Soft arc consistency revisited

M. C. Cooper

IRIT, University of Toulouse III, 31062 Toulouse, France

S. de Givry, M. Sanchez, T. Schiex*, M. Zytnicki

UBIA, UR-875, INRA, F-31320 Castanet Tolosan, France

T. Werner

Center for Machine Perception, Czech Technical University, 12135 Praha 2, Czech Republic

Abstract

The Valued Constraint Satisfaction Problem (VCSP) is a generic optimization problem defined by a network of local cost functions defined over discrete variables. It has applications in Artificial Intelligence, Operations Research, Bioinformatics and has been used to tackle optimization problems in other graphical models (including discrete Markov Random Fields and Bayesian Networks). The incremental lower bounds produced by local consistency filtering are used for pruning inside Branch and Bound search.

In this paper, we extend the notion of arc consistency by allowing fractional weights and by allowing several arc consistency operations to be applied simultaneously. Over the rationals and allowing simultaneous operations, we show that an optimal arc consistency closure can theoretically be determined in polynomial time by reduction to linear programming. This defines *Optimal Soft Arc Consistency* (OSAC).

To reach a more practical algorithm, we show that the existence of a sequence of arc consistency operations which increases the lower bound can be detected by establishing arc consistency in a classical Constraint Satisfaction Problem (CSP) derived from the original cost function network. This leads to a new soft arc consistency method, called, *Virtual Arc Consistency* which produces improved lower bounds compared with previous techniques and which can solve submodular cost functions.

These algorithms have been implemented and evaluated on a variety of problems, including two difficult frequency assignment problems which are solved to optimality for the first time. Our implementation is available in the open source `toulbar2` platform.

Keywords: valued constraint satisfaction problem, weighted constraint satisfaction problem, soft constraints, constraint optimization, local consistency, soft arc consistency, graphical model, submodularity

*Corresponding author

Email addresses: `cooper@irit.fr` (M. C. Cooper), `simon.degivry@toulouse.inra.fr` (S. de Givry), `marti.sanchez@toulouse.inra.fr` (M. Sanchez), `thomas.schiex@toulouse.inra.fr` (T. Schiex), `matthias.zytnicki@toulouse.inra.fr` (M. Zytnicki), `werner@cmp.felk.cvut.cz` (T. Werner)

1. Introduction

Graphical model processing is a central problem in AI. The optimization of the combined cost of local cost functions, central in the valued CSP framework [52], captures problems such as weighted Max-SAT, Weighted CSP or Maximum Probability Explanation in probabilistic networks. It also has applications in areas such as *resource allocation* [9], *combinatorial auctions*, *optimal planning*, and *bioinformatics* [50]. Valued constraints can be used to code both classical crisp constraints and cost functions.

Since valued constraint satisfaction is NP-hard, heuristics are required to speed up brute-force exhaustive search. By shifting weights between cost functions, soft arc consistency allows us to transform a problem in an equivalent problem. This problem reformulation can provide strong, incrementally maintainable lower bounds which are crucial for Branch and Bound search [44].

Similarly to classical arc consistency in CSPs (constraint satisfaction problems), previously-defined soft arc consistency properties are enforced by the *chaotic application* of local *soft arc consistency operations* shifting *integer costs* between different scopes, until a fixpoint is reached [19, 3]. Unlike the arc consistency closure in CSPs, this fixpoint is often not unique and may lead to different lower bounds. In this paper, we instead consider local consistencies enforced by carefully planned *sequences* of soft arc consistency operations which necessarily increase the lower bound. Since costs may need to be divided into several parts in order to be shifted in several directions, the resulting transformed problem may contain fractional costs. By allowing the introduction of rational multiples of costs, we both avoid the intractability of finding an optimal soft arc consistency closure involving only integer costs [19] and produce a strictly stronger notion of soft arc consistency.

The two new techniques presented in this paper aim at finding a reformulation of the original problem P with an optimized *constant cost term* $c_\varnothing$. This constant cost provides an explicit lower bound provided that all costs are non-negative. *Optimal soft arc consistency* (OSAC) identifies a sequence of soft arc consistency operations (shifting of costs between cost functions, of which at most one has arity greater than 1) which yields an optimal reformulation. Intermediate reformulations may contain negative costs provided all costs in the final version are non-negative. Such operations can be found in polynomial time by solving a linear program [54]. We considerably extend this result by showing that a polynomial-time algorithm exists even in the presence of crisp constraints coded by infinite costs and an upper bound coded by using an addition-with-ceiling aggregation operator.

Alternatively, we show that when a problem is not *Virtual Arc Consistent* (VAC), it is possible to find a sequence of soft arc consistency operations which improve the lower bound and are such that all intermediate problems have non-negative costs. Our iterative VAC algorithm is based on applying arc consistency in a classical CSP which has a solution if and only if P has a solution of cost $c_\varnothing$. We show that OSAC is strictly stronger than VAC. However, finding a lower bound using our VAC algorithm is much faster than establishing OSAC, and hence has potentially many more practical applications.

The idea of using classical local consistency to build lower bounds in Max-CSP or Max-SAT is not new. On Max-CSP problems, [48] used independent arc inconsistent subproblems to build a lower bound. For Max-SAT, [45] used minimal Unit Propagation inconsistent subproblems to build a lower bound. These approaches do not use problem

transformations but rely on the fact that the inconsistent subproblems identified are independent and costs can simply be summed. They lack the incrementality of soft consistency operations. In Max-SAT again, [31] used Unit Propagation inconsistency to build sequences of *integer* problem transformations but possibly strictly above the arc level, generating higher-arity weighted clauses (cost functions). OSAC and VAC remain at the arc level by allowing rational costs. It should be pointed out that our VAC algorithm is similar to the "Augmenting DAG" algorithm independently proposed by [39] for preprocessing 2-dimensional grammars, recently reviewed in [56]. Our approach is more general, in that we can treat cost functions of arbitrary arity, infinite costs and a finite upper bound.

Note that the special case of real-valued binary VCSPs over Boolean domains has been extensively studied under the name of quadratic pseudo-Boolean function optimization [7]. In the case of Boolean domains, it is well known that finding an equivalent quadratic posiform representation (i.e. an equivalent binary VCSP) with an optimal value of $c_\varnothing$ can be formulated as a linear programming problem [30] and can even be solved by finding a maximum flow in an appropriately defined network [7]. It is also worth noting that in this special case of Boolean binary VCSPs, determining whether there exists a zero-cost solution is an instance of 2SAT and hence can be completely solved in polynomial time.

The two new notions presented in this paper (optimal soft arc consistency and virtual arc consistency) can be applied to optimization problems over finite domains of arbitrary size, involving local cost functions of arbitrary arity. Crisp constraints can be coded by infinite costs and an upper bound can be coded by using an addition-with-ceiling aggregation operator. We show that the resulting arc consistency properties have attractive theoretical properties, being capable of solving different polynomial classes of weighted CSP without detecting them *a priori*. We also show their strengths and limitations on various random and real problem instances. Some of the problems considered are solved for the first time to optimality using these local consistencies.

We begin in Section 2 with the definition of a valued constraint satisfaction problem. Section 3 introduces the notion of an equivalence-preserving transformation and gives the three basic equivalence-preserving transformations that are required to establish all forms of soft arc consistency considered in this paper. In Section 4 we review previously defined notions of soft arc consistency. These definitions are necessary to define the soft arc consistency EDAC [43], with which we compare both theoretically and experimentally the new notions of soft arc consistency defined in this paper. Section 5 defines OSAC (Optimal Soft Arc Consistency) and Section 6 reports the results of experimental trials which demonstrate the potential utility of OSAC during preprocessing. The rest of the paper is devoted to Virtual Arc Consistency (VAC) which provides a practical alternative to OSAC which can be applied during search. Section 7 introduces VAC and shows formally the connection between this definition and the existence of a sequence of soft arc consistency operations which increase the lower bound. Section 8 introduces our VAC algorithm through examples while Section 9 gives the necessary subroutines in detail. Section 10 shows that certain tractable classes, including permuted submodular functions, can be directly solved by VAC. As the detailed example in Appendix A shows, our VAC algorithm may enter an infinite loop. This justifies the use of a heuristic version called VAC$_\varepsilon$. Section 11 reports the results of our experimental trials on VAC$_\varepsilon$. Finally, an alternative algorithm converging towards VAC and techniques for finding better bounds are discussed in Section 12.

2. Valued constraint satisfaction

The Constraint Satisfaction Problem (CSP) consists in finding an assignment to n finite-domain variables such that a set of constraints are satisfied. Crisp yes/no constraints in the CSP are replaced by cost functions in the Valued Constraint Satisfaction Problem (VCSP) [52]. A cost function returns a valuation (a cost, a weight or a penalty) for each combination of values for the variables in the scope of the function. Crisp constraints can still be expressed by, for example, assigning an infinite cost to inconsistent tuples. In the most general definition of a VCSP, costs lie in a valuation structure (a positive totally-ordered monoid) $\langle E, \oplus, \succcurlyeq \rangle$ where E is the set of valuations totally ordered by $\succcurlyeq$ and combined using the aggregation operator $\oplus$. In this paper we only consider integer or rational costs.

A Valued Constraint Satisfaction Problem can be seen as a set of valued constraints, which are simply cost functions placed on particular variables. Formally,

Definition 2.1 (Schiex [51]). *A* Valued Constraint Satisfaction Problem *(VCSP) is a tuple $\langle X, D, C, \Sigma \rangle$ where X is a set of n variables $X = \{1, \ldots, n\}$, each variable $i \in X$ has a domain of possible values $d_i \in D$, C is a set of cost functions and $\Sigma = \langle E, \oplus, \succcurlyeq \rangle$ is a valuation structure. Each cost function $\langle S, c_S \rangle \in C$ is defined over a tuple of variables $S \subseteq X$ (its scope) as a function c_S from the Cartesian product of the domains $d_i (i \in S)$ to E.*

Purely for notational convenience, we suppose that no two cost functions have the same scope. This allows us to identify C with the set of scopes S of cost functions c_S in the VCSP. We write c_i as a shorthand for $c_{\{i\}}$ and c_{ij} as a shorthand for $c_{\{i,j\}}$. Without loss of generality, we assume that C contains a cost function c_i for every variable $i \in X$ as well as a zero-arity constant cost function $c_\varnothing$.

Notation: For $S \subseteq X$ we denote the Cartesian product of the domains $d_i (i \in S)$ (i.e. the set of possible labellings for the variables in S) by $\ell(S)$.

Let $Z \subseteq Y \subseteq X$ with $Y = \{y_1, \ldots, y_q\}$ and $Z = \{z_1, \ldots z_p\}$. Then, given an assignment $t = (t_{y_1}, \ldots t_{y_q}) \in \ell(Y)$, $t[Z]$ denotes the sub-assignment of t to the variables in Z, i.e. $(t_{z_1}, \ldots, t_{z_p})$. If Z is a singleton $\{z_1\}$ then $t[Z]$ will also be denoted as t_{z_1} for simplicity.

The usual query on a VCSP is to find an assignment t whose valuation (i.e. total cost) is minimal.

Definition 2.2. *In a VCSP $V = \langle X, D, C, \Sigma \rangle$, the* valuation *of an assignment $t \in \ell(X)$ is defined by*

$$Val_V(t) = \bigoplus_{S \in C} \Big[c_S(t[S]) \Big]$$

To solve a VCSP we have to find an assignment $t \in \ell(X)$ with a minimum valuation.

2.1. Weighted CSP

In the VCSPs studied in this paper, the aggregation operator $\oplus$ is either the usual addition operator or the addition-with-ceiling operator $+_m$ defined as follows:

$$\forall a, b \in \{0, 1, \ldots, m\} \quad a +_m b = \min\{a + b, m\}$$

A Weighted Constraint Satisfaction Problem (WCSP) [44] is a VCSP over the valuation structure $\mathcal{S}_m = \langle \{0, 1, \ldots, m\}, +_m, \geq \rangle$ where m is a positive integer or infinity. It has been shown that the WCSP framework is sufficient to model all VCSPs over discrete valuation structures in which $\oplus$ has a partial inverse (a necessary condition for soft arc consistency operations to be applicable) [14].

When m is finite, all solutions with a cumulated cost reaching m are considered as equally and absolutely bad. This is a situation which applies at a node of a branch and bound search tree on a WCSP problem whenever the best known solution has cost m.

The Boolean valuation structure $\mathcal{S}_1 = \langle \{0, 1\}, +_1, \geq \rangle$ allows us to express only crisp constraints, with the valuation 0 representing consistency and 1 representing inconsistency. In this paper, in order to express VCSPs and CSPs in a common framework, we will often represent CSPs as VCSPs over the valuation structure $\mathcal{S}_1$.

The valuation structure $\mathcal{S}_\infty = \langle \mathbb{N} \cup \{\infty\}, +, \geq \rangle$, where $\mathbb{N}$ is the set of non-negative integers, can be embedded in the valuation structure $\overline{\mathbb{Q}}^+ = \langle \mathbb{Q}^+ \cup \{\infty\}, +, \geq \rangle$ where $\mathbb{Q}^+$ represents the set of non-negative rational numbers. Similarly, the valuation structure $\mathcal{S}_m$ can be embedded in the valuation structure $\overline{\mathbb{Q}}_m = \langle \mathbb{Q}_m \cup \{\infty\}, +_m, \geq \rangle$ where $\mathbb{Q}_m$ is the set of rational numbers α satisfying $0 \leq \alpha < m$. For clarity of presentation, we use ∞ as a synonym of m in $\overline{\mathbb{Q}}_m$, since this valuation represents complete inconsistency. We use $\oplus$ to represent the aggregation operator (which is $+$ in $\overline{\mathbb{Q}}^+$ and $+_m$ in $\overline{\mathbb{Q}}_m$). The partial inverse of the aggregation operator $\oplus$ is denoted by $\ominus$ and is defined in both $\overline{\mathbb{Q}}^+$ and $\overline{\mathbb{Q}}_m$ by $\alpha \ominus \beta = \alpha - \beta$ (for all valuations α, β such that $\infty > \alpha \geq \beta$) and $\infty \ominus \beta = \infty$ (for all valuations β).

In the remainder of the paper, we assume, unless stated otherwise, that the valuation structure Σ of the VCSP to be solved is either $\overline{\mathbb{Q}}^+$ or $\overline{\mathbb{Q}}_m$. These rational valuation structures enrich the set of available operations on costs, compared to the integer valuation structures $\mathcal{S}_\infty$ and $\mathcal{S}_m$, by allowing for the circulation of fractional weights.

3. Soft arc consistency operations

In this section we introduce the basic operations which allows us to reformulate a VCSP by shifting costs.

Definition 3.1. *Two VCSPs* $V_1 = \langle X, D, C_1, \Sigma \rangle$, $V_2 = \langle X, D, C_2, \Sigma \rangle$ *are* equivalent *if* $\forall t \in \ell(X)$, $Val_{V_1}(t) = Val_{V_2}(t)$.

