A General Conflict-Set Based Framework for Partial Constraint Satisfaction

Thierry Petit[1], Christian Bessière[2] and Jean-Charles Régin[3]

[1]Ecole des Mines de Nantes,
4 rue Alfred Kastler, 44307 Nantes Cedex 3, France.
thierry.petit@emn.fr
[2]LIRMM,
161 Rue Ada, 34392 Montpellier Cedex 5, France.
bessiere@lirmm.fr
[3]ILOG,
1681 routes des Dollines, 06560 Valbonne, Sophia Antipolis, France.
regin@ilog.fr

Abstract. Partial constraint satisfaction [5] was widely studied in the 90's, and notably Max-CSP solving algorithms [21, 20, 1, 10]. These algorithms compute a lower bound of violated constraints without using propagation. Therefore, recent methods focus on the exploitation of propagation mechanisms to improve the solving process. Soft arc-consistency algorithms [11, 18, 19] propagate inconsistency counters through domains. Another technique consists of using constraint propagation to identify conflict-sets which are pairwise disjoint [16]; the number of conflict-sets extracted leads to a lower bound. In this paper, we place this technique in a more general context. We show this technique reduces to a polynomial case of the NP-Complete Hitting Set problem. Conflict-sets are chosen disjoint to compute the lower bound polynomially. We present a new polynomial case where the conflict-sets share some constraints, and a third case which is not polynomial but such that the cardinality of the Hitting Set can be reasonably under estimated. For each one we provide the algorithm and a schema to generate incrementally the conflict-set collection. We show its extension to weighted CSPs.

1 Introduction

A Constraint Network $\mathcal{R}$ is a triplet $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ where $\mathcal{X}$ is a set of variables and $\mathcal{D}$ is a set of domains such that a domain $D(x)$ is associated with each $x \in \mathcal{X}$. $D(x)$ defines the values that can be assigned to x. $\mathcal{C}$ is a set of constraints. Each constraint expresses a restrictive property on a subset of variables. The Constraint Satisfaction Problem (CSP) consists of finding a complete assignment of values to variables in $\mathcal{X}$ such that no constraint is violated.

However, it often happens that a CSP has no solution. In this case the goal is to find a good compromise and some constraints can be violated according to certain rules. The Maximal Constraint Satisfaction Problem (Max-CSP) consists of finding an assignment of values to variables which minimizes the number of

violations (i.e., the number of constraints violated by the solution). The propositional logic formulation of this problem is Max-SAT [6]. Although this framework is very simple, at least two strong arguments motivate the study of algorithms for solving Max-CSP : 1. the existence of sub-problems of real-world applications which correspond to Max-CSPs. 2. the ability to encapsulate these sub-problems into global constraints where the solving algorithms are integrated as filtering algorithms [15, 13].

Existing algorithms for Max-CSP are basically designed to follow a branch-and-bound schema. Branch and Bound is not required and any search algorithm can be adapted [16] but it is simpler to present the algorithms in this way. A depth first exploration of the search tree is performed, where internal nodes represent partial assignments of values to variables. A leaf that ends a branch of $|\mathcal{X}|$ nodes corresponds to a complete assignment. At each node, P (past) denotes the set of variables having a domain of size one (we call them instantiated variables) and F (future) denotes $\mathcal{X} \setminus P$. We call *distance* the number of constraints such that all the variables involved in these constraints are in P and which are violated. The goal is then to minimize the objective UB which is the best number of violations of a complete assignment found so far. If a node is such that $distance \geq UB$ then it is not useful to continue the search below this node. Any complete assignment extended from this current partial assignment will not improve UB. This condition is improved by the computation of an underestimation LB (lower bound) of the minimal distance of any leaf of depth $|\mathcal{X}|$ located below the current node. Obviously $LB \geq distance$, because *distance* is the exact number of violated constraints which involve instantiated variables. If $LB \geq UB$ then it is not useful to continue the search below the current node because we will not improve the best solution found so far. Note that the lower bound LB can also be used in filtering algorithms[1] [10].

Recent works focus on the exploitation of propagation mechanisms to improve the value of LB, through soft arc-consistency algorithms ([11, 18, 19], initially introduced in [4]) or by using conflict-set based algorithms [16]. These methods detect violations which are ignored by the previous reference algorithm PFC-MRDAC [10]. The conflict-set based lower bound for Max-CSP was also proven to be combinable with the lower bound computed by PFC-MRDAC.

