

Definition 7 *Let $\mathcal{C}$ be a set of constraint. A minimal conflict set w.r.t.* computeConflictSet *is a subset $\mathcal{K}$ of $\mathcal{C}$ such that $\forall C \in \mathcal{K}$,* computeConflictSet($\mathcal{K} - \{C\}$) *detects no conflict set.*

A simple algorithm for finding a minimal conflict set from a conflict set was suggested by De Siqueira and Puget [3]. It requires only a monotonic propagation of constraints, that is, not dependent on the order in which constraints are added.

Is is implemented by the function computeMinConflictSet(ol). The first step consists of computing an initial OL $firstOL$. This OL contains a subset of the constraint set given as parameter which forms a conflict set, if such a conflict set can be identified. Then, the algorithm repeatedly calls computeConflictSet with an OL which is the same OL than the previous one, except that the last constraint became the first one. This repetition is done until the last constraint of a new computed OL is the last constraint of $firstOL$. The latest computed OL contains the constraints of a minimal conflict set.

```
computeMinConflictSet(OL ol) returns OL
  1: M ← computeConflictSet(OL);
  2: if M ≠ emptyOL then
         firstLast ← getLast(ol);
         do
             C ← removeLast(M);
             addFirst(C,M);
             M ← computeConflictSet(M);
         while getLast(M) ≠ firstLast
  3: return M;
```

5.3 Conflict Set Based Lower Bound

We can now propose an original algorithm for computing a lower bound of $minUnsat(\mathcal{C}, S[\mathcal{C}])$.

This algorithm is based on computation of disjoint conflict sets. Therefore, it performs successive calls of computeMinConflictSet. This lower bound will be denoted by $LB_{DCS}(\mathcal{C})$:

computeConflictBasedLB($\mathcal{C}$)

1: $LB_{DCS}(\mathcal{C}) \leftarrow min(D(unsat))$;
create an OL ol **and add all the constraints of** $\mathcal{C}$ **to it;**
2: $cs \leftarrow$computeMinConflictSet(ol);
While $cs \neq emptyOL$ **do**
$LB_{DCS}(\mathcal{C}) \leftarrow LB_{DCS}(\mathcal{C}) + 1$;
$ol \leftarrow ol - cs$
$cs \leftarrow$ computeMinConflictSet(ol);
3: return $LB_{DCS}(\mathcal{C})$;

$LB_{DCS}(\mathcal{C})$ can be used to check the consistency of a *ssc*, as the variable-based lower bound $LB(\mathcal{P}(\mathcal{C}))$ described in section 4:

Corollary 3 *If* $LB_{DCS}(\mathcal{C}) > max(D(unsat))$ *then* $ssc(\mathcal{C}, S[\mathcal{C}], unsat)$ *is not consistent.*

6 Identification of Independent Set of Ignored Constraints w.r.t. a Var-Partition

In this section we show how to improve results presented in section 4, by integrating such a conflict set based lower bound of violations into Property 4 and Theorem 2. The idea is to identify ignored constraints, that is, constraints which are not taken into account in $LB(\mathcal{P}(\mathcal{C}))$. Then, it is possible to compute a conflict set based lower bound on a particular subset of these constraints, which can be added to $LB(\mathcal{P}(\mathcal{C}))$.

Definition 8 *Let* $\mathcal{P}(\mathcal{C})$ *be a var-partition. An ignored constraint w.r.t.* $\mathcal{P}(\mathcal{C})$ *is a constraint* C *such that* $\forall x \in X(\mathcal{C}) : \#inc(x, P(x) - \{C\}) = \#inc(x, P(x))$.

Thus, one ignored constraint can be removed from $\mathcal{C}$ without changing the value of $LB(\mathcal{P}(\mathcal{C}))$.

Definition 9 *Let* $\mathcal{P}(\mathcal{C})$ *be a var-partition. A set of constraints* S *satisfying* $\forall x \in X(\mathcal{C}) : \#inc(x, P(x) - S) = \#inc(x, P(x))$ *is called an independent set of ignored constraints w.r.t. the var-partition.*

If an independent set S is found then it is possible to improve Property 4 and Theorem 2, by adding $LB_{DCS}(S)$ to them. The identification of ignored constraints w.r.t var-partition is given by the following property:

Definition 10 *Let* x *be a variable, the set of ignored constraints w.r.t.* $P(x)$ *is the set* $ignored(P(x)) = P(x) - \{C \in P(x), C \text{ is violated by } a \in D(x) \text{ with } \#inc((x,a), P(x)) = \#inc(x, P(x))\}$

Unfortunately, the whole set $\mathcal{K}$ of ignored constraints w.r.t. $P(x)$ is not necessarily independent. Each constraint $C \in \mathcal{K}$ taken separately satisfies $\#inc(x, P(x) - \{C\}) = \#inc(x, P(x))$, but this fact does not guarantee that $\#inc(x, P(x) - \mathcal{K}) = \#inc(x, P(x))$.