Definition 3.2. *The* subproblem *of a VCSP* $\langle X, D, C, \Sigma \rangle$ *induced by* $F \subseteq C$ *is the problem VCSP(F)* $= \langle X_F, D_F, F, \Sigma \rangle$, *where* $X_F = \cup_{c_S \in F} S$ *and* $D_F = \{d_i : i \in X_F\}$.

Definition 3.3. *For a VCSP* $\langle X, D, C, \Sigma \rangle$, *an* equivalence preserving transformation *on* $F \subseteq C$ *is an operation which transforms the subproblem VCSP(F) into an equivalent VCSP.*

When F contains at most one cost function c_S *such that* $|S| > 1$, *such an equivalence-preserving transformation is called a* Soft Arc Consistency *(SAC) operation.*

Algorithm 1 gives three basic equivalence-preserving transformations which are also SAC operations [19]. Project projects weights from a cost function (on two or more variables) to a unary cost function. Extend performs the inverse operation, sending

weights from a unary cost function to a higher-order cost function. Finally UnaryProject projects weights from a unary cost function to the nullary cost function $c_\varnothing$ which is a lower bound on the value of any solution. For example, if $\forall a \in d_i$, $c_i(a) \geq \alpha$, then a call UnaryProject(i, α) increases the constant term $c_\varnothing$ by α while decreasing by α each $c_i(a)$ $(a \in d_i)$. For each of the SAC operations given in Algorithm 1, a precondition is given which guarantees that the resulting costs are non-negative.

The addition and then subtraction of the same weight β in line 10 of Extend allows us to detect certain inconsistent tuples, since this sets $c_S(t)$ to ∞ when $c_S(t) \oplus \beta = \infty$. Similarly, the addition and then subtraction of the weight $c_\varnothing$ in line 15 of UnaryProject sets $c_i(a)$ to ∞ when $c_i(a) + c_\varnothing = \infty$. Extend and UnaryProject can thus modify cost functions even when the argument $\alpha = 0$. This happens, for example, for UnaryProject in the valuation structure $\overline{\mathbb{Q}}_{10}$ if $c_i(a) = c_\varnothing = 5$ since $c_i(a)$ becomes $((5 \oplus 5) \ominus 5) \ominus 0$ which is equal to $10 = \infty$ in $\overline{\mathbb{Q}}_{10}$.

Of course, if $\oplus$ is the addition of real numbers and all costs are finite, then Extend and UnaryProject cannot modify cost functions when $\alpha = 0$. Indeed, in the case of finite costs, Extend and UnaryProject can be considerably simplified by canceling β and $c_\varnothing$ respectively.

4. Soft arc consistency techniques

In this section we briefly review previously-defined notions of soft arc consistency and, in particular, Existential Directional Arc Consistency (EDAC) [43]. EDAC was the strongest known polynomial-time achievable form of soft arc consistency before the

Algorithm 1: The basic equivalence-preserving transformations required to establish different forms of soft arc consistency.

```
1  (* Precondition: α ≤ min{c_S(t) : t ∈ ℓ(S) and t_i = a} *);
2  Procedure Project(S, i, a, α)
3      c_i(a) ← c_i(a) ⊕ α;
4      foreach (t ∈ ℓ(S) such that t_i = a) do
5          c_S(t) ← c_S(t) ⊖ α;
6  (* Precondition: α ≤ c_i(a) and |S| > 1 *);
7  Procedure Extend(i, a, S, α)
8      foreach (t ∈ ℓ(S) such that t_i = a) do
9          β ← c_∅ ⊕ (⨁_{j∈S} c_j(t_j)) ;
10         c_S(t) ← ((c_S(t) ⊕ β) ⊖ β) ⊕ α;
11     c_i(a) ← c_i(a) ⊖ α;
12 (* Precondition: α ≤ min{c_i(a) : a ∈ d_i} *);
13 Procedure UnaryProject(i, α)
14     foreach (a ∈ d_i) do
15         c_i(a) ← ((c_i(a) ⊕ c_∅) ⊖ c_∅) ⊖ α;
16     c_∅ ← c_∅ ⊕ α;
```

introduction of the two notions (OSAC and VAC) presented in this paper. Note that EDAC has only been defined in the special case of binary [43] and ternary [50] VCSPs. Recall that we assume that the valuation structure of the VCSP is either $\overline{\mathbb{Q}}^+$ or $\overline{\mathbb{Q}}_m$.

Definition 4.1 (Larrosa and Schiex [44]). *A VCSP is* node consistent *if for any variable $i \in \{1, \ldots, n\}$,*

1. $\forall a \in d_i$, $c_i(a) \oplus c_\varnothing < \infty$
2. $\exists a \in d_i$ *such that* $c_i(a) = 0$

Node consistency can be established by repeated calls to UnaryProject until convergence. We assume, for simplicity of presentation, that values a such that $c_i(a) = \infty$ are automatically deleted from d_i. Node consistency determines the maximum lower bound that can be deduced from the unary and nullary constraints; it transforms the VCSP accordingly so that this lower bound is stored explicitly in the nullary constraint $c_\varnothing$.

A VCSP is generalized arc consistent if all infinite weights have been propagated and no weights can be projected down to unary constraints. Formally,

Definition 4.2 (Cooper and Schiex [19]). *A VCSP $\langle X, D, C, \Sigma \rangle$ is* generalized arc consistent *if for all $S \in C$ such that $|S| > 1$ we have:*

1. $\forall t \in \ell(S)$, $c_S(t) = \infty$ *if* $c_\varnothing \oplus (\bigoplus_{i \in S} c_i(t_i)) \oplus c_S(t) = \infty$
2. $\forall i \in S$, $\forall a \in d_i$, $\exists t \in \ell(S)$ *such that* $t_i = a$ *and* $c_S(t) = 0$

If the VCSP is binary, then generalized arc consistency is known as (soft) arc consistency. Generalized arc consistency can be established by repeated calls to Project, together with extensions of zero weights (i.e. calls of the form Extend(_,_,_,0)) to propagate inconsistencies, until convergence.

Consider a VCSP which is node consistent and generalized arc consistent. Extending non-zero weights and re-establishing generalized arc consistency and node consistency may lead to an increase in $c_\varnothing$ [51]. One way to guarantee the convergence of such a process is to restrict the direction in which non-zero weights can be extended by placing a total ordering on the variables.

Definition 4.3 (Cooper [12]). *A binary VCSP is* directional arc consistent (DAC) *with respect to an order $<$ on the variables if for all c_{ij} such that $i < j$, $\forall a \in d_i$, $\exists b \in d_j$ such that $c_{ij}(a, b) = c_j(b) = 0$*

If for all $b \in d_j$ either $c_{ij}(a, b)$ or $c_j(b)$ is non-zero, then it is possible to increase $c_i(a)$ by transferring the non-zero costs $c_j(b)$ to c_{ij} by calls to Extend and then projecting costs from c_{ij} to $c_i(a)$. Hence establishing Directional Arc Consistency not only projects weights down to unary constraints, but also shifts weights towards variables which occur earlier in the order $<$. This tends to concentrate weights on the same variables which, after applying node consistency, tends to lead to an increase in the lower bound $c_\varnothing$.

Consider a binary VCSP with e binary cost functions and maximum domain size d. Then directional arc consistency can be established in $\mathcal{O}(ed^2)$ time [19, 44]. As in classical CSP, DAC solves tree-structured VCSP if the variable order used is built from a topological ordering of the tree.

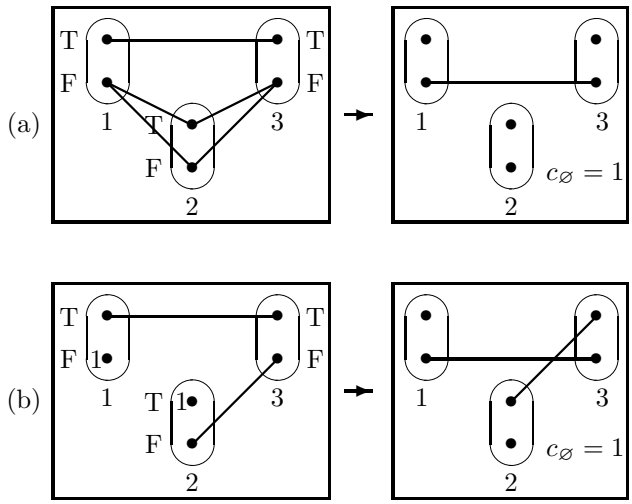

Figure 1: Examples of (a) full directional arc consistency (b) existential arc consistency.

Definition 4.4 (Cooper [12]). *A binary VCSP is* full directional arc consistent (FDAC) *with respect to an order $<$ on the variables if it is arc consistent and directional arc consistent with respect to $<$.*

Full directional arc consistency can be established in $\mathcal{O}(ed^2)$ time if the valuation structure is $\overline{\mathbb{Q}}^+$ [12] and in $\mathcal{O}(end^3)$ time if the valuation structure is $\mathcal{S}_m$ [44].

Existential arc consistency (EAC) is independent of a variable order. For each variable i in turn, EAC shifts costs to c_i if this can lead to an immediate increase in $c_\varnothing$ via UnaryProject.

Definition 4.5 (Larrosa et al. [43]). *A binary VCSP is* existential arc consistent (EAC) *if it is node consistent and if $\forall i$, $\exists a \in d_i$ such that $c_i(a) = 0$ and for all cost functions c_{ij}, $\exists b \in d_j$ such that $c_{ij}(a, b) = c_j(b) = 0$. Value a is called the* EAC support value *of variable i.*

Definition 4.6 (Larrosa et al. [43]). *A binary VCSP is* existential directional arc consistent *(EDAC) with respect to an order $<$ on the variables if it is existential arc consistent and full directional arc consistent with respect to $<$.*

Over the valuation structure $\mathcal{S}_m$, existential directional arc consistency can be established in $\mathcal{O}(ed^2 \max\{nd, m\})$ time [43].

An important difference between local consistency in CSPs and local consistency in VCSPs is that the closure under the corresponding local consistency operations is unique in CSPs but this is not, in general, the case for VCSPs [51]. For example, even for a 2-variable VCSP with domains of size 2, the arc consistency and existential arc consistency closures are not necessarily unique. Similarly, for problems with more than two variables, in general, the FDAC closure is not unique.

Figure 1(a),(b) illustrates separately the two techniques FDAC and EAC (which together form the stronger notion EDAC). In both cases, the VCSP on the left (over the

valuation structure $\overline{\mathbb{Q}}^+$) can be transformed into the equivalent VCSP on the right by establishing, respectively, FDAC and EAC. In both cases, the lower bound $c_\varnothing$ is increased from 0 to 1. Each oval represents a domain and each $\bullet$ a value. Names of values and the variable number are written outside the oval (names of values on the side and the variable number underneath). A line joining (i, a) and (j, b) represents a weight $c_{ij}(a, b) = 1$ and a value α written next to $a \in d_i$ (and inside the oval) represents $c_i(a) = \alpha$. The absence of a line or the absence of a cost next to a domain value indicates a zero cost. In Figure 1(a) the VCSP on the right is obtained by establishing FDAC with a lexicographic DAC ordering, via the following SAC operations:

1. Project($\{1, 2\}$,1,F,1), Project($\{2, 3\}$,3,F,1): this moves unit costs from the binary cost functions c_{12} and c_{23} down to c_1(F) and c_3(F) (which establishes arc consistency).
2. Extend(3,F,$\{1, 3\}$,1): we send a unit cost from c_3(F) up to the binary cost function c_{13}, so that c_{13}(T,F)= c_{13}(F,F)= 1.
3. Project($\{1, 3\}$,1,T,1): this moves a unit cost from c_{13} to c_1(T) (which establishes directional arc consistency).
4. UnaryProject(1,1): we increase the lower bound $c_\varnothing$ by replacing c_1(T)= c_1(F)= 1 by $c_\varnothing = 1$ (which establishes node consistency).

In order to establish EAC, weights are shifted towards the same variable whenever this can lead to an immediate increase in $c_\varnothing$. In Figure 1(b) the existential arc consistent VCSP on the right is obtained by shifting weights towards variable 3, via the following SAC operations:

1. Extend(2,T,$\{2, 3\}$,1), Project($\{2, 3\}$,3,F,1): we send a unit cost from c_2(T) up to c_{23} which allows us to project a unit cost from c_{23} down to c_3(F).
2. Extend(1,F,$\{1, 3\}$,1), Project($\{1, 3\}$,3,T,1): in an entirely similar manner, we send a unit cost from c_1(F) to c_3(T).
3. UnaryProject(3,1): we increase the lower bound by replacing c_3(F) = c_3(T)= 1 by $c_\varnothing = 1$.

The VCSP on the left of Figure 1(a) is EAC and the problem on the left of Figure 1(b) is FDAC, which proves that these two properties are complementary. EDAC [43], which is simply the combination of FDAC and EAC, represents the state-of-the-art soft arc consistency technique against which we must compare the new techniques defined in this paper.

EDAC tries to find a set of SAC operations which increases $c_\varnothing$, but does not perform an exhaustive search over all such sets. This is because FDAC can only extend non-zero weights in one direction, while EAC can only extend weights in the neighborhood of each variable. In the next section we will show, somewhat surprisingly, that it is possible to perform an exhaustive search over all sets of SAC operations in polynomial time.

5. Optimal soft arc consistency

An arc consistency closure of a VCSP P is any VCSP obtained from P by repeated calls to Project and UnaryProject until convergence. After each call of Project or UnaryProject, the resulting VCSP must be valid in the sense that the cost functions take values lying in the valuation structure.

Definition 5.1. *An arc consistency closure of a VCSP P is* optimal *if it has the maximum lower bound $c_\varnothing$ among all arc consistency closures of P.*

In a previous paper we proved that over a discrete valuation structure such as the non-negative integers together with infinity, the problem of finding the optimal arc consistency closure is NP-hard [19]. However, we will show in this section that extending the valuation structure to include all rationals and extending our notion of arc consistency closure allows us to determine an optimal arc consistency closure in polynomial time by a simple reduction to linear programming. This is not so much a practical proposition as a theoretical result to demonstrate that extending the valuation structure not only allows us to produce better lower bounds but also avoids intractability.

We now relax the preconditions of the soft arc consistency (SAC) operations Extend, Project and UnaryProject so that these operations can introduce negative finite costs. Over the rationals, the only restriction on costs after application of a relaxed SAC operation is that they are not $-\infty$.

Definition 5.2. *Over the valuation structure $\overline{\mathbb{Q}}^+$ (respectively $\overline{\mathbb{Q}}_m$), a* relaxed SAC operation *is a call to Extend, Project or UnaryProject such that the resulting cost functions take values in $\mathbb{Q} \cup \{\infty\}$ (respectively $\{\alpha \in \mathbb{Q} : \alpha < m\} \cup \{\infty\}$).*

If we apply a sequence of relaxed SAC operations to produce a VCSP P, then in order to be able to use $c_\varnothing$ as a lower bound, we must ensure that the costs in P are all non-negative (although intermediate problems may contain negative finite costs).