In this paper, we propose new conflict-set based algorithms. Firstly, we recall some definitions about conflict-sets in CSPs. We show that the technique initially presented in [16] is a particular polynomial case of a NP-Complete problem. Then, we present a second polynomial case which leads to a new lower bound of the number of violations, computed from a matching algorithm. We propose a third algorithm for a case which is not polynomial but such that a good approximation can be computed. We show how to generate incrementally the different collections of conflict-sets. We show the extension to weighted CSPs.

[1] We define here a filtering algorithm as an algorithm which removes from the domains of future variables the values which cannot belong to a solution.

2 Conflict-set Detection

A conflict-set is a set of constraints that leads to a contradiction [9]. No feasible solution contains a conflict-set. In this paper we consider a sub-class of conflict-sets, formed by the conflict-sets which can be identified by propagating constraints. For sake of clarity, we assume that all the filtering algorithms of the constraints enforce arc-consistency. In all definitions and theorems above we will consider a constraint network $\mathcal{R} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$.

Notation 1 *Let $\mathcal{K} \subseteq \mathcal{C}$. $\mathcal{X}[\mathcal{K}]$ is the subset of variables in $\mathcal{X}$ which occur in at least one constraint of $\mathcal{K}$. The set of domains of variables in $\mathcal{X}[\mathcal{K}]$ is denoted by $\mathcal{D}[\mathcal{K}]$.*

Notation 2 *$\mathcal{D}' \sqsubseteq \mathcal{D}$ means $\forall x_{i_k} \in \mathcal{X}, D'(x_{i_k}) \in \mathcal{D}'$ satisfies $D'(x_{i_k}) \subseteq D(x_{i_k})$.*

Definition 1. *Let $C \in \mathcal{C}$ defined on variables $var(C) = \{x_{i_1}, \ldots, x_{i_k}\}$. An element of $D(x_{i_1}) \times \ldots \times D(x_{i_k})$ is called a tuple of $var(C)$. Value v for variable x is denoted (x, v). A tuple τ of $var(C)$ is valid if $\forall (x, v) \in \tau, v \in D(x)$. Value $v \in D(x)$ is consistent with a constraint C iff $x \notin var(C)$ or $\exists \tau$ valid such that $(x, v) \in \tau$. v is arc-consistent iff $\forall C \in \mathcal{C}$, v is consistent with C. A domain $D(x)$ is arc-consistent iff $D(x) \neq \emptyset$ and all values in $D(x)$ are arc-consistent. Then we say that x is arc-consistent. $\mathcal{R}$ is arc-consistent iff all variables in $\mathcal{X}$ are arc-consistent. $\mathcal{R}$ is arc-inconsistent iff $\forall\ \mathcal{D}' \sqsubseteq \mathcal{D}$, $\mathcal{R}' = (\mathcal{X}, \mathcal{D}', \mathcal{C})$ is not arc-consistent.*

Definition 2. *Let $\mathcal{K} \subseteq \mathcal{C}$. $\mathcal{K}$ is a conflict-set of $\mathcal{C}$ iff $\mathcal{R}[\mathcal{K}] = (\mathcal{X}[\mathcal{K}], \mathcal{D}[\mathcal{K}], \mathcal{K})$ has no solution.*

We will extract conflict-sets from a set of constraints by propagating successively the constraints until a failure occurs (when a domain is emptied).

Definition 3. *Let $\mathcal{K} \subseteq \mathcal{C}$. $\mathcal{K}$ is an AC-Conflict-set of $\mathcal{C}$ iff $\mathcal{R}[\mathcal{K}] = (\mathcal{X}[\mathcal{K}], \mathcal{D}[\mathcal{K}], \mathcal{K})$ is arc-inconsistent.*

Now we define what is an AC-conflict-set minimum "by inclusion":

Definition 4. *A minAC-Conflict-set $\mathcal{K}$ is an AC-Conflict-set such that $\forall C \in \mathcal{K}$, $(\mathcal{K} \setminus \{C\})$ is not an AC-conflict-set.*

It is possible to reduce an AC-Conflict-set to a minAC-Conflict-set with efficient polynomial algorithms [7, 8]. The principle is described in section 4.1.

3 Conflict-set Based Lower Bounds for Max-CSP

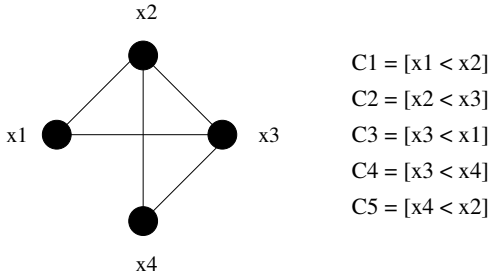
In all this section we will consider only minAC-Conflict-sets.