For instance, consider a variable x with 3 values a, b and c and suppose that a is not consistent with C_1, b is not consistent with C_2 and c is not consistent with C_3 and C_4; Assume $P(x) = \{C_1, C_2, C_3, C_4\}$.

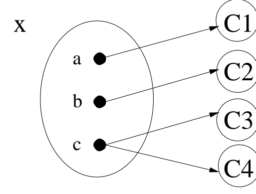

Figure 2

Then, $\#inc(x, P(x)) = 1 = \#inc((x, a), P(x)) = \#inc((x, b), P(x))$ and $ignored(P(x)) = \{C_3, C_4\}$. Unfortunately, $ignored(P(x))$ does not form an independent set of ignored constraints. That is, constraints C_3 and C_4 cannot be simultaneously removed from $P(x)$, because in this case: $\#inc(x, P(x) - \{C_3, C_4\}) = \#inc((x, c), P(x) - \{C_3, C_4\}) = 0$, which is less than $\#inc(x, P(x))$.

Nevertheless, a simple example of an independent set of ignored constraints is the set containing constraints involving only variables with $\#inc$ counters equal to 0. Now, we propose general method to identify such a set. Since $\mathcal{P}(\mathcal{C})$ is a partition, it is sufficient to identify *for each variable* x an independent subset of ignored constraints of $P(x)$. The union of these subsets will form an independent set.

Property 6 *Let $\mathcal{P}(\mathcal{C})$ be a var-partition, x be a variable of $X(\mathcal{C})$ and S be an independent set of ignored constraints included in $P(x)$. Then, $\cup_{x \in X(\mathcal{C})} S$ is an independent set of ignored constraints w.r.t. $\mathcal{P}(\mathcal{C})$.*

Thus, we can focus our attention to the determination of an independent set of ignored constraints included in a $P(x)$:

Property 7 *Let T be any subset of $P(x)$. If each value of $D(x)$ violate at least $\#inc(x, P(x))$ constraints of T, then $S = P(x) - T$ is an independent set of ignored constraints.*

Proof: $\forall a \in D(x)$ $\#inc((x, a), P(x) - S) = \#inc((x, a), T)$ which is greater than or equal to $\#inc(x, P(x))$. Therefore, by definition 9, S is an independent set of ignored constraints.

Such a set T can be found by solving a covering problem:

Proposition 1 *Let x be a variable, $G(x, P(x)) = (D(x), P(x), E)$ be the bipartite graph such that $(a, C) \in E$ iff $a \in D(x), C \in P(x)$ and (x, a) violates C. Let T be*

a subset of $P(x)$ *such that* $\forall a \in D(x)$ *there are at least* $\#inc(x, P(x))$ *edges with an endpoint in* T*. Then,* $S = P(x) - T$ *is an independent set of ignored constraints w.r.t.* $P(x)$*.*

The proof of this proposition is straightforward. Finding a minimal set T is an NP-Complete problem, but it is not mandatory to search for a minimal set. From Property 7, we propose a greedy algorithm which returns an independent set of constraints from a set $P(x)$ of a var partition $\mathcal{P}(\mathcal{C})$ ($\#inc(x, K - \{C\})$ can be easily updated at each step):

computeIndependentSet$(x, P(x))$

1: $K \leftarrow P(x)$**;**
2: $S \leftarrow \emptyset$**;**
3: While $\exists C \in K$, $\#inc(x, K - \{C\}) \geq \#inc(x, P(x))$ **do**
$S \leftarrow S \cup \{C\}$**;**
$K \leftarrow K - \{C\}$**;**
4: return S**;**

$ISNC(\mathcal{P}(\mathcal{C})) = \cup_{x \in X(\mathcal{C})}$computeIndependentSet$(x, P(x))$ is an independent set of ignored constraints w.r.t. $\mathcal{P}(\mathcal{C})$. We can propose a new property which improve Property 4, and the corresponding necessary condition of consistency of a *ssc*:

Property 8 $\forall \mathcal{P}(\mathcal{C})$ *a var-partition of* $\mathcal{C}$*,*
$LB(\mathcal{P}(\mathcal{C})) + LB_{DCS}(ISNC(\mathcal{P}(\mathcal{C}))) \leq minUnsat(\mathcal{C}, S[\mathcal{C}])$

Corollary 4 *If* $LB(\mathcal{P}(\mathcal{C})) + LB_{DCS}(ISNC(\mathcal{P}(\mathcal{C})))$
$> max(D(unsat))$ *then* $ssc(\mathcal{C}, S[\mathcal{C}], unsat)$ *is not consistent.*

Note that if we compute these bounds in the example given in Introduction, we obtain the following result: at least one constraint among $x < y$, $y < z$ and $z < x$ is violated for any var-partition, since in all cases the independent set of ignored constraints contains the three constraints.

Moreover, Property 8 can be used in order to improve the filtering Theorem 2:

Theorem 3 $\forall \mathcal{P}(\mathcal{C})$ *a var-partition of* $\mathcal{C}, \forall x \in X(\mathcal{C}, \forall a \in D(x)$*,*
if $\#inc((x, a), P(x)) + LB(\mathcal{P}(\mathcal{C} - P(x)))$
$+ LB_{DCS}(ISNC(\mathcal{P}(\mathcal{C} - P(x))))$
$> max(D(unsat))$ *then a can be removed from its domain.*

7 Summary

The two following tables recapitulate the results of this paper and compare them to the previous studies. Let $P(\mathcal{C})$ be a var-partition of $\mathcal{C}$:

1. Consistency:

Previous studies (binary constraints)
$LB(\mathcal{P}(\mathcal{C}))$ $> max(D(unsat))$
New Condition (any arity)
$LB_{DCS}(\mathcal{C})$ $> max(D(unsat))$
Improved Condition (any arity)
$LB(\mathcal{P}(\mathcal{C}))$ $+LB_{DCS}(ISNC(\mathcal{P}(\mathcal{C})))$ $> max(D(unsat))$

2. Filtering algorithm:

Previous studies (binary constraints)
$\#inc((x,a),P(x))$ $+LB(\mathcal{P}(\mathcal{C}-P(x)))$ $> max(D(unsat))$
New results (any arity)
$\#inc((x,a),P(x))$ $+LB(\mathcal{P}(\mathcal{C}-P(x)))$ $+LB_{DCS}(ISNC(\mathcal{P}(\mathcal{C}-P(x))))$ $> max(D(unsat))$

8 Conclusion

Some new properties improving existing results have been proposed. The lower bounds presented in this paper take into account some inconsistencies between constraints that are ignored by the previous studies. The constraints ignored by the existing algorithms for Max-CSP have been identified and an algorithm for computing a lower bound of the number of inconsistencies implied by these constraints have been proposed. One additional advantage of the framework we suggest is that the filtering algorithm associated with the constraints are used in a way similar to classical CSPs. Moreover, all the results make no assumption on the arity of constraints and generalize the previous studies which consider only binary Max-CSP.

9 Acknowledgements

The work of ILOG authors was partially supported by the IST Programme of the Commission of the European Union through the ECSPLAIN project (IST-1999-11969). We would like to thank Ulrich Junker and Olivier Lhomme for helpful comments they provided on the ideas of this paper.

References

1. S. Bistarelli, U. Montanari, F. Rossi, T. Schiex, G. Verfaillie, and H. Fargier. Semiring-based csps and valued csps: Frameworks, properties, and comparison. *Constraints*, 4:199–240, 1999.
2. E. Freuder and R. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58:21–70, 1992.
3. J.L. de Siqueira N. and J. Puget. Explanation-based generalization of failures. *Proceedings ECAI*, pages 339–344, 1988.
4. J. Larrosa, P. Meseguer, and T. Schiex. Maintaining reversible dac for max-csp. *Artificial Intelligence*, 107:149–163, 1999.
5. J. Larrossa and P. Meseguer. Partition-based lower bound for max-csp. *Proceedings CP*, pages 303–315, 1999.
6. T. Petit, J. Régin, and C. Bessière. Meta constraints on violations for over constrained problems. *Proceedings ICTAI*, 2000.
7. P. Van Hentenryck. Constraint satisfaction in logic programming. *The MIT Press*, 1989.
8. G. Verfaillie, M. Lemaître, and T. Schiex. Russian doll search for solving constraint optimisation problems. *Proceedings AAAI*, pages 181–187, 1996.
9. R. Wallace. Directed arc consistency preprocessing as a strategy for maximal constraint satisfaction. *Proceedings ECAI*, pages 69–77, 1994.