Definition 5.3. *Given a VCSP P over the valuation structure $\overline{\mathbb{Q}}^+$ or $\overline{\mathbb{Q}}_m$, a* SAC transformation *is a sequence of relaxed SAC operations which transforms P into a valid VCSP (i.e. such that all cost functions take values in the valuation structure).*

Definition 5.4. *A VCSP P over the valuation structure $\overline{\mathbb{Q}}^+$ or $\overline{\mathbb{Q}}_m$ is* optimal soft arc consistent (OSAC) *if no SAC transformation applied to P increases $c_\varnothing$.*

Over the valuation structure $\overline{\mathbb{Q}}^+$, a SAC transformation involving the shifting of only *finite* costs can be considered as a *set* of relaxed SAC operations: the order in which operations are applied is of no importance since, in this case, the operations Extend, Project and UnaryProject all commute.

Affane & Bennaceur [1] split integer costs by propagating a fraction w_{ij} of the binary cost function c_{ij} towards variable i and a fraction $1 - w_{ij}$ towards variable j (where $0 \leq w_{ij} \leq 1$) and suggested determining the optimal values of the weights w_{ij}. In a more recent paper, Bennaceur & Osmani [4] suggested introducing different weights w_{iajb} for each pair of domain values $(a, b) \in d_i \times d_j$. As we show in this paper, it turns out that assigning a different weight to each triple (i, j, a), where $a \in d_i$, allows us to find optimal weights in polynomial time.

Theorem 5.5. *Let P be a VCSP over the valuation structure $\overline{\mathbb{Q}}^+$ such that the arity of cost functions in P is bounded by a constant. It is possible to find in polynomial time a SAC transformation of P which maximizes the lower bound $c_\varnothing$ and hence establishes optimal soft arc consistency.*

$$\text{Maximize } \sum_{i=1}^{n} u_i \quad \text{subject to}$$

$$\forall i \in \{1,\ldots,n\},\ \forall a \in d_i, \quad c_i(a) - u_i + \sum_{(S \in C) \wedge (i \in S)} p_i^S(a) \ \geq \ 0$$

$$\forall S \in C \text{ such that } |S| > 1,\ \forall t \in \ell(S), \quad c_S(t) - \sum_{i \in S} p_i^S(t_i) \ \geq \ 0$$

Figure 2: The linear program to establish optimal soft arc consistency (after propagation of infinite weights).

Proof: Firstly, as in [12], we can assume that all infinite costs have been propagated using a standard generalized arc consistency algorithm [46]. Note that we assume that $c_S(t)$ has been set to ∞ if $c_i(t_i) = \infty$ for some $i \in S$. At this point no more infinite costs can be propagated in the VCSP by the operations Extend, Project or UnaryProject.

We then want to determine the set of finite SAC operations which when applied simultaneously maximizes the increase in $c_\varnothing$. For each $S \in C$ such that $|S| > 1$ and for each $i \in S$, let $p_i^S(a)$ be the sum of the weights projected from c_S to $c_i(a)$ minus the sum of the weights extended from $c_i(a)$ to c_S. Let u_i be the sum of the weights projected (by UnaryProject) from c_i to $c_\varnothing$. Thus the problem is to maximize $\sum_i u_i$ such that the resulting cost functions take on non-negative values. This is equivalent to the linear program given in Figure 2. We can simply ignore the inequalities for which $c_i(a) = \infty$ or $c_S(t) = \infty$ since they are necessarily satisfied. The remaining inequalities define a standard linear programming problem with $\mathcal{O}(ed + n)$ variables (if e is the number of cost functions, n the number of variables and d the maximum domain size) which can be solved in polynomial time [33]. Since no infinite weights can be propagated and no further propagation of finite weights can increase $c_\varnothing$, the resulting VCSP is optimal soft arc consistent. ∎

Karmarkar's interior-point algorithm for linear programming has $\mathcal{O}(N^{3.5}L)$ time complexity, where N is the number of variables and L the number of bits required to encode the problem [33]. Under the reasonable assumption that $e \geq n$, the number of variables N in the linear program in Figure 2 is $\mathcal{O}(ed)$ and the number of bits L required to code it is $\mathcal{O}(ed^r \log M)$, where r is the maximum arity of cost functions and M the maximum finite cost. Therefore this linear program can be solved in $\mathcal{O}(e^{4.5} d^{(r+3.5)} \log M)$ time.

A weaker version of Theorem 5.5, limited to 3-variable subproblems, is the basis of the algorithm to establish 3-cyclic consistency [13]. Note that the linear program in Figure 2 is the dual of the linear relaxation of the 01-integer program defined in thesis [38, 36]. Both the primal and dual linear programs were first studied in [54].

It is important to note that there is a difference between SAC transformations (which are sequences of *relaxed* SAC operations) and sequences of SAC operations: the former are stronger due to the fact that intermediate problems can contain negative costs. When only finite costs are shifted in $\overline{\mathbb{Q}}^+$, a SAC transformation is equivalent to a set of SAC operations. Several SAC operations applied simultaneously can produce a valid VCSP even when no individual SAC operation can be applied. As an example, consider the

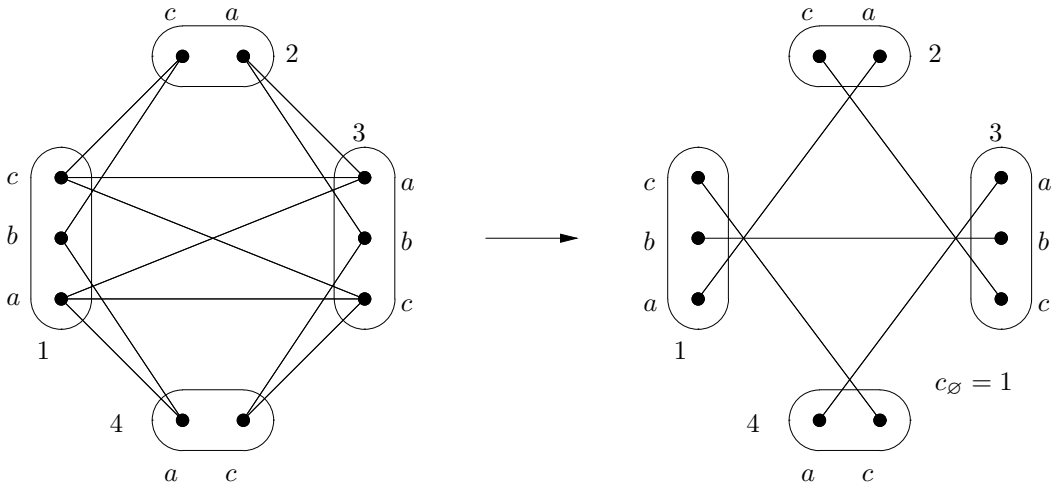

Figure 3: No sequence of SAC operations can be applied to the VCSP in (a), but a set of simultaneous SAC operations transforms it into the VCSP in (b).

binary VCSP P over domains $d_1 = d_3 = \{a, b, c\}$, $d_2 = d_4 = \{a, b\}$ and valuation structure $\overline{\mathbb{Q}}^+$ illustrated in Figure 3(a). All unary costs are equal to zero. All edges represent a unit cost. $c_\varnothing$ is assumed to be zero. P is node consistent and arc consistent, and hence no cost $\alpha > 0$ can be projected (or unary-projected) without introducing a negative cost. Also, since all unary costs are equal to zero, no cost $\alpha > 0$ can be extended without introducing a negative cost. It follows that no SAC operation (Extend, Project or UnaryProject) can transform P into a valid VCSP. This implies that no sequence of SAC operations can modify P, and, in particular, that P is EDAC.

However, we may perform the following relaxed SAC operations:

1. Extend$(2, c, \{2, 3\}, 1)$: we move a (virtual) cost of 1 from $c_2(c)$ to three pairs inside c_{23}, namely $c_{23}(c, a)$, $c_{23}(c, b)$ and $c_{23}(c, c)$. This introduces a negative cost $c_2(c) = -1$.
2. Project$(\{2, 3\}, 3, a, 1))$, Project$(\{2, 3\}, 3, b, 1)$: this moves two unit costs to $c_3(a)$ and $c_3(b)$.
3. Extend$(3, a, \{3, 4\}, 1)$, Extend$(3, b, \{3, 1\}, 1)$: these two unit costs are moved inside c_{34} and c_{31} respectively.
4. Project$(\{3, 4\}, 4, c, 1)$: this moves a unit cost of 1 to $c_4(c)$.
5. Project$(\{3, 1\}, 1, a, 1)$, Project$(\{3, 1\}, 1, c, 1)$: this moves two unit costs of 1 to $c_1(c)$ and $c_1(a)$.
6. Extend$(1, a, \{1, 2\}, 1)$, Project$(\{1, 2\}, 2, c, 1)$: we reimburse our initial loan on value $c_2(c)$.
7. Extend$(1, c, \{1, 4\}, 1)$, Project$(\{1, 4\}, 4, a, 1)$: we send a unit cost to value $c_4(a)$.
8. Finally, the application of UnaryProject$(4, 1)$ yields the problem on the right of Figure 3 with a lower bound $c_\varnothing = 1$.

If the relaxed SAC operations are applied in the above order, then the intermediate problems between steps 1 and 6 have the invalid negative weight $c_2(c) = -1$, but in the final problem all weights are non-negative. Since all costs movements are finite this sequence of relaxed SAC operations is equivalent to a set of simultaneous relaxed SAC operations. This set of operations corresponds to a solution of the linear programming problem given in Figure 2 in which $p^{23}_{2c} = p^{34}_{3a} = p^{31}_{3b} = p^{12}_{1a} = p^{14}_{1c} = -1$ and $p^{23}_{3a} = p^{23}_{3b} = p^{34}_{4c} = p^{31}_{1a} = p^{31}_{1c} = p^{12}_{2c} = p^{14}_{4a} = u_4 = 1$ (all other variables being equal to zero).

We have seen that applying a set of SAC operations simultaneously leads to a stronger notion of consistency than applying a set of SAC operations sequentially. An obvious question is whether another even stronger form of consistency exists which transforms a VCSP into an equivalent VCSP.

Definition 5.6. *A VCSP P is* in-scope $c_\varnothing$-irreducible *if there is no equivalent VCSP Q with the same set of cost function scopes as P and such that $c^Q_\varnothing > c^P_\varnothing$ (where $c^P_\varnothing$, $c^Q_\varnothing$ are the nullary cost functions in P, Q).*

The following theorem is a direct consequence of Lemma 5.2 in [13] (in which it was proved for any finitely-bounded strictly monotonic valuation structure, hence in $\mathbb{Q}^+$).

Theorem 5.7. *Let P be a binary VCSP with all unary and binary cost functions and in which cost functions take values in $\mathbb{Q}^+$ (and hence all costs are finite). If no SAC transformation applied to P produces a VCSP Q with $c^Q_\varnothing > c^P_\varnothing$, then P is in-scope $c_\varnothing$-irreducible.*

Thus, when all costs are finite rational numbers, the linear programming approach can be used to establish in-scope $c_\varnothing$-irreducibility in binary VCSPs. This is unfortunately not the case if infinite costs can occur. Consider, for example, the graph-coloring problem on a triangle with two colors, expressed as a VCSP with costs in $\{0, \infty\}$. The problem is clearly inconsistent and hence equivalent to a VCSP with a single cost function $c_\varnothing = \infty$, but no SAC transformation can be applied to this VCSP to increase $c_\varnothing$.

We conclude this section by showing that optimal soft arc consistency can also be established in polynomial time over the valuation structure $\overline{\mathbb{Q}}_m$. In this case, however, we may have to solve many linear programs.

Theorem 5.8. *Let $P = \langle X, D, C, \overline{\mathbb{Q}}_m\rangle$ be a VCSP such that the arity of cost functions in P is bounded by a constant r. Then it is possible to find in polynomial time an optimal soft arc consistent VCSP equivalent to P.*

Proof: In the following, we use S to represent any constraint scope such that $|S| \geq 1$. For each $\langle S, c_S\rangle \in C$ and for each $t \in \ell(S)$, let $P_{S,t}$ denote the VCSP which is identical to P except that the domain of each variable $i \in S$ has been reduced to a singleton consisting of the value t_i assigned by the tuple t to variable i and the valuation structure of $P_{S,t}$ is $\overline{\mathbb{Q}}^+$. By performing operations in the valuation structure $\overline{\mathbb{Q}}^+$, the upper bound m is temporarily ignored. If establishing OSAC in $P_{S,t}$ produces a lower bound $c_\varnothing \geq m$, then in the original valuation structure $\overline{\mathbb{Q}}_m$ this represents an inconsistency. This means that setting $c_S(t) = \infty$ in P produces a VCSP which is equivalent to the original VCSP P. Denote by $\text{OSAC}_m(S, t)$ the establishment of OSAC in $P_{S,t}$ and the setting of $c_S(t)$ to ∞ in P if the resulting lower bound in the transformed $P_{S,t}$ is greater than or equal to m.

Now consider the algorithm OSAC_m which simply repeatedly applies $\text{OSAC}_m(S,t)$ for all constraint scopes S and all tuples $t \in \ell(S)$ until convergence. Denote by P^∞ the VCSP which results when OSAC_m is applied to P. The complexity of OSAC_m is bounded by the time complexity of $(ed^r)^2$ times the time complexity of the linear program in Figure 2, where r is the maximum arity of cost functions in P.

We now only need to establish OSAC one more time in P^∞, considered as a VCSP over the valuation structure $\overline{\mathbb{Q}}^+$. Let σ denote the corresponding sequence of relaxed SAC operations which establish OSAC in P^∞, and let P^* denote the VCSP which results when this sequence of operations σ is applied to P^∞. Clearly P^* is equivalent to P.

It remains to show that P^* is optimal soft arc consistent over $\overline{\mathbb{Q}}_m$. To prove this, it is sufficient to show that establishing OSAC over $\overline{\mathbb{Q}}_m$ cannot introduce new infinite costs. Suppose, for a contradiction, that there exists a sequence σ' of relaxed SAC operations in $\overline{\mathbb{Q}}_m$ which when applied to P^* sets some cost $c_S(t)$ to ∞. Without loss of generality, we can assume that σ' is minimal, so that $c_S(t)$ is the first cost set to ∞ by σ'. Then the combined sequence σ, σ' applied to $P^\infty_{S,t}$ sets $c_S(t)$ to a value $\rho \geq m$. $P^\infty_{S,t}$ represents the VCSP which is identical to P^∞ except that the domain of each variable $i \in S$ has been reduced to a singleton and the valuation structure is $\overline{\mathbb{Q}}^+$. By adding at most one Project and one UnaryProject (to transfer this cost ρ from $c_S(t)$ to $c_\varnothing$), the sequence σ, σ' can be expanded so that it sets $c_\varnothing$ to $\rho \geq m$ in $P^\infty_{S,t}$. But, by the definition of P^∞ no such sequence can exist. Hence no sequence of relaxed SAC operations can introduce infinite costs in P^*, and therefore, by the definition of P^*, no sequence of relaxed SAC operations can increase $c_\varnothing$ in P^*. ∎

6. Experimental trials of OSAC

In this section, the linear programming problem defined by OSAC was solved using ILOG CPLEX version 9.1.3 (using the barrier algorithm). We first evaluate the strength and the computational cost of the lower bounds produced after a direct application of OSAC on different problems.