3.1 Principle

From definition 2 we know that the existence of one minAC-conflict-set leads to at least one violation. The intuitive idea is to identify several minAC-conflict-sets to compute a lower bound of violations. Such conflict-sets will be extracted from the constraints involving some future variables (for each constraint involving only instantiated variables we know wether it is satisfied or not: either we have a conflict-set of size one, either the constraint does not belong to a minAC-Conflict-set. The constraint is taken into account in the *distance*).

It is not safe to use the cardinality of a set of minAC-conflict-sets as a lower bound because of the constraints common to several conflict-sets. Consider the following example :

Example 1. Let $D(x_1) = D(x_2) = D(x_3) = D(x_4) = \{0, 1, 2, 3\}$. $\mathcal{C} = \{C_1, C_2, C_3, C_4, C_5\}$ where: $C_1 = [x_1 < x_2]$, $C_2 = [x_2 < x_3]$, $C_3 = [x_3 < x_1]$, $C_4 = [x_3 < x_4]$, $C_5 = [x_4 < x_2]$.



Two minAC-conflict-sets have a common constraint, $\mathcal{K}_1 = \{C_1, C_2, C_3\}$ and $\mathcal{K}_2 = \{C_2, C_4, C_5\}$, and the CSP defined by $\mathcal{R} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ has a solution which violates exactly one constraint, $C_2 : \{(x_1, 1), (x_2, 2), (x_3, 0), (x_4, 1)\}$.

In this example the necessary condition of existence of a solution which violates only one constraint is that C_2 is common to the two minAC-conflict-sets. If such a solution exists then the violated constraint is necessarily C_2. More generally, the problem of evaluation of the minimum number of violations LB induced by a set of minAC-Conflict-sets is a hitting problem. We search for the minimum number of constraints required to match all the minAC-conflict-sets. This is the Hitting Set [6] problem: given a collection of subsets of a set E, the goal is to find a cover $E' \subseteq E$ of minimum size of the collection of subsets. In our context this size is a lower bound.

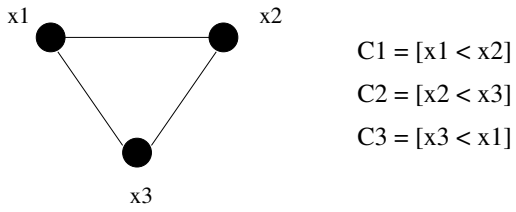
The Hitting Set problem is NP-Complete but it would be wrong to conclude that this complexity forbids us to use conflict-set to compute a lower bound of violations for Max-CSP. According to some particular properties of the collection of minAC-Conflict-sets generated it is possible to compute polynomially or to approximate the lower bound. This remark is enforced by the fact that the number of conflict-sets can itself be huge: it is not reasonable to generate all of them. Therefore, we can build a specific set of minAC-Conflict-sets that respects some properties. In the remaining of this section we will present different collections of minAC-Conflict-sets and the algorithms used to compute the lower bound LB.

3.2 Disjoint Conflict-Sets

Régin et al. [15] suggested to guarantee the independence of violations counted for each minimal AC-Conflict-set by considering a collection Ψ of conflict-sets disjoint one another. Since the conflict-sets have no constraint in common, the LB is equal to the number of conflict-sets in the collection. The property of having minimal AC-Conflict-set is not necessary but leads to a better lower bound: the smaller are the conflict-sets computed, the greater is the set of constraints that can be used to compute new ones. Finally the number of conflict-set generated should also be greater. We denote by Ψ the collection of disjoint minAC-Conflict-Sets.

computeLB(Ψ) {
  return $|\Psi|$;
}

This technique has proven to detect inconsistencies ignored by the best "non-propagating" algorithm PFC-MRDAC [10]. For instance, cycles of inequalities that occur in scheduling problems (precedence constraints). They can be detected because they form conflict-sets, whereas they are ignored by PFC-MRDAC.