6.1. Evaluation of OSAC lower bounds

Random MaxCSP. The first set of instances processed are random Max-CSP instances created by the *random_vcsp* generator[1] using the usual four parameter model (n: number of variables, d: size of domains, e: number of randomly-chosen binary constraints, and t: percentage of randomly-chosen forbidden tuples inside each constraint). The aim is to find an assignment that minimizes the number of violated constraints. Four different categories of problems with domain size 10 were generated following the same protocol as in [44]: sparse loose (SL, 40 variables), sparse tight (ST, 25 variables), dense loose (DL, 30 variables) and dense tight (DT, 25 variables). These instances are available in the Cost Function Library archive at https://mulcyber.toulouse.inra.fr/projects/costfunctionlib.

Samples have 50 instances. Table 1 shows respectively the average optimum value, the average values of the EDAC lower bound and the average value of the OSAC lower

[1] http://www.inra.fr/mia/ftp/T/VCSP/src/random_vcsp.c

	SL	ST	DL	DT
Optimum	2.84	19.68	2.22	29.62
EDAC lb.	0	4.26	0	9.96
OSAC lb.	0	12.30	0	19.80

Table 1: Results of preprocessing random WCSPs by OSAC and EDAC. For each category of problems (S: Sparse ($e = 2.5n$), D: Dense ($e = \frac{n(n-1)}{8}$), L: Loose, T: Tight), the average cost of an optimal solution and the average lower bound $c_\varnothing$ produced by EDAC and OSAC is reported.

bound. On loose problems, OSAC and EDAC leave the lower bound unchanged. This shows that higher level local consistencies are required here. However for tight problems, OSAC is extremely powerful, providing lower bounds which are sometime three times better than EDAC bounds.

Frequency assignment. The second set of benchmarks is defined by instances of the Radio Link Frequency Assignment Problem of the CELAR [9][2]. This problem consists in assigning frequencies to a set of radio links in such a way that all the links may operate together without noticeable interference. Some RLFAP instances can be naturally cast as binary WCSPs.

These problems have been extensively studied and their current state is reported on the FAP web site at http://www.zib.de/fap/problems/CALMA. Despite extensive studies, the gap between the best upper bound (computed by local search methods) and the best lower bound (computed by exponential time algorithms) is not closed except for instance `scen06`, and more recently instance `scen07` [49]. The problems considered here are the `scen0{6,7,8}reduc.wcsp` and the `graph1{1,3}reducmore.wcsp` instances which have already been through different strong preprocessing (see the Benchmarks section in [22]). In order to differenciate these from the equivalent full unprocessed instances, a subscript $_r$ is used to identify them in the following tables.

As Table 2 shows, OSAC offers substantial improvements over EDAC, especially on the `graph11` and `graph13` instances. For these instances, OSAC reduces the optimality gap $\frac{ub-lb}{ub}$ to 4% and 3% respectively. The polynomial time lower bounds obtained by OSAC are actually close to the best known (exponential time) lower bounds.

6.2. OSAC preprocessing before tree search

To actually assess the practical interest of OSAC we tried to solve problems using a tree-search algorithm maintaining EDAC after OSAC preprocessing.

Tight random MaxCSP. The first experiment was performed on problems where OSAC preprocessing seems effective: random tight MaxCSPs. The difficulty here lies in the fact that CPLEX is a floating point solver while the open source WCSP solver used (`toolbar` version 3.0 in C language, section Algorithms in [22], extended with OSAC) deals with integer costs. To address this issue, we use "fixed point" costs: for all WCSPs considered, we first multiply all costs by a large integer constant $\lambda = 1000$, and then solve the linear

[2]We would like to thank the french Centre Electronique de l'Armement for making these instances available.

	scen06$_r$	scen07$_r$	scen08$_r$	graph11$_r$	graph13$_r$
Total # of values	3196	4824	14194	5747	13153
Best known ub	3389	343592	262	3080	10110
Best known lb	3389	343592	216	3016	9925
Best lb cpu-time	221"	386035"	13452"	74113"	23211"
EDAC lb	0	10000	6	2710	8722
OSAC lb	3.5	31453.1	48	2957	9797.5
EDAC cpu-time	<1"	<1"	<1"	<1"	<1"
OSAC cpu-time	621"	3530"	6718"	492"	6254"

Table 2: Radio link frequency assignment problems: for each problem, the problem size (number of values), the best known upper bound, the best known lower bound and the corresponding cpu-time needed to produce it. These cpu-times are taken from [49] using a 2.66 GHz Intel Xeon with 32 GB (`scen06`$_r$,`scen07`$_r$), from [21] on a SUN UltraSparc 10 300MHz workstation (`scen08`), and from [37] on a DEC 2100 A500MP workstation (`graph11`$_r$,`graph13`$_r$). These are followed by the lower bounds ($c_\varnothing$) produced by EDAC and OSAC, as well as the cpu-time needed to enforce EDAC and OSAC using CPLEX on a 3 GHz Intel Xeon with 2 GB.

programming problem defined by OSAC using integer variables (instead of floating point). The first integer solution found is used. The resulting problem has integer costs and can be tackled by `toolbar`[3]. This means that we shift from a polynomial problem to an NP-hard one. In practice, we found that the problems obtained have a very good linear continuous relaxation and are not too expensive to solve as integer problems (up to 3.5 slower than LP relaxation in the following experiments). Using a polytime rational LP solver would allow to recover a polynomial time bound.

Figure 4 reports cpu-time (top) and size of the tree search (bottom) for dense tight problems of increasing size. The time limit was set to 1800 seconds.

Clearly, for small problems (with less than 29 variables), OSAC is more expensive than the resolution itself. As the problem size increases, OSAC becomes effective and for 33 variables, it divides the total cpu-time by roughly 2. The number of nodes explored in both cases shows the strength of OSAC used as a preprocessing technique (remember that EDAC is maintained during search).

OSAC and DAC ordering. The strength of OSAC compared to local consistencies such as directional arc consistency (DAC) is that is does not require an initial variable ordering. Indeed, DAC directly solves tree-structured problems but only if the variable ordering used for DAC enforcing is a topological ordering of the tree. To evaluate to what extent OSAC can overcome these limitations, we used random problems structured as binary clique trees as in [23]. Each clique contains 6 variables with domain size 5, each sharing 2 variables with its parent clique. The overall tree height is 4, leading to a total number of 62 variables, with a graph density of 11%.

On these clique-tree problems, two DAC orderings were used. One is compatible with a topological ordering of the binary tree (and should give good lower bounds),

[3]The code of `toolbar` has been modified accordingly: if a solution of cost 2λ is known for example and if the current lower bound is 1.1λ then backtrack occurs since all global costs in the original problem are integer and the first integer above 1.1 is 2, the upper bound.

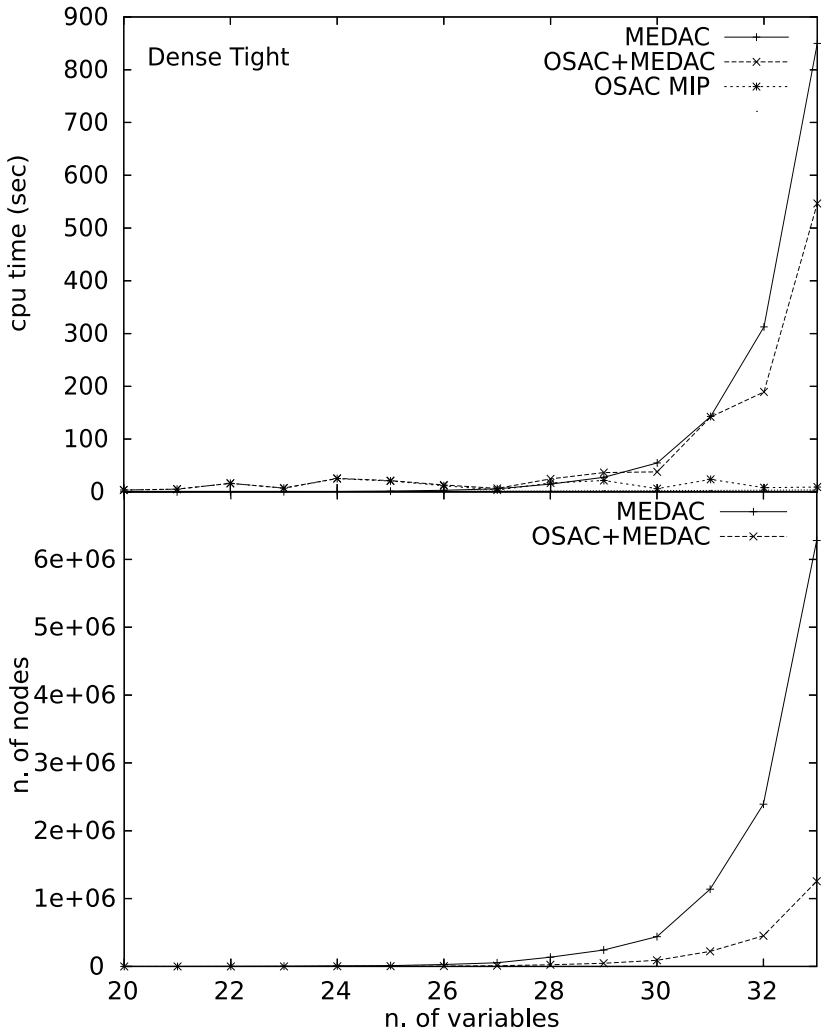

Figure 4: Experimental evaluation of OSAC as a preprocessing technique on random dense tight problems. Three cpu-times are reported: (1) OSAC MIP: time taken to get the first integer solution, (2) MEDAC: time taken to solve the original problem by maintaining EDAC [43] in `toolbar` with default parameters and a good initial upper bound, (3) OSAC+MEDAC is the sum of OSAC MIP with the time needed by MEDAC to solve the OSAC problem (with the same default parameters and upper bound).

the inverse order can be considered as pathological. The cpu-times for MEDAC alone (default `toolbar` parameters and a good initial upper bound) and OSAC+MEDAC (as previously) are shown in each case in Figure 5. Clearly, OSAC leads to drastic (up to 20 fold) improvements when a bad DAC ordering is used. Being used just during preprocessing, it does not totally compensate for the bad ordering. But, even when a good DAC ordering is used, OSAC gives impressive (up to 4 fold) speedups, especially on tight problems.

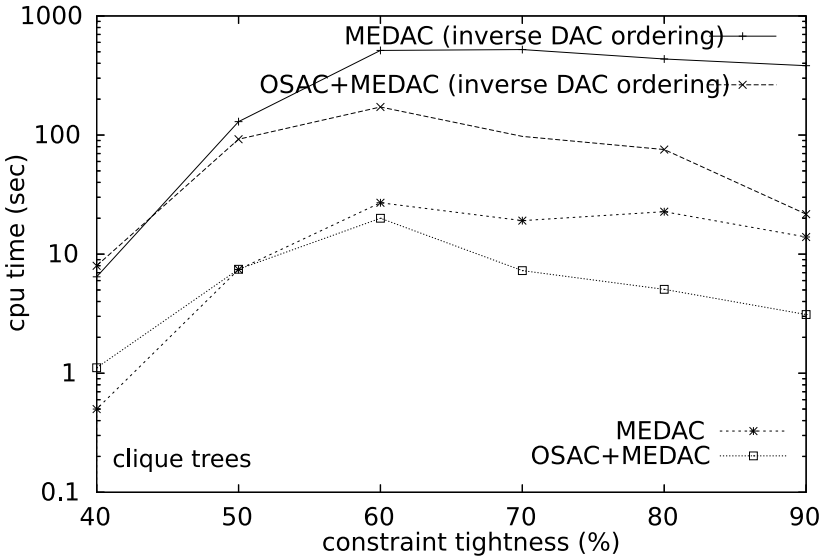

Figure 5: Experimental evaluation of OSAC as a preprocessing technique on random problems with a binary clique tree structure. The figure uses a logarithmic scale for cpu-time for different constraint tightnesses (below 40%, problems are satisfiable).

Finally, we tried to solve the challenging open CELAR instances after OSAC preprocessing. Despite the strength of OSAC, all problems remained unsolvable.

7. Virtual arc consistency

Although OSAC is optimal in terms of strength of the induced lower bound, the associated linear program is often too large for OSAC to be beneficial in terms of resolution speed. However, OSAC showed that instead of the chaotic application of integer equivalence-preserving transformations, the planning of a set of rational SAC operations may be extremely beneficial. In this section, we introduce Virtual Arc Consistency (VAC) which plans *sequences* of rational SAC operations which increase the lower bound $c_\varnothing$. These sequences are found by means of classical (generalized) arc consistency in a CSP Bool(P) derived from the VCSP P. Over the valuation structures $\overline{\mathbb{Q}}^+$ or $\overline{\mathbb{Q}}_m$ (and under the reasonable assumption that $c_\varnothing \neq \infty$), the relations in Bool(P) contain exactly those tuples which have zero cost in P. Bool(P) is a CSP whose solutions are exactly those n-tuples x such that $Val_P(x) = c_\varnothing$.

Definition 7.1. *If $P = \langle X, D, C, \Sigma\rangle$ is a VCSP over the valuation structure $\overline{\mathbb{Q}}^+$ or $\overline{\mathbb{Q}}_m$, then Bool(P) is the classical CSP $\langle X, D, \overline{C}\rangle$ where, for all scopes $S \neq \varnothing$, $\langle S, R_S\rangle \in \overline{C}$ if and only if $\exists\langle S, c_S\rangle \in C$, where R_S is the relation defined by $\forall x \in \ell(S)$ $(t \in R_S \Leftrightarrow c_S(t) = 0)$.*

We say that a CSP is *empty* if at least one of its domains is the empty set.

Definition 7.2. *A VCSP P is* virtual arc consistent *if the (generalized) arc consistency closure of the CSP Bool(P) is non-empty.*

The following theorem shows that if establishing arc consistency in Bool(P) detects an inconsistency, then it is possible to increase $c_\varnothing$ by a sequence of soft arc consistency operations.

Theorem 7.3. *Let P be a VCSP over the valuation structure $\overline{\mathbb{Q}}^+$ or $\overline{\mathbb{Q}}_m$ such that $c_\varnothing < \infty$. Then there exists a sequence of soft arc consistency operations which when applied to P leads to an increase in $c_\varnothing$ if and only if the arc consistency closure of Bool(P) is empty.*

Proof: Throughout this proof we consider Bool(P) as a VCSP over the Boolean valuation structure $\mathcal{S}_1 = \langle \{0,1\}, +_1, \geq \rangle$. To differentiate the cost functions in Bool(P) from those in P, we denote the cost functions of scope S in P and Bool(P) by c_S and $\overline{c}_S$, respectively.

$\Rightarrow$: Let $O_1, \ldots, O_k$ be a sequence of soft arc consistency operations (Project, Extend or UnaryProject) in P which produce an equivalent VCSP in which $c_\varnothing$ has increased. We assume, without loss of generality, that O_k is the UnaryProject operation which increases $c_\varnothing$. For each $i = 1, \ldots, k$, if O_i projects or extends a weight α, let O'_i be the corresponding operation in Bool(P) except that α is replaced by $\overline{\alpha}$ where

$$\overline{\alpha} = \begin{cases} 1 & \text{if } \alpha > 0 \\ 0 & \text{if } \alpha = 0 \end{cases}$$

For example, if O_i is Project $(S, j, a, 0.5)$ in P, then O'_i is Project $(S, j, a, 1)$ in Bool(P); if O_i is Extend $(j, a, S, 0)$, then O'_i is Extend $(j, a, S, 0)$. Let Bool$(P)_i$ represent the result of applying $O'_1, \ldots, O'_i$ to Bool(P) and P_i represent the result of applying $O_1, \ldots, O_i$ to P. The sequence $O'_1, \ldots, O'_k$ never decreases a cost function $\overline{c}_S$ (since $1 \ominus 1 = 1$ in $\mathcal{S}_1$). By a simple inductive argument we can see that, for $|S| \geq 1$ and $i < k$, $\overline{c}_S(t) = 1$ in Bool$(P)_i$ whenever $c_S(t) > 0$ in P_i (and hence the preconditions of O'_{i+1} are satisfied). If O_i is a projection which assigns a non-zero weight to $c_j(a)$, then $\overline{c}_j(a) = 1$ after applying O'_i. If O_i is an extension which assigns a non-zero weight to $c_S(t)$, then $\overline{c}_S(t) = 1$ after applying O'_i. Finally, since O_t is a unary projection which increases $c_\varnothing$ by some weight $\alpha > 0$, it follows that O'_k sets $c_\varnothing$ to $\overline{\alpha} = 1$.

$\Leftarrow$: Suppose that there exists a sequence of arc consistency operations which lead to a domain wipe-out in Bool(P). We can assume, without loss of generality, that no two of these operations are identical since applying the same arc consistency operation twice is redundant in CSPs. There is a corresponding sequence $O_1, \ldots, O_k$ of soft arc consistency operations (Project, Extend or UnaryProject) in Bool(P), viewed as a VCSP over the Boolean valuation structure $\mathcal{S}_1$, which set $\overline{c}_\varnothing$ to 1 in Bool(P). We assume, without loss of generality, that O_k is the UnaryProject operation which sets $\overline{c}_\varnothing$ to 1 in Bool(P).

Let δ be the minimum non-zero weight occurring in P, i.e. $\delta = \min\{c_S(t) : (\langle S, c_S \rangle \in C) \wedge (t \in \ell(S)) \wedge (c_S(t) > 0)\}$. For $i = 1, \ldots, k$, let O'_i be the soft arc consistency operation in P which is identical to O_i except that the weight being projected or extended is $\delta/2^i$. For example, if O_i is Project $(S, i, a, 1)$ in Bool(P), then O'_i is Project $(S, i, a, \delta/2^i)$ in P. We divide by two each time to ensure that strictly positive costs remain strictly positive.

Let Bool$(P)_i$ represent the result of applying $O_1, \ldots, O_i$ to Bool(P) and P_i represent the result of applying $O'_1, \ldots, O'_i$ to P. By a simple inductive argument, the minimum non-zero cost in P_i is at least $\delta/2^i$. Since the operations O_i and O'_i are identical except for the weight being projected or extended, Bool(P_i) is identical to Bool$(P)_i$ for $i < k$ (and hence the preconditions of O'_{i+1} are satisfied). It follows that O'_k necessarily increases $c_\varnothing$ by $\delta/2^k > 0$ in P since O_k sets $c_\varnothing$ to 1 in Bool(P). ∎

It may not seem that increasing $c_\varnothing$ by a very small amount (such as the increase of $\delta/2^k$ demonstrated in the proof of Theorem 7.3) is worthwhile. However, if the original weights in P were all integers, then $c_\varnothing > 0$ actually implies that $Val_P(x) \geq 1$, for all x, thus allowing us to increase the lower bound used by branch and bound by 1. In this case the lower bound is strictly greater than $c_\varnothing$.

VAC is easily shown to be stronger than Existential Arc Consistency [43]. Indeed, EAC can be seen as applying virtual arc consistency but limited to a single iteration of arc consistency in Bool(P). In EAC, weights are transferred virtually to each variable from all its neighbors; if a unary projection with a non-zero weight is possible, then we trace back and actually perform the necessary soft arc consistency operations. Thus EAC avoids the problem of fractional weights by applying only a weak form of virtual arc consistency.

Corollary 7.4. *If a VCSP P over the valuation structure $\overline{\mathbb{Q}}^+$ or $\overline{\mathbb{Q}}_m$ is virtual arc consistent, then establishing EDAC cannot increase the lower bound $c_\varnothing$ in P.*

Proof: EDAC is established by applying a sequence of SAC operations [43], but by Theorem 7.3, no sequence of SAC operations can increase $c_\varnothing$ in P. ∎

Corollary 7.5. *If a VCSP P over the valuation structure $\overline{\mathbb{Q}}^+$ or $\overline{\mathbb{Q}}_m$ is optimal arc consistent, then P is also virtual arc consistent.*

Proof: Since P is optimal soft arc consistency, no sequence of relaxed SAC operations increases $c_\varnothing$. Hence no sequence of SAC operations increases $c_\varnothing$ and therefore, by Theorem 7.3, P is virtual arc consistent. ∎

8. Increasing the lower bound using VAC

We know by Theorem 5.5 and Theorem 5.8 that we can establish OSAC (and hence VAC) in polynomial time. Unfortunately, the time complexity of OSAC limits its use to preprocessing. In this section we introduce a low-order polynomial-time algorithm which determines a sequence of SAC operations which necessarily increases $c_\varnothing$ if such a sequence exists. By Theorem 7.3, a VCSP is virtual arc consistent if and only if no such sequence exists. VAC is strictly weaker than OSAC due to the fact that, in the case of VAC, intermediate problems must have non-negative cost functions.

In soft arc consistency [19] we often have a choice as to which direction we project or extend weights. Note that the name virtual arc consistency comes from the fact that instead of making such choices, we effectively project or extend simultaneously virtual weights in all possible directions, by establishing arc consistency in Bool(P). One iteration of our VAC algorithm consists of three phases:

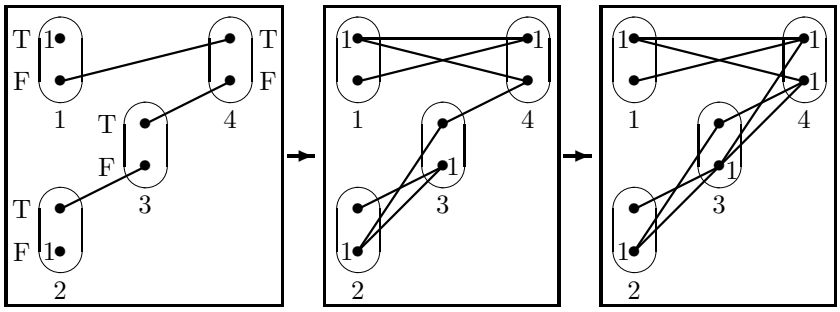

Figure 6: A VCSP P (leftmost box) which is EDAC but not virtual arc consistent, as shown by establishing arc consistency in Bool(P).

1. Establish arc consistency in Bool(P), stopping if domain wipe-out occurs (i.e. as soon as the domain of some variable i becomes empty). If Bool(P) is arc consistent, then quit, since P is virtual arc consistent.
2. Suppose that domain wipe-out occurred at variable i in Bool(P), and that σ is the sequence of arc consistency operations which led to this domain wipe-out. Find a minimal subsequence of σ which provokes this domain wipe-out by tracing back from variable i only retaining those arc consistency operations which are strictly necessary.
   Convert the this minimal sequence of arc consistency operations in Bool(P) into a corresponding sequence σ' of soft arc consistency operations in P which produces the maximum increase λ in $c_\varnothing$ while keeping all costs non-negative.
3. Apply the sequence σ' of operations to P.

Consider the following instance P of Max-SAT: $\neg X_1$; $X_1 \vee \neg X_4$; $\neg X_3 \vee X_4$; X_2; $\neg X_2 \vee X_3$. This VCSP is illustrated in the leftmost box of Figure 6. A line joining (i, a) and (j, b) represents a cost $c_{ij}(a, b) = 1$. Unary costs $c_i(a) = 1$ are noted next to the domain element (i, a). Note that P is existential directional arc consistent (EDAC). However, it is not virtual arc consistent, since establishing arc consistency in Bool(P) leads to an inconsistency. The leftmost box in Figure 6 also represents Bool(P) where now weights are interpreted as being Boolean values. For ease of comparison with the corresponding VCSP P, in figures we will always represent the CSP Bool(P) as a VCSP over the Boolean valuation structure $S_1 = \langle\{0, 1\}, +_1, \geq\rangle$ in which $0 < 1$ and $1 +_1 1 = 1$ (i.e. 0 represents consistency, 1 inconsistency and $+_1$ is the idempotent plus operator in the classical 2-element Boolean algebra). In other words, in Bool(P) a line between (i, a) and (j, b) represents the fact that (a, b) is *not* a consistent assignment to variables (i, j) and a unary cost of 1 next to (i, a) represents the fact that a is not a consistent assignment to variable i. In this representation of Bool(P), propagating inconsistencies, as illustrated in the middle and right-hand boxes of Figure 6, means adding lines and setting unary costs to 1. For example, the inconsistency $c_1(\mathrm{T}) = 1$ is propagated to the binary cost function c_{12} ($c_{12}(\mathrm{T},\mathrm{T}) = c_{12}(\mathrm{T},\mathrm{F}) = 1$) and then to value T in d_4 ($c_4(\mathrm{T}) = 1$), as shown in the middle box in Figure 6. A domain wipe-out occurs at variable 4 in the right-hand box of Figure 6: $c_4(\mathrm{T}) = c_4(\mathrm{F}) = 1$ meaning that both elements of d_4 are inconsistent.

During establishment of arc consistency in Bool(P), the reason for each inconsistency

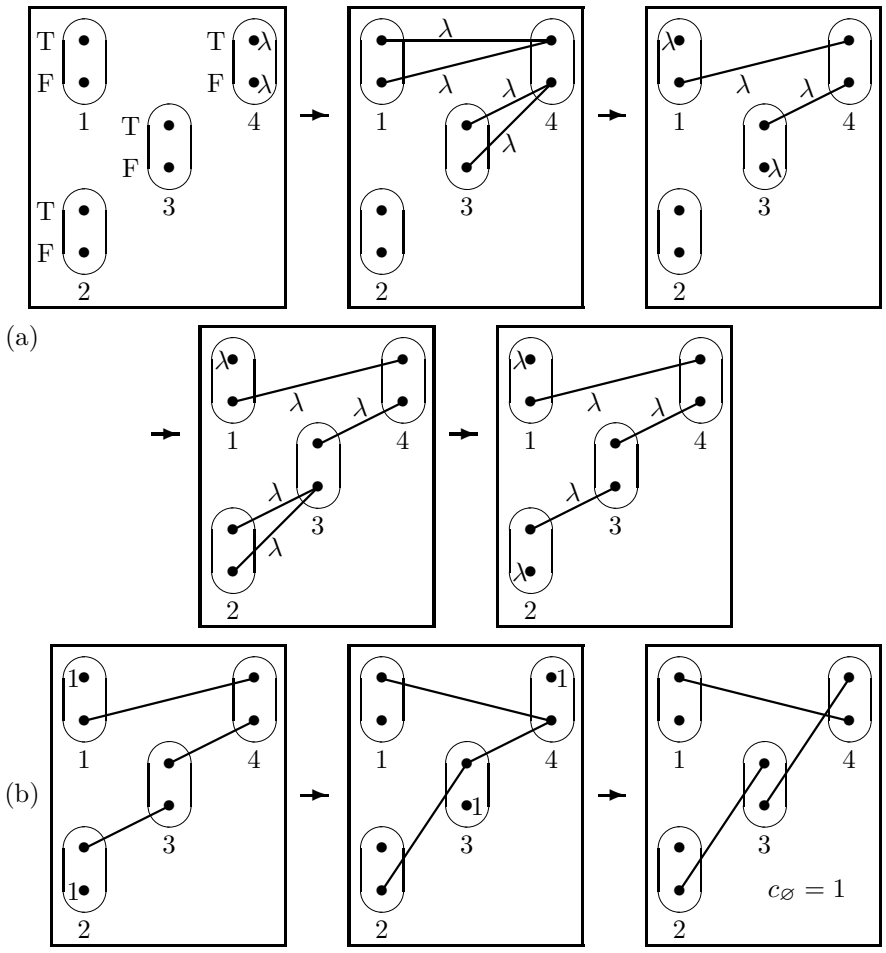

Figure 7: (a) Tracing back weights of λ from variable 4 until we arrive at non-zero weights in the original VCSP P of Figure 6; (b) applying the corresponding soft arc consistency operations to P (in the reverse order to which they were found in (a)).