D(x1) = D(x2) = D(x3) = {1,2,3}

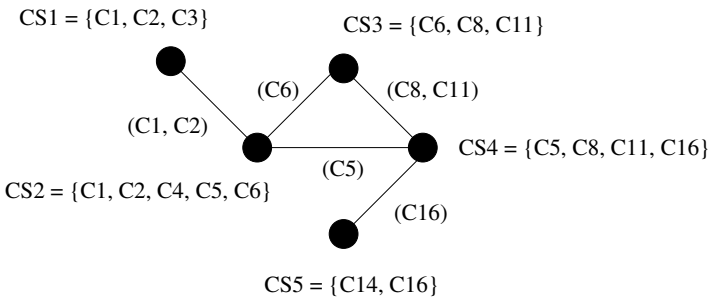
Value 2 in each domain does not violate directly the constraint. Propagation is required to detect the violation.

A method for combining this bound with the one of PFC-MRDAC is proposed in [16]. The principle of this combination can be applied with no restriction to all the other collections of minimal AC-Conflict-sets presented in this paper.

3.3 Constraints Common to At Most Two Conflict-sets

Assume that given two conflict-sets, no restriction is made on the total number of constraints they share, but each of these constraints do not belong to a third conflict-set. In this case we can represent the collection by a graph:

– Vertices are conflict-sets,
– Each edge represents a constraint or a set of constraints shared by two conflict-sets (the two conflict-sets are represented by the two incident vertices of the edge).



The lower bound can be computed by a polynomial algorithm. Indeed, the problem consists of finding an Edge Cover of the graph. In the Edge Cover problem the goal is to find the minimum number of edges required to cover all the vertices. In [12, 6] the Edge Cover problem is mentioned to be solvable by matching algorithms. Let us go further in details. We call $d(x)$ the number of edges incident to a vertex x in a graph G (the degree of x).

Definition 5. *Let $G = (\mathcal{X}, E)$ be a graph. A matching is a subset $E' \subseteq E$ such that no two edges in E' have a vertex in common. $|E'|$ is called the cardinality of the matching. A matching of maximum cardinality μ is called a maximum matching.*

Theorem 1. : Norman and Rabin, 1959 [14]
Let $G = (\mathcal{X}, E)$ be a graph. Let μ be the cardinality of a maximum matching in G. $|\mathcal{X}| - \mu$ is the cardinality of a minimum edge cover in G.

We propose a proof that gives the intuitive idea of the result:

Proof: let E' be an edge cover of minimal cardinality $|E'|$. Let $(u, v) \in E'$. Consider $G' = (X, E')$. Then either $d(u) = 1$ or $d(v) = 1$ in G'. If it was not the case E' would not be minimal, since by removing (u, v) we would obtain another edge cover. If we remove from each vertex of degree > 1 all its incident edges but one then we obtain a matching M of cardinality m. Each of the m edges of this matching covers two vertices, so the matching covers $2 * m$ vertices. By definition to cover any vertex which does not belong to M one edge is necessary, that is to cover all of them, $|X| - 2 * m$ edges are necessary. So $|E'| = m + |X| - 2 * m = |X| - m$. By definition of a maximum matching of cardinality μ' in G' we have $m \leq \mu'$. Since G contains more edges than G' we have a fortiori $m \leq \mu$. Consequently, $|E'| \geq |X| - \mu$ (1). Moreover, let M be a maximum matching of cardinality μ. $2 * \mu$ vertices are covered by M and $|X| - 2 * \mu$ vertices remain. Select one edge incident to each vertex that does not belong to the matching. The second vertex of such an edge necessary belongs to the matching. Indeed, if it was

not the case then the matching would not be maximum since by adding the edge we would obtain a matching of cardinality $\mu+1$. Finally, if we add the selected edges to the ones of the matching we obtain an edge cover. Its cardinality is $\mu+|X|-2*\mu = |X|-\mu$. We deduce $|E'| \leq |X| - \mu$ (2). From (1) and (2) we know that $|X| - \mu$ is exactly the cardinality of a minimum edge cover.

The following algorithm is stem from this theorem. $\Psi = \{\mathcal{K}_1, \ldots, \mathcal{K}_{|\Psi|}\}$ denotes a set of conflict-sets and E the set of edges, each one representing a non empty intersection between two conflict-sets. If the graph has some isolated vertices, we just need to count them and add their number to the lower bound, because each one corresponds to a disjoint conflict-set.

```
computeLB(G = (Ψ, E)) {
    LB ← 0
    for each x ∈ Ψ s.t. d(x) = 0 do
        LB ← LB + 1;
        Ψ ← Ψ \ {x};
    μ ← cardinality of a maximum matching in G;
    LB ← LB + |Ψ| − μ;
    return LB;
}
```

Any number of constraints can be shared by two conflict-sets (violating one of them is sufficient to cover the two conflict-sets). This property enforces the interest of such a computation. The complexity for computing a matching on a bipartite graph $G = (\Psi, E)$ is $O(|E| * \sqrt{|\Psi|})$ [2]. When constraints are common to more than two conflict-sets the problem is not polynomial.