(i.e. a cost which changes from 0 to 1 in the valuation structure $\mathcal{S}_1$) is recorded. In this example, inconsistency in Bool(P) is first detected at variable 4. By Theorem 7.3 this means that by soft arc consistency operations in P we can transform P into an equivalent VCSP in which $\forall x \in d_4, c_4(x) \geq \lambda$ for some $\lambda > 0$. We can associate λ with each $x \in d_4$ and trace back these weights by, at each step, using the reason for inconsistency as recorded during the establishment of arc consistency in Bool(P). This is illustrated in Figure 7(a). The weights of λ in each $c_4(x)$ ($x \in d_4$) shown in the top left box can be obtained by projection from cost functions c_{14} and c_{34} (as illustrated in the second box). If the corresponding cost in the original problem P is non-zero, which is the case for c_{14}(F,T) and c_{34}(T,F), then these weights do not need to be traced back further. The remaining weights, namely c_{14}(T,T) and c_{34}(T,F), can be obtained by projections from c_1(T) and c_3(F) as illustrated in the third box. The algorithm halts when all weights have been traced back to a non-zero costs in the original VCSP P. All the weights of λ shown in the final box of Figure 7(a) correspond to non-zero costs in the original problem P. The value of λ must not exceed any of these original costs. In this example, the maximal value we can assign to λ is clearly 1. Tracing back is equivalent to finding in

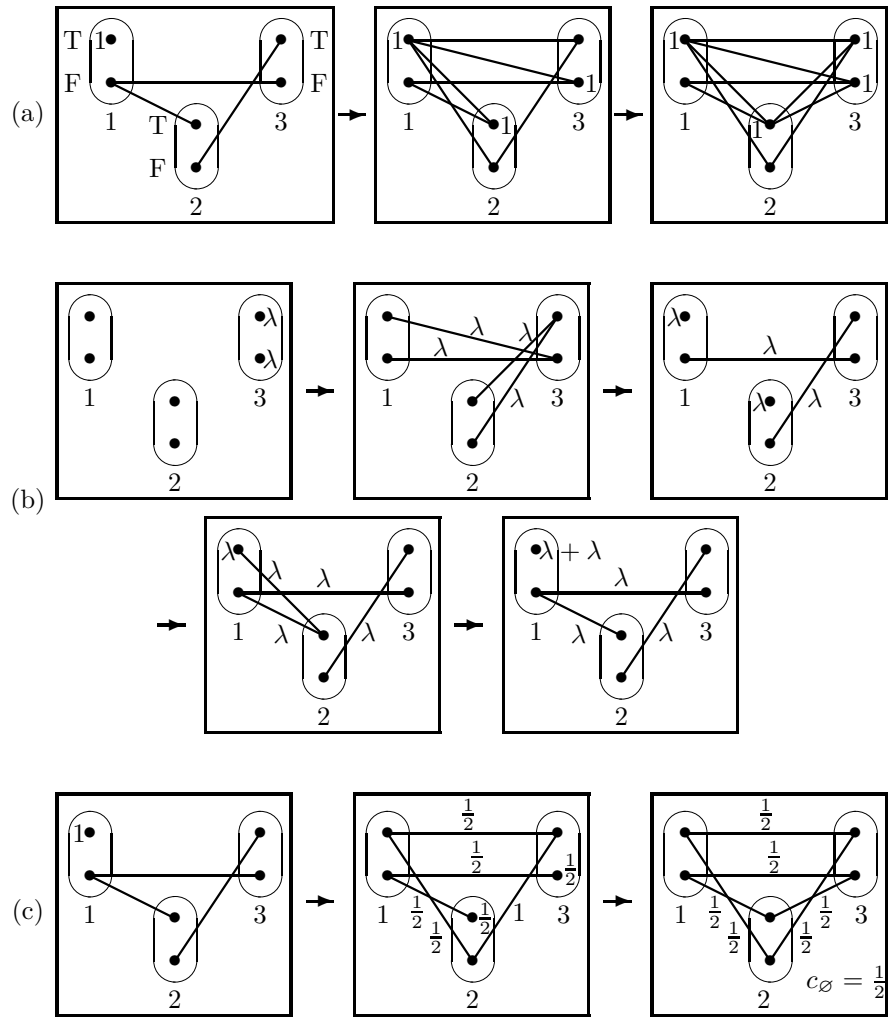

Figure 8: An example of a VCSP where virtual arc consistency produces a better lower bound than EDAC by allowing fractional weights.

reverse order a sequence of soft arc consistency operations which would produce a VCSP with $\forall x \in d_4, c_4(x) \geq \lambda$. The soft arc consistency operations can now be applied in the right order. This is illustrated in Figure 7(b). In the resulting VCSP we have $c_\varnothing = 1$. This VCSP P', shown in the final box of Figure 7(b) is virtual arc consistent since the corresponding CSP Bool(P') is arc consistent.

Unfortunately, establishing virtual arc consistency may require the introduction of fractional weights, as the following example illustrates. Consider the instance P of Max-SAT given by: $\neg X_1$; $X_1 \vee \neg X_2$; $X_1 \vee X_3$; $X_2 \vee \neg X_3$. This problem is illustrated in the leftmost box of Figure 8(a). As usual, each line represents a cost of 1 and unary costs are noted next to the corresponding domain element. Bool(P) can also be represented by the same figure, where now the value 1 is understood to be the element of the Boolean valuation structure $\mathcal{S}_1 = \langle \{0, 1\}, +_1, \geq \rangle$ in which $0 < 1$ and $1 +_1 1 = 1$ since 1 represents

complete inconsistency. Figure 8(a) illustrates the process of establishing arc consistency in Bool(P), where the detection of an inconsistency means the addition of a line or a unary cost of 1 in the figure: arc consistency operations are performed on the pairs of variables $(1,2)$, $(1,3)$ and then on the pair $(2,3)$, which leads to a domain wipe-out at variable 3. We can therefore already deduce a lower bound of the integer value 1 for the original problem P. However, in this example, no set of soft arc consistency operations with *integer* weights produces a non-zero lower bound.

In order to determine a sequence of soft arc consistency operations in P which lead to an increase $\lambda > 0$ in $c_\varnothing$, we have to retrace the steps made while establishing arc consistency in Bool(P). We place a value of λ at each element of d_3, as illustrated by the leftmost box in Figure 8(b). Retracing our steps, we know that these weights can be obtained by projection from the binary cost functions c_{13} and c_{23} (as illustrated in the next box in Figure 8(b)). If the corresponding weight in the original problem P was non-zero, such as c_{13}(F,F) and c_{23}(F,T), then such weights do not need to be traced back any further. We know that the other weights can be obtained by extension from c_1 and c_2. A weight of λ has to be traced back further via c_{12} to c_1. The algorithm halts when all remaining weights were non-zero in the original VCSP P (as shown in the last box in Figure 8(b)). We have traced a combined weight of 2λ back to c_1(T). Since c_1(T)=1 in P, the maximum value we can assign to λ is $\frac{1}{2}$.

To concretely collect this cost of $\frac{1}{2}$ in $c_\varnothing$, we apply these soft arc consistency operations, found in reverse order in Figure 8(b), to the original VCSP P with $\lambda = \frac{1}{2}$. This is shown in Figure 8(c): a weight of $\lambda = \frac{1}{2}$ is extended from $c_1(T)$ to c_{12} and then projected onto $c_2(T)$. We now have $c_2(T) = \frac{1}{2}$, which matches the virtual deletion of value $(2,T)$ in Figure 8(a). The same amount of cost $\lambda = \frac{1}{2}$ is extended from $c_1(T)$ to c_{13} and projected onto $c_3(F)$. We now have $c_3(F) = \frac{1}{2}$, which matches the virtual deletion of $(1,F)$ in Figure 8(a). In the last step of Figure 8(c), $c_2(T) = \frac{1}{2}$ is extended to c_{23} and projected onto $c_3(T)$. The situation matches the virtual wipe-out previously obtained on the right of Figure 8(a). We finally project c_3 onto $c_\varnothing$ and get an equivalent VCSP with $c_\varnothing = \frac{1}{2}$.

The example of Figure 8 shows that applying a sequence of SAC operations found by our virtual arc consistency algorithm may lead to the introduction of fractional weights in the VCSP. We have to ensure that we avoid an infinite loop in which we make smaller and smaller increases to $c_\varnothing$ each time. We give a concrete example of such an infinite loop in Appendix A. A pragmatic solution to this problem is presented in Section 11.

9. Virtual Arc Consistency subroutines

In this section we give algorithms to trace back the value of λ from $c_\varnothing$ until we reach non-zero weights in P and to propagate forward in order to actually increase $c_\varnothing$. We assume that the valuation structure used is either $\overline{\mathbb{Q}}^+$ or $\overline{\mathbb{Q}}_m$.

We give these algorithms for non-binary cost functions. This means that we in fact apply generalized arc consistency [46] rather than arc consistency in Bool(P). We assume that the generalized arc consistency algorithm applied in the first phase to Bool(P) is instrumented as follows: each time a value $a \in d_i$ is eliminated from d_i in Bool(P) because it has no support in the constraint relation R_S, this is recorded by setting $\mathsf{killer}[i,a] \leftarrow S$ and by pushing the value (i,a) itself onto a dedicated queue denoted by Q. A similar instrumentation is used in dynamic CSP algorithms such as [5]. For simplicity, we give

a formal description of this modification in the framework of an AC3-based algorithm. A time-optimal GAC algorithm is used to compute space and time complexities in our implementation.

Algorithm 2: VAC iteration - Phase 1: Instrumented AC

```
1  (* Revise variable i w.r.t. constraint R_S *);
2  Function Revise(i, S)
3      change ← false;
4      foreach a ∈ d_i do
5          if ∄t ∈ (ℓ(S) ∩ R_S) s.t. t_i = a then
6              delete a from d_i;
7              killer[i, a] ← S;
8              Q.Push(i, a);
9              change ← true;
10     return change;
11 Function Instrumented-AC()
12     P ← {(i, S) | c_S ∈ C, i ∈ S};
13     while P ≠ ∅ do
14         (i, S) ← P.Pop();
15         if Revise(i, S) then
16             if d_i = ∅ then return i;
17             else P ← P ∪ {(j, S') | c_S' ∈ C, S' ≠ S, {i, j} ⊂ S', j ≠ i};
18     return 0;
```

Compared to the traditional Revise() procedure, lines 7 and 8 have been added. The same modifications can be applied to an AC6 or AC2001 based algorithm. If no wipe-out occurs when AC is enforced on Bool(P), the problem is already VAC and our Instrumented-AC algorithm returns 0. Otherwise, the wiped-out variable is returned. The stack Q has a space complexity in $\mathcal{O}(n.d)$ as each value can be deleted at most once. Implemented as pointers to cost functions, the killer data-structure is also of $\mathcal{O}(nd)$. These complexities do not change the asymptotic space complexity of any GAC algorithms.

The second phase is described in Algorithm 3. It exploits the queue Q and the killer data structure to rewind the propagation history and collect an inclusion-minimal subset of value deletions that is sufficient to explain the domain wipe-out observed. For this, a Boolean $M(i, a)$ is set to true whenever the deletion of (i, a) is needed to explain the wipe-out and needs to be traced back. This phase also computes the quantum of cost λ that we will ultimately add to $c_\varnothing$. Using the previous killer structure, it is always possible to trace back the cause of deletions until a non zero cost is reached: this will be the source from which the cost of λ must be taken. However, in classical CSP, the same forbidden labeling or value may be used multiple times, as has been shown in the example of Figure 8. In order to compute the value of λ, we must know how many quanta of costs are requested for each solicited source of cost in the original VCSP, at the unary or r-ary level. For a labeling t_S of scope S, such that $c_S(t_S) \neq 0$, we use an integer

$k(S, t_S)$ to store the number of requests of the quantum λ on $c_S(t_S)$. Using the queue Q guarantees that the deleted values are explored in anti-causal order: a deleted value is always explored before any of the deletions that caused its deletion. Thus, when the cost request for a given tuple is computed, it is based on already computed counts and it is correct. Ultimately, we will be able to compute λ as the minimum of $\frac{c_S(t_S)}{k(S,t_S)}$ for all t_S such that $k(S, t_S) \neq 0$. This ratio represents the cost the constraint c_S can provide divided by the number of requests for this cost.

Algorithm 3: VAC iteration - Phase 2: Computing λ

```
1  Initialize all k, k_S to 0, λ ← ∞ ;
2  i_0 ← Instrumented-AC() ;
3  if (i_0 = 0) then return;
4  foreach a ∈ D_{i_0} do
5      k(i_0, a) ← 1, M(i_0, a) ← true;
6      if (c_{i_0}(a) ≠ 0) then M(i_0, a) ← false, λ ← min(λ, c_{i_0}(a)) ;
7  while (Q ≠ ∅) do
8      (i, a) ← Q.Pop() ;
9      if (M(i, a)) then
10         S ← killer[i, a];
11         R.Push(i, a) ;
12         foreach t ∈ ℓ(S) s.t. t_i = a do
13             if (c_S(t) ≠ 0) then
14                 k(S, t) ← k(S, t) + k(i, a);
15                 λ ← min(λ, c_S(t) / k(S,t));
16             else
17                 Let j ∈ S, j ≠ i be a variable that invalidates t in Bool(P);
18                 if (k(i, a) > k_S(j, t_j)) then
19                     k(j, t_j) ← k(j, t_j) + k(i, a) − k_S(j, t_j);
20                     k_S(j, t_j) ← k(i, a) ;
21                     if (c_j(t_j) = 0) then M(j, t_j) ← true;
22                     else λ ← min(λ, c_j(t_j) / k(j,t_j));
```

Initially, all k are equal to 0 except at the variable i_0 that has been wiped-out where one quantum is needed for each value (line 5). In the simplest case, some cost is already available for some values of the wiped out variable: no backtracing is required and the value of λ is updated accordingly (line 6). Otherwise, a value (i, a) extracted from Q (line 8) was deleted during arc consistency in Bool(P) by lack of support in the constraint relation R_S associated with c_S of scope killer$[i, a] = S$. If cost is needed at (i, a) (line 9), this lack of support on each tuple $t \in \ell(S)$ extending (i, a) can be due to the fact that:

1. t is forbidden by R_S in Bool(P) which means that $c_S(t) \neq 0$ (line 13). The traceback can stop as the number of quanta requested can directly be taken from $c_S(t)$. The counter k associated with labeling t (line 14) and λ (line 15) are updated accordingly.

2. otherwise, t is not valid because for one of the variables $j \in S, j \neq i$, the value (j, t_j) was deleted and $k(i, a)$ quanta of costs are needed from it. Note that if different values of other variables in S request different numbers of quanta from value (j, t_j) through c_S, just the *maximum* amount is needed since one extension from (j, t_j) to c_S provides cost to all $c_S(t)$ for t extending (j, t_j). To maintain this maximum, we use another data structure, $k_S(j, t_j)$ to store the number of quanta requested on (j, t_j) through c_S. We therefore have $k(j, b) = \sum k_S(j, b)$, where we sum over all $S \in C$ such that $j \in S$. Here, if the new request is higher than the known request (line 18), $k(j, t_j)$ (line 19) and $k_S(j, t_j)$ (line 20) must be increased accordingly. If there is no unary cost $c_j(t_j)$ explaining the deletion, this means that the value (j, t_j) has been deleted by GAC enforcing and we need to trace back the deletion of (j, t_j) inductively (line 21). Otherwise, the traceback can stop at (j, t_j) and λ is updated (line 22).

The last phase is described in Algorithm 4 and actually modifies the original VCSP by applying the sequence of equivalence-preserving transformations identified in the previous phase in reverse order, thanks to the queue R. For each value (j, b) which has been deleted in Bool(P) and which is needed to explain the wipe-out, we identify the cost function c_S that enabled this deletion in Bool(P). We then move all the unary costs required in the scope S using Extend() (line 4) and move it to the deleted value (j, b) using Project() (line 4). The amounts of cost extended and projected are always equal to the cost quantum λ multiplied by the number of requests given by the k data-structure. Ultimately, we reach the wipe-out variable i_0 and move the quantum cost to $c_\varnothing$. The new VCSP will have an improved $c_\varnothing$, as Theorem 7.3 shows.

Algorithm 4: VAC iteration - Phase 3: Applying equivalence-preserving transformations

```
1 while (R ≠ ∅) do
2     (j, b) ← R.Pop() ;
3     S ← killer[j, b] ;
4     foreach i ∈ S, i ≠ j, a ∈ D_i s.t. k_S(i, a) ≠ 0 do
5         Extend(i, a, S, λ × k_S(i, a));
6         k_S(i, a) ← 0 ;
7     Project(S, j, b, λ × k(j, b)) ;
8 UnaryProject(i_0, λ) ;
```

Because of the $k(S, t)$ data structure, the algorithm has a $\mathcal{O}(ed^r)$ space complexity where r is the maximum arity of cost functions in P. It is possible to get round this exponential number of counters by observing that quanta requests on $c_S(t)$ for $|S| > 1$ can come only from some variables $i \in S$. For every variable $i \in S$, $k(i, t_i)$ quanta are requested by i if $\mathsf{killer}[i, t_i] = S$ and $M(i, t_i)$ is true. Thus, the $k(S, t)$ need not to be maintained (removing line 14 of Alg. 3). When the value of a $k(S, t)$ is needed (line 15), it can be computed on the fly as:

$$k(S, t) = \sum_{\substack{(i \in S) \\ (\mathsf{killer}[i, t_i] = S) \wedge (M(i, t_i))}} k_S(i, t_i)$$

By implementing killer as pointers to cost functions, we get a time complexity of $\mathcal{O}(|S|)$ instead of constant time. Because of the k_S counters, we ultimately get an $\mathcal{O}(erd)$ space complexity. As for time complexity, one iteration of the algorithm has time complexity of $\mathcal{O}(ed^r)$. This is true for the first phase as long as an optimal GAC algorithm is used since the instrumentation itself is $\mathcal{O}(nd)$. The 2nd phase is $\mathcal{O}(nd^r)$ since there are at most nd values in P and the loop at line 12 takes $\mathcal{O}(d^{r-1})$. An $\mathcal{O}(ed^r)$ complexity applies to the last phase.

10. Problems solved by virtual arc consistency

When a problem P is virtual arc consistent, it is known that the problem Bool(P) has a non-empty (generalized) arc-consistency closure. This allows VAC to inherit various tractable problem classes which are solved by (generalized) arc-consistency in CSP. For example, VAC can solve submodular minimization problems, a non-trivial polynomial language of VCSP over the valuation structure $\overline{\mathbb{Q}}^+$ [11]. It is already known that OSAC solves VCSPs with submodular cost functions [15]. In this section, we give a simpler proof that the weaker notion of VAC is sufficient to solve such problems.

Definition 10.1. *In the valuation structure $\overline{\mathbb{Q}}^+$ or $\overline{\mathbb{Q}}_m$, assuming a given total ordering on every domain, a cost function c_S is submodular if $\forall t, t' \in \ell(S)$, $c_S(\max(t,t')) \oplus c_S(\min(t,t')) \leq c_S(t) \oplus c_S(t')$ where* max *and* min *represent component-wise applications of* max *(resp.* min*) on the tuples t, t'.*

Over the valuation structure $\overline{\mathbb{Q}}^+$, the class of submodular cost functions includes functions such as $\sqrt{x^2+y^2}$ or ϕ_r (for $r \geq 1$) [11] where

$$\phi_r(x,y) \;=\; \begin{cases} (x-y)^r & \text{if } x \geq y \\ \infty & \text{otherwise} \end{cases}$$

useful in bioinformatics [57] and captures simple temporal CSP with linear preferences [34]. Other well-known examples of submodular functions are the cut function of a graph [20] or of a hypergraph [28], and the rank function of a matroid. The complexity of the fastest known fully-combinatorial algorithm for submodular function minimization in $\overline{\mathbb{Q}}^+$ is $\mathcal{O}(N^5\gamma + N^6)$ where N is the number of boolean variables and γ is the time to calculate the submodular function to be minimized [47]. The standard coding of a VCSP with submodular cost functions and n variables of domains-size d as a submodular function minimization problem requires $N = n(d-1)$ Boolean variables [11].

Theorem 10.2. *Over the valuation structure $\overline{\mathbb{Q}}^+$ or $\overline{\mathbb{Q}}_m$, let P be a VCSP whose cost functions are all of arity bounded by a constant and are all submodular for a given domain ordering. If P is VAC, then an optimal solution to P can be found in polynomial time and its cost is given by $c_\varnothing$.*

Proof: If $c_\varnothing = \infty$, then any assignment is trivially an optimal solution of cost $c_\varnothing$. Suppose now that $c_\varnothing$ is finite. It follows directly from Definition 10.1 that if c_S is submodular then $\forall t, t' \in \ell(S)$ such that $c_S(t) = c_S(t') = 0$, we have $c_S(\max(t,t')) =$

$c_S(\min(t,t')) = 0$. Thus cost function submodularity implies the following property on relations in Bool(P): if R_S is a relation with scope S, then $\forall t, t' \in \ell(S)$,

$$(t \in R_S) \wedge (t' \in R_S) \;\Rightarrow\; (\max(t,t') \in R_S) \wedge (\min(t,t') \in R_S)$$

where the operations max and min are applied component-wise.

This means that all the relations of Bool(P) are both min-closed and max-closed [32]. Since the VCSP P is VAC, Bool(P) has a non-empty (generalized) arc consistency closure. It follows that a solution x to Bool(P) exists and can be found in polynomial time by establishing (generalized) arc consistency and then taking maximum values in each domain [32]. The cost of x in the VCSP P is equal to $c_\varnothing$ by definition of Bool(P) and therefore optimal. ∎

The previous proof suggests a very useful and simple value ordering heuristic to use while maintaining VAC inside a branch and bound algorithm: after making Bool(P) arc consistent, the first value which has not been deleted in the arc consistent closure of Bool(P) should be tried first (as submodular cost functions are both max-closed and min-closed). This specific value ordering heuristic will be denoted as `Hval` in the experimental section.

Submodularity is defined based on an order on each domain. It may be the case that all the cost functions of a VCSP are submodular but the orders on each domain that make all these cost functions explicitly submodular is unknown. Finding the suitable domain orders for so-called permuted submodular cost functions is a polynomial problem that can be directly reduced to 2-SAT [53]. Interestingly, VAC can directly solve VCSPs with permuted submodular cost functions, without determining the permutations.

Theorem 10.3. *Over the valuation structure $\overline{\mathbb{Q}}^+$ or $\overline{\mathbb{Q}}_m$, let P be a VCSP whose cost functions are all of arity bounded by a constant and are all submodular for unknown domain orders. If P is VAC, then an optimal solution to P can be found in polynomial time and its cost is given by $c_\varnothing$.*

Proof: The VCSP P can be transformed, by some unknown domain permutations, into a VCSP P' with submodular cost functions. The (generalized) arc consistency closure of a CSP being independent of domain orderings, the (generalized) arc consistency closure of Bool(P') is also non-empty and hence P' is also VAC. The existence of a solution of cost $c_\varnothing$ follows directly from Theorem 10.2.

Although a solution cannot be directly identified in this case (by taking maximum values in each domain), a solution can nevertheless be identified without backtrack by maintaining (generalized) arc consistency in Bool(P) during search. This search is backtrack-free provided we only accept an assignment if making this assignment and establishing (generalized) arc consistency in Bool(P) leads to a non-empty closure. Assigning a value to a variable preserves the max-closed nature of the constraints, and a (generalized) arc consistent CSP with max-closed constraints necessarily has a solution [32]. ∎

Corollary 10.4. *Over the valuation structure $\overline{\mathbb{Q}}^+$, let P be a VCSP whose cost functions are all submodular for some (known or unknown) domain orders. Then after establishing virtual arc consistency, the cost of an optimal solution to P is given by $c_\varnothing$.*

Proof: Because Project, Extend and UnaryProject preserve submodularity over $\overline{\mathbb{Q}}^+$ [15], establishing VAC on the submodular problem P produces an equivalent submodular VCSP which is virtual arc consistent. Hence, by Theorem 10.2 or Theorem 10.3, establishing VAC solves P. ∎

The simplicity of these proofs highlights the fact that VAC solves all polynomial classes such that the corresponding CSP Bool(P) is solved by arc consistency, provided that the property defining the tractable class is preserved under establishing VAC. Very simple cases can become significant in the VCSP case. For example, tree-structured VCSP can be solved by DAC (directional arc consistency) but this requires the tree structure to be detected and a specific variable order to be specified for DAC enforcing. A VAC tree-structured problem will be solved automatically, as arc consistency does in classical CSP. We can even give a more general result.

Proposition 10.5. *Over the valuation structure $\overline{\mathbb{Q}}^+$ or $\overline{\mathbb{Q}}_m$, if P is VAC and Bool(P) is in a class of CSPs for which arc consistency is a decision procedure, then P has an optimal solution of cost $c_\varnothing$.*

Proof: By the definition of VAC, the arc-consistency closure of Bool(P) is non-empty. Since arc consistency is a decision procedure for Bool(P), this implies that Bool(P) has a solution and hence, by definition of Bool(P), that P has a solution of cost $c_\varnothing$. This solution is necessarily optimal since $c_\varnothing$ is a lower bound on the cost of any solution. ∎

Tractable classes of CSP solved by arc consistency include max-closed CSPs [32] and CSP instances satisfying the broken-triangle property (a hybrid class which strictly generalizes tree-structured CSPs [18]). By Proposition 10.5, if after establishing VAC, Bool(P) falls into one of these tractable classes, then the VCSP P is also solved. As an example of a very simple case, one can observe that any VCSP problem P which is VAC and such that Bool(P) has all domains reduced to singletons is also solved. Note that for the VCSP P, this just means that there is no variable which has two (or more) values with unary cost 0. Note, however, that in general, these properties of Bool(P) may be destroyed under soft arc consistency operations and hence may not define a tractable class that can be recognized before establishing VAC.

11. Experimental trials of our VAC algorithm

11.1. Heuristic Implementation of our VAC algorithm

To study the actual quality of the VAC bound for solving VCSP, we restricted ourselves to binary cost functions for simplicity. Since the number of iterations of our VAC algorithm described in Section 9 can be unbounded (as shown on an example in Appendix A), we enforce an approximation of VAC using a threshold ε. If more than a given number of iterations never improve $c_\varnothing$ by more than ε then VAC enforcing stops prematurely. This is called VAC$_\varepsilon$. In $\overline{\mathbb{Q}}_m$, the number of iterations is thus $\mathcal{O}(\frac{m}{\varepsilon})$ and hence the total complexity of VAC$_\varepsilon$ is $\mathcal{O}(ed^2m/\varepsilon)$. When one iteration does not increase the lower bound by more than ε, one bottleneck (a cost that fixed the value of λ) is identified and the unary and binary costs corresponding to one of the variables concerned by the bottleneck are ignored in Bool(P) at following iterations.

In order to rapidly collect large cost contributions, and similarly to what has previously been done in maximum flow algorithms [2], we replaced $\text{Bool}(P)$ by a relaxed but increasingly strict variant $\text{Bool}_\theta(P)$. A tuple t is forbidden in $\text{Bool}_\theta(P)$ iff its cost in P is larger than θ. After sorting the list of non zero binary costs $c_{ij}(a,b)$ in a fixed number k of buckets, the decreasing minimum costs observed in each bucket define a sequence of thresholds $(\theta_1, \ldots, \theta_k)$. Starting from θ_1, iterations are performed at a fixed threshold until no wipe-out occurs. Then the next value θ_{i+1} is used. After θ_k, a geometric schedule defined by $\theta_{i+1} = \frac{\theta_i}{2}$ is used and stopped when $\theta_i \leq \varepsilon$.

11.2. Value ordering heuristic

When P is virtual arc consistent, values which have been deleted in the arc consistent closure of $\text{Bool}(P)$ imply a cost larger than $c_\varnothing$. This information can be used to direct search towards good solutions. Quickly finding a good (but not necessarily optimal) solution is an essential ingredient of branch and bound, since it provides a tighter upper bound on the optimal cost. Since the valuation structure used during branch and bound is $\mathcal{S}_m$ where m is the current upper bound, a tighter upper bound will lead to more effective pruning during search.

In this experimental section, we therefore consider a new value ordering heuristic which selects the minimum domain value which has not been deleted in $\text{Bool}(P)$. This value ordering heuristic is more informed than the value ordering heuristic that selects the EAC support values (see Definition 4.5) used in the `toulbar2` solver. It also has the nice property (see section 10) that it will guide the solver towards an optimal solution for non-permuted submodular problems. The combination of this value ordering heuristic with VAC_ε maintenance in a branch and bound procedure is known as `VAC+Hval` in the following experimental results.

11.3. Experiment setup

In this section we present experimental results on VAC_ε using `toulbar2` version 0.8 written in C++ (section Algorithms in [22]). Our implementation uses fixed point representation of costs. To achieve this, all initial costs in the problem are multiplied by $\frac{1}{\varepsilon}$ which is assumed to be an integer. To exploit the knowledge that the original problem had integer costs, branch and bound pruning occurs as soon as $\frac{\lceil c_\varnothing \times \varepsilon \rceil}{\varepsilon} \geq m$ where m is the global upper bound (the cost of the best known solution). As VAC_ε is incapable of producing unary costs, VAC_ε is always enforced together with FDAC.

Experiments were performed on a 3 GHz (2.66 GHz for submodular benchmarks) Intel Xeon with 16 GB. Our solver includes a last conflict driven variable selection heuristic [8], elimination of variable with degree lower than two during search [42] and binary branching[4]. The default value of ε used in VAC_ε was $\varepsilon = \frac{1}{10{,}000}$.

Because of the overhead of each iteration of VAC_ε, which implies a reconstruction of $\text{Bool}_\theta(P)$, the convergence of VAC_ε is stopped prematurely during search (except for the random benchmark problems), using a final θ larger than during preprocessing. This enforces VAC_ε only when it is capable of providing large improvements in the lower bound. No non-trivial initial upper bound was used on the random instances.

[4]For small domains ($d \leq 10$) ,a value is assigned or removed. Larger domains are split in two halves

preprocessing		ST		DT		CT	
		lb	time	lb	time	lb	time
	EDAC	16	<.01s	18	<.01s	40	<.01s
	VAC_ε	25	.06s	28	.09s	49	.25s
	OSAC	27	10.5s	32	2.1s	74	631s

Table 3: A comparison of EDAC, VAC_ε and OSAC for preprocessing random MaxCSP.

11.4. Evaluation of VAC_ε lower bounds

In this first set of experiments, we analyse the strength of the lower bounds provided by VAC_ε compared to other lower bounds, including OSAC.

Random MaxCSP. We report results on the problems described in Section 6.1. These are Sparse Tight, Dense Tight, Complete Tight (ST, DT, CT with 32 variables, 10 values, 50 instances per class) where VAC_ε and OSAC preprocessing yield non-trivial lower bounds. Table 3 shows the time and the quality of the lower bound (lb) after preprocessing by EDAC, VAC_ε and OSAC (ILP formulation solved by CPLEX 11.0).

As expected, OSAC always provides the strongest lower bound. VAC_ε computes a lower bound which is 8% (ST) to 33% (CT) weaker than OSAC and is one to three orders of magnitude faster. These considerable speedups thus have only a fairly moderate impact on the strengths of the lower bounds.

Frequency assignment problems. The problems considered here were already described in section 6.1. Considering just the lower bounds produced, Table 4 shows that VAC_ε is again one to two orders of magnitude faster than OSAC and gives almost the same lower bounds on the $\texttt{graph11}_r$ and $\texttt{graph13}_r$ instances.

preprocessing			$scen07_r$	$scen08_r$	$graph11_r$	$graph13_r$
	lb	EDAC	10000	6	2710	8722
		VAC_ε	29498	35	2955	9798
		OSAC	31454	48	2957	9798
	time	VAC_ε	211s	86s	3.5s	29s
		OSAC	3530s	6718s	492s	6254s

Table 4: A comparison of the lower bounds produced by OSAC and VAC_ε on different RLFAP instances.

Overall, our unoptimized version of VAC_ε seems capable of producing significantly stronger lower bounds than EDAC alone and is also one to three orders of magnitude faster than a highly optimized linear programming solver which does not always produce a better lower bound. VAC_ε is therefore an attractive component for a branch and bound search.

11.5. Submodularity

In this section, we try to evaluate the efficiency of VAC_ε on submodular problems (or on problems with a large part of submodular cost functions).

Random binary submodular problems. The following procedure was used to generate random binary submodular problems: at the unary level, every value receives a 0/1/2/3 cost with identical probability. Binary submodular cost functions can be decomposed into a sum of so-called *generalized interval functions* [10]. A generalized interval function $\eta_{a,b}(x, y)$ is defined by a fixed cost (we used 3) and bounds a and b for the variables x and y:

$$\eta_{a,b}(x,y) \;=\; \begin{cases} 0 & \text{if } (x < a) \vee (y > b) \\ 3 & \text{otherwise} \end{cases}$$

We summed together p (with p a randomly-chosen integer value in $[0, d[$, where d is the size of each domain) such generalized interval functions $\eta_{a,b}(x, y)$, using uniformly-sampled random values a and b, to generate each submodular binary cost function. The domains of all variables were then randomly permuted to "hide" submodularity.

Problems have from $n = 100$ to $n = 450$ variables, 20 domain values, and $(n-1)n/8$ binary constraints, and 50 instances per class. The cpu-time to solve these problems, including the proof of optimality, is reported in Figure 9 (with a time limit of 1 hour). Figure 9 shows that maintaining VAC_ε rapidly outperforms EDAC on these problems. Although OSAC can solve permuted submodular problems in polynomial time [15], the degree of this polynomial is such that OSAC could not be applied to problems of this size. Thus, even though VAC_ε only establishes an approximation of virtual arc consistency, maintaining VAC_ε proved to be much faster than OSAC on these submodular problems. Similarly, the state-of-the-art fully combinatorial polynomial-time algorithm for submodular function minimization [47] could not be applied to problems of this size since its complexity is $\mathcal{O}((nd)^5 e)$.

Notice the speed-up offered by the enhanced value ordering (`VAC+Hval` in Figure 9) compared to the default value ordering heuristic (`VAC` in Figure 9).

Partly-submodular random problems. To evaluate the influence of the existence of a limited number of submodular cost functions, we started from random dense tight problems as generated in [16], replacing a given percentage of cost functions by permuted binary submodular cost functions (100% means a fully submodular instance). Problems have 100 variables, 10 values, 1,237 binary constraints, and 50 instances per class. The results are reported in Figure 10 where a logarithmic scale is used for the cpu-time axis. We set a time limit of 1 hour (the average being calculated assigning 1 hour to problems that were unsolved within this time limit). When 90% of the cost functions are submodular, VAC_ε (`VAC` or `VAC+Hval`) is two orders of magnitude faster than EDAC. For less than 75% submodular cost functions, both EDAC and VAC_ε did not solve the instances within the 1-hour time limit. As the percentage of submodular cost functions decreases, `VAC+Hval` becomes less efficient than `VAC` although it develops slightly less search nodes. This is due to the overhead in maintaining a more complex value ordering heuristic. In the rest of the experiments, we therefore used the enhanced `VAC+Hval` value ordering heuristic for submodular benchmarks only.

Feedback arc set. Given a directed graph, the feedback arc set problem consists in removing a minimum subset of the arcs in order to obtain an acyclic subgraph. An alternative formulation is to find a total order $<$ on the vertices such that there is a minimum number of feedback arcs (i.e. an arc from i to j with $i > j$). This problems is NP-hard [25]. In order to experiment with submodular problems, we modified the penalty function so that

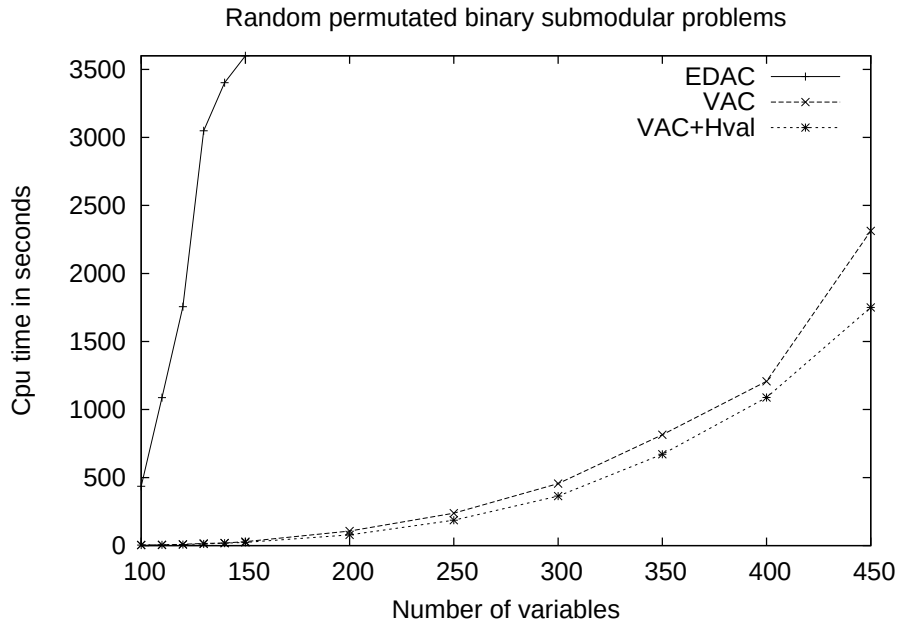

Figure 9: A comparison of the efficiency of algorithms maintaining EDAC and VAC$_\varepsilon$ (with or without the enhanced value ordering heuristic described in Section 11.2) on random permuted binary submodular problems.

if there is a feedback arc (i, j), instead of having a cost of 1 we have a cost proportional to the difference between the ordering positions of i and j. The resulting problem is similar to a simple temporal CSP with linear preferences [34]. We took instances with $n = 50$ vertices and from 100 to 900 arcs from Resende's home page[5]. In our WCSP model there is a variable x_i with domain $[1, n]$ corresponding to each vertex i. For each arc (i, j), there is a cost function $max(0, x_i - x_j + 1)$. The results are reported in Figure 11. The time limit was almost 2 days. When the number of arcs is less than 150, OSAC preprocessing solves the problem without search. However, it is much more expensive than EDAC or VAC$_\varepsilon$. As the problem is submodular, VAC$_\varepsilon$ is quite efficient compared to EDAC. However, despite this submodularity, VAC$_\varepsilon$ was slower than EDAC on the densest instances. When the graph density is high, VAC$_\varepsilon$ tends to more frequently find cyclic arc-inconsistency proofs in Bool(P), resulting in small rational cost increments that may cause the premature termination of VAC$_\varepsilon$, with a loose lower bound $c_\varnothing$. As shown in Figure 11, lowering the value of ε effectively improves the lower bound $c_\varnothing$ and reduces the search effort especially when the constraint graph density increases. Using a smaller threshold $\varepsilon = \frac{1}{1{,}000{,}000}$, VAC$_\varepsilon$ was always significantly faster than EDAC.

Minimum cut problems. Our last submodular problem example is the (s, t) minimum cut problem which consists in finding a partition of the vertices of a weighted undirected

[5] http://www.research.att.com/~mgcr/data/index.html

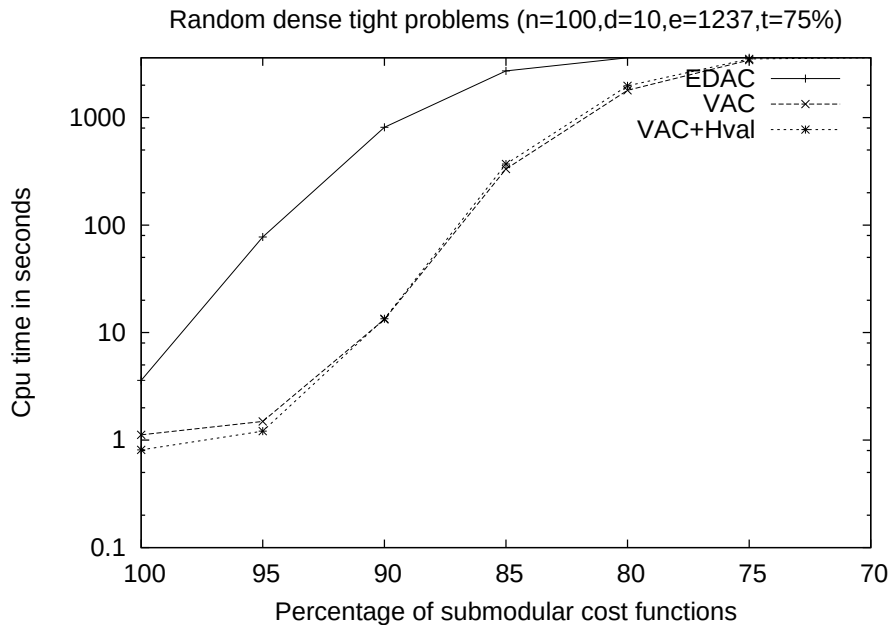

Figure 10: A comparison of the efficiency of algorithms maintaining EDAC and VAC$_\varepsilon$ on random dense tight problems with a percentage of permuted binary submodular cost functions.

graph $G = (V, E, w)$ into two disjoint subsets, one containing the source node and the other the terminal node, such that the weighted sum of edges whose end points are in different subsets of the partition is minimum. Our WCSP formulation associates one 0/1 variable with each vertex in V. For each edge $e = (i, j) \in E$, there is a soft equality cost function which returns a cost of $w(e)$ if $x_i \neq x_j$ (and 0 otherwise). We fix $x_1 = 0$ and $x_n = 1$ (since they correspond, respectively, to the source and terminal nodes). Instances were produced by the `genrmf` generator[6] [29] used in the First DIMACS Challenge. The graph is a succession of b grids each of size $a \times a$ in which each vertex is connected to its neighbours and to a randomly chosen vertex in the next grid. Capacities are selected uniformly at random in $[c_1..c_2]$ for inter-grid arcs and are fixed to $c_2 \times a^2$ for intra-grid arcs. Problems have from 16 (`genrmf_long` coefficients $a = 2, b = a^2 = 4$ and $c_1 = 1, c_2 = 100$) to 20,736 variables ($a = 12, b = 144$), from 46 to 96,626 binary constraints, and 50 instances per class. The results are reported in Figure 12 where a logarithmic scale is used for the cpu-time axis.

We compared EDAC and VAC$_\varepsilon$ ($\varepsilon = 1$) with a dedicated maximum flow algorithm (Goldberg-Tarjan push-relabel method H_PRF, cpu-time interpolated from [27] by taking cpu clock frequency ratio 1.8/2.66) and a general submodular (not restricted to binary cost functions) minimum-norm point algorithm [26] (cpu-times from [26] with the same cpu ratio applied and for $a = 10$ from [41] who have a faster implementation). The algorithms compared have widely different capabilities. The Goldberg-Tarjan algorithm

[6] www.informatik.uni-trier.de/~naeher/Professur/research/generators/maxflow/genrmf/index.html

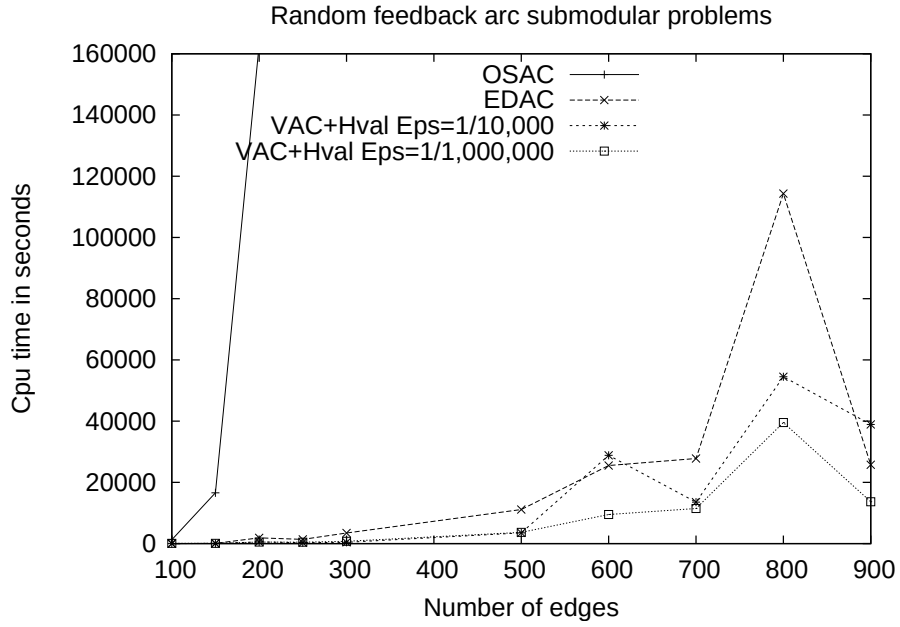

Figure 11: A comparison of the efficiency of algorithms maintaining EDAC and VAC$_\varepsilon$ for different ε values and OSAC on submodular feedback arc set problems.

is capable of solving Maxflow/Mincut problems and therefore arbitrary finite binary submodular WCSPs [7]. The general submodular algorithm is limited to submodular functions of arbitrary arities while the EDAC/VAC-based algorithms are not restricted to submodular functions or to boolean domains (although our present implementation is only designed for binary cost functions).

Not surprisingly, VAC$_\varepsilon$ is faster than a general submodular solver (7.6 times faster for $a = 10, n = 10,000$) and much slower than the dedicated and finely tuned maximum flow algorithm. Although it develops two times less nodes than EDAC, it is up to 30 times slower than EDAC due to its overhead during search. Interestingly, the arc inconsistency proofs found by arc consistency on Bool(P) were always acyclic, meaning that VAC$_\varepsilon$ (whatever the value of ε) solved this specific problem in preprocessing. It is rather surprising that a relatively simple generic WCSP solver such as EDAC solves minimum cut problems with $n \approx 15,000$ vertices in only 5.5 seconds (even if this is considerably slower than the 0.04 seconds required by a specialized and optimized maximum flow algorithm).

11.6. Solving general problems

Our final tests are