3.4 Unary Intersection between Pairs of Conflict-Sets

The Hitting Set problem remains NP-Complete even when the considered subsets have a size of at most 2 [6]. We deduce that, more generally, even in the case where any intersection between two minAC-Conflcit-set is empty or a singleton computing the lower bound remains NP-Complete. We propose to under-estimate the cardinality of the minimum number of constraints required to cover all the minAC-Conflict-sets.

Let $G = ((\mathcal{C}, \Psi), E)$ the bipartite graph[2] such that an edge is defined between a constraint C and a conflict-set $\mathcal{K} \in \Psi$ iff $C \in \mathcal{K}$. Let us order the constraints by decreasing degrees in G. The degree is the number of incident edges of a vertex. Consider the subset $\mathcal{C}'$ of the first constraints such that:

- The sum of degrees of constraints in $\mathcal{C}'$ is greater or equal than $|\Psi|$
- $\forall$ $\mathcal{C}$" $\subset \mathcal{C}'$ the sum of degrees is strictly less than $|\Psi|$.

[2] A graph $G = ((X, X'), E)$ is bipartite iff $\forall e = (u, v) \in E$, either $u \in X$ and $v \in X'$ or $u \in X'$ and $v \in X$.

In any graph the minimal possible number of constraints required to cover all the conflict-sets cannot be less than $|\mathcal{C}'|$. Therefore the cardinality of $\mathcal{C}'$ is a lower bound. This technique can be applied in the general case but it should be precise only when (almost all) intersections are unary.

```
computeLB(G = ((C, Ψ), E)){
    LB ← 0;
    sumDeg ← 0;
    L ← list of constraints ∈ C ordered by decreasing degree in G;
    while sumDeg < |Ψ| do
        pick and remove the first constraint C from L;
        sumDeg ← sumDeg+ degree of C;
        LB ← LB + 1;
    return LB;
}
```

Given Ψ, the complexity of this algorithm is $O(|\mathcal{C}|)$. Note this approximation can be related to the minimum constraint family algorithms presented by Beldiceanu [3].

4 Generation of Collections

In this section we discuss the generation of the set Ψ of minAC-Conflict-sets useful to compute the lower bound LB, according to the property we want to maintain: disjoint conflict-sets, constraints shared at most once, unary intersections between pairs of conflict-sets. We present an incremental schema based on the concepts presented in [17] for disjoint conflict-sets. We maintain two sets:

- $\mathcal{A} \subseteq \mathcal{C}$ the set of constraints available to compute new conflict-sets.
- Ψ the current collection of conflict-sets.

Initially, when the search starts, $\Psi = \emptyset$ and $\mathcal{A} = \mathcal{C}$. The first function useful to all the algorithms we present is the one used to maintain AC-Conflict-sets which are min-AC-Conflict-sets.

4.1 Minimizing AC-Confllict-sets

Notation 3 *Given a set of constraint $\mathcal{A}$ and a total order σ on the constraints, we denote by $\mathcal{A}_\sigma$ the set containing all constraints in $\mathcal{A}$ ordered by σ.*

The function computeAC-Conflict-Set($\mathcal{A}_\sigma$) returns the set $\mathcal{K}$ of the k first constraints in $\mathcal{A}_\sigma$ such that the network $\mathcal{R}[\mathcal{K}_\sigma]$ is arc-inconsistent and the network obtained by considering $\mathcal{K}_\sigma$ but its last constraint C_{last} is arc-consistent (this function will be described in section 4.2.2 because it depends on each collection). $\mathcal{K}$ forms a conflict-set but it is not necessarily minimal. We only know that C_{last} belongs to the minimal conflict-set we will finally extract from $\mathcal{K}$.

Therefore, the following schema is then repeated successively : we consider a new order σ' on constraints deduced from the previous one by placing last constraint in the conflict-set at the first position. $\mathcal{K}$ is assigned with the result of computeAC-Conflict-Set($\mathcal{K}_{\sigma'}$). The new conflict-set obtained may be smaller than the previous one if a domain has been emptied by propagating a smaller number of constraints, because the ordering has changed.

The schema stops when the last constraint in the conflict-set is again C_{last}. A complete turn has been made. By construction the last AC-Conflict-set generated cannot be reduced more. It is a minAC-Conflict-set.

4.2 Incremental Generation

At each node, we will first remove some constraints from Ψ and add them to $\mathcal{A}$. This step is required if we need to maintain conflict-sets which are minimal. The principles are explained below.

4.2.1 Removal of Constraints from the collection Ψ When a minAC-Conflict-set $\mathcal{K}$ is generated and added to Ψ, it is such that $\forall C \in \mathcal{K}$, $(\mathcal{K} - \{C\})$ is not an AC-conflict-set. However, when we continue the search, domains of variables may be reduced. The previous AC-Conflict-sets remain AC-conflict-sets but they are not necessarily minimal. From these two properties we can update the previous set of conflict-sets Ψ instead of recomputing it from scratch. In this section we focus on the constraints that can be removed from Ψ to be re-added to $\mathcal{A}$. Next section will focus on the generation of new conflict-sets from the updated set $\mathcal{A}$. We consider that minAC-Conflict-sets are required.

A. General Schema We first describe a general algorithm common to all collections. This algorithm calls updateCtData, a function specific to each collection. This function is used to update some data required to maintain the property defining the collection. It will described just after for each collection (see paragraphs B., C., D.). The algorithm uses also the computeMinAC-Conflict-Set function defined in section 4.1.

```
updateData(Ψ, A) {
    for each K ∈ Ψ do
        K' ← computeMinAC-Conflict-Set(K);
        Ψ ← (Ψ \ {K}) ∪ {K'};
        for each C ∈ K \ K' such that C ∉ A do
            A ← A ∪ {C};
            updateCtData(K, K');
    return (Ψ, A);
}
```

The complexity is $|\mathcal{C}|^2$ times the cost of propagating a constraint (the worst case, when the size of $\mathcal{A}$ is negligible compared with Ψ). Indeed as we will

see the cost of the updateCtData procedure is not significant. Note that in the case of a collection of disjoint conflict sets it is not required to have minimal AC-Conflict-sets; to obtain a cheaper computation we may replace the call to computeMinAC-Conflict-Set($\mathcal{K}$) by a call to computeAC-Conflict-Set($\mathcal{K}$, σ) with any order σ. Now, let us define the updateCtData procedure according to each collection.

B. Disjoint Minimal Conflict Sets In this case it is not required to store anything concerning the constraints.

```
updateCtData(𝒦, 𝒦′) { }
```

C. Constraints Shared at Most Once In this case we will maintain for each $\mathcal{C} \in \mathcal{A}$ the number of conflict-sets where it appears. For sake of clarity we consider a global data structure such that for each constraint $C \in \mathcal{C}$ the value $\#\mathbf{CS}(C)$ is the number of conflict-sets $\mathcal{K} \in \Psi$ such that $C \in \mathcal{K}$.

```
updateCtData(𝒦, 𝒦′) {
    for each C ∈ 𝒦 \ 𝒦′ do
        #CS(C) ← #CS(C) − 1;
}
```

D. Unary Intersection between Pairs of Conflict-sets In this case we will maintain the bipartite graph $G = ((\mathcal{C}, \Psi), E)$ such that an edge is defined between a constraint C and a conflict-set $\mathcal{K} \in \Psi$ iff $C \in \mathcal{K}$. By removing edges we keep stable on the property of having unary intersections. For sake of clarity we consider the graph as a global structure.

```
updateCtData(𝒦, 𝒦′) {
    remove vertex 𝒦 from G and all its incident edges;
    add a new vertex 𝒦′ to G;
    for each C ∈ 𝒦′ do
        add (C, 𝒦′) in E;
}
```

4.2.2 Generation of Conflict-sets from the set $\mathcal{A}$ In this section we describe how to increase the size of the collection Ψ from the set of available constraints $\mathcal{A}$.

A. General Schema The algorithm uses a sub-routine that checks if a set of constraint $\mathcal{K}$ is an AC-Conflict-set (by achieving arc-consistency). We call this sub-routine isAC-Conflict-set. Each conflict-set detected in $\mathcal{A}$ is added to Ψ, and $\mathcal{A}$ and eventually the global structures are updated. The function used to extract an AC-Conflict-set with respect to the properties required by the collection is called computeAC-Conflict-Set. The function used to update $\mathcal{A}$ and

the global data with respect to the collection is called updateAfterGenerate. It takes $\mathcal{A}$ and the current extracted conflict-set $\mathcal{K}$ as argument. These two functions are specific to each collection and will be described later. Beforehand, we define the common part, that is, the algorithm computeCollection used to generate a collection of min-AC-Conflict-sets. The order σ on constraints is arbitrary, but we keep it to have only one definition of computeAC-Conflict-Set in the paper (see section 4.1).

```
computeCollection(A_σ, Ψ) {
    K ← computeAC-Conflict-Set(A_σ);
    while K ≠ ∅ do
        K ← computeMinAC-Conflict-Set(K);
        Ψ ← Ψ ∪ {K};
        A_σ ← updateAfterGenerate(A_σ, K);
        K ← computeAC-Conflict-Set(A_σ);
    return (A_σ, Ψ);
}
```

B. Disjoint Minimal Conflict Sets The constraints in $\mathcal{A}_\sigma$ are successively added until a failure occurs.

```
computeAC-Conflict-Set(A_σ) {
    K ← {first constraint C in A_σ}
    A_σ ← A_σ \ {C}
    while ¬ isAC-Conflict-Set(K) ∧ A ≠ ∅ do
        K ← K ∪ {first constraint C in A_σ}
        A_σ ← A_σ \ {C}
    if ¬ isAC-Conflict-Set(K) then return ∅;
    return K;
}
```

All the constraints of the minAC-Conflict-set $\mathcal{K}$ are removed from $\mathcal{A}$.

```
updateAfterGenerate(A_σ, K) {
    A_σ ← A_σ \ K;
    return A_σ;
}
```

The complexity of the computeCollection algorithm is $|\mathcal{A}_\sigma|^2$ times the cost of propagating a constraint.

C. Constraints Shared at Most Once The function computeAC-Conflict-Set($\mathcal{A}$, σ) is identical to the previous one. The condition for removing a constraint C from $\mathcal{A}$ depends on value of the accessor $\#\mathbf{CS}(C)$, which gives the number of conflict-sets $\mathcal{K} \in \Psi$ such that $C \in \mathcal{K}$. $\#\mathbf{CS}(C)$ (see section 4.2).

```
updateAfterGenerate(A_σ, K) {
    for each C ∈ K do
        #CS(C) ← #CS(C) + 1;
        if #CS(C) = 2 then A_σ ← A_σ \ {C};
    return A_σ;
}
```

The order of complexity is the same as in the disjoint case.

D. Unary Intersection between Pairs of Conflict-sets In this case the algorithm used to compute each conflict-set must verify that all intersections with existing conflict-sets are unary. We use the bipartite graph $G = ((\mathcal{C}, \Psi), E)$ such that an edge is defined between a constraint C and a conflict-set $\mathcal{K} \in \Psi$ iff $C \in \mathcal{K}$. It is defined as a global structure. This graph is useful to compute the lower bound and in the algorithm below it is used to check if constraints belong to conflict-sets.

```
computeAC-Conflict-Set(A_σ) {
    K ← ∅;
    C ← first constraint C in A_σ
    A_σ ← A_σ \ {C}
    add ← true;
    while ¬ isAC-Conflict-Set(K) ∧ A_σ ≠ ∅ do
        for each C' ∈ K s.t. C' ≠ C do
            if ∃ K' ∈ Ψ: C, C' ∈ K' then
                add ← false;
        if add then K ← K ∪ {C};
        C ← {first constraint C in A_σ}
        A_σ ← A_σ \ {C}
        add ← true;
    if ¬ isAC-Conflict-Set(K) then return ∅;
    return K;
}
```

The updateAfterGenerate function updates G with each new conflict-set. Initially, when we start the search, G contains all constraints as isolated vertices.

```
updateAfterGenerate(A_σ, K) {
    add a new vertex K to G;
    for each C ∈ Kdo
        add (C, K) in E;
    return A_σ;
}
```

The order of complexity of the computeCollection algorithm remains $|\mathcal{A}|^2$ times the cost of propagating a constraint.

5 Perspectives: Weighted CSP

We recall that Weighted CSP [5] is the framework where a weight is associated with each constraint and the goal is to minimize the sum of weights of violated constraints. Adapting the approach of disjoint conflict-sets to weighted CSPs just consists of counting the weight of a constraint of minimal weight in each minAC-Conflict-Set instead of counting one[3]. When constraints can be shared by conflict-sets, it is also possible to adapt the framework to the weighted case. Consider two minAC-Conflict-sets that share at least one constraint. To perform the sum of weights, it is not safe to consider only the constraints common to these two conflict-sets. Indeed, it may exist two constraints which do not belong to the intersecting set and such that their sum of weights is less that the minimum existing weight of a constraint in the common set. Therefore the contribution of these two conflict-sets will be the minimum value obtained by comparing the minimal weight of a constraint shared by the two conflict-sets and the sum of the two minimal weights of constraints in the conflict-sets, one per conflict-set. Since we only compute the cardinality of an edge cover, for each conflict-set we will consider the minimum value over all the possible intersections. In this way it is possible to obtain a lower bound of the sum of weights of any complete assignment below the current node. Since we can only take the minimum weight into account the LB may likely suffer. Experimentation is needed. Therefore this extension should be considered as an interesting topic for future work.

6 Conclusion

We proposed a general conflict-set based framework for computing a lower bound of violations in a Max-CSP that extends the algorithms proposed in [16]. We presented a schema to generate incrementally the conflict-set collection and to compute incrementally the lower bound. We discussed the extension to weighted CSPs.

References

1. M. S. Affane and M. Bennaceur. A weighted arc consistency technique for Max-CSP. *Proceedings ECAI*, pages 209–213, 1998.
2. R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. Networks flows: theory, algorithms, and applications. *Prentice Hall, Inc.*, 1993.
3. N. Beldiceanu. Pruning for the minimum constraint family and for the number of distinct values constraint family. *Proceedings CP*, pages 377–391, 2001.
4. S. Bistarelli, P. Codognet, Y. Georget, and F. Rossi. Labeling and partial local consistency for soft constraint programming. *Second International Workshop on Practical Aspects of Declarative Languages, PADL'OO*, pages 230–248, 1999.

[3] Max-CSP can be encoded as a Weighted CSP where all the constraints have a weight equal to one.

5. E.C. Freuder and R.J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58:21–70, 1992.
6. M. R. Garey and D. S. Johnson. Computers and intractability : A guide to the theory of NP-completeness. *W.H. Freeman and Company*, ISBN 0-7167-1045-5, 1979.
7. J-L. de Siqueira N. and J-F. Puget. Explanation-based generalization of failures. *Proceedings ECAI*, pages 339–344, 1988.
8. Ulrich Junker. QUICKXPLAIN: Conflict detection for arbitrary constraint propagation algorithms. *IJCAI'01 workshop on modelling and solving problems with constraints*, pages 75–82, 2001.
9. Narendra Jussien and Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. *Artificial Intelligence*, pages 21–45, 2001.
10. J. Larrosa, P. Meseguer, and T. Schiex. Maintaining reversible DAC for Max-CSP. *Artificial Intelligence*, 107:149–163, 1999.
11. Javier Larrosa. Node and arc consistency in weighted CSP. *Proceedings AAAI*, pages 48–53, 2002.
12. E. Lawler. Combinatorial optimization: Networks and matroids. *Holt, Rinehart and Winston*, 1976.
13. M. Lemaître, G. Verfaillie, E. Bourreau, and F. Laburthe. Integrating algorithms for Weighted CSP in a constraint programming framework. *CP'2001 workshop on soft constraints, SOFT'O1, Paphos, Cyprus*, 2001.
14. R. Z. Norman and M. O. Rabin. An algorithm for minimum cover of a graph. *American Math. Soc.*, 10, 1959.
15. J-C. Régin, T. Petit, C. Bessière, and J-F. Puget. An original constraint based approach for solving over constrained prolems. *Proceedings CP*, pages 543–548, 2000.
16. J-C. Régin, T. Petit, C. Bessière, and J-F. Puget. New lower bounds of constraint violations for over constrained prolems. *Proceedings CP*, pages 332–345, 2001.
17. J-C. Régin, J-F. Puget, and T. Petit. Representation of soft constraints by hard constraints. *Proceedings JFPLC'02*, 2002.
18. T. Schiex. Arc consistency for soft constraints. *Proceedings CP*, pages 411–424, 2000.
19. T. Schiex. Une comparaison des cohérences d'arc dans les Max-CSP. *Proceedings JNPC*, pages 209–223, 2002.
20. G. Verfaillie, M. Lemaître, and T. Schiex. Russian doll search for solving constraint optimisation problems. *Proceedings AAAI*, pages 181–187, 1996.
21. R.J. Wallace. Directed arc consistency preprocessing as a strategy for maximal constraint satisfaction. *Proceedings ECAI*, pages 69–77, 1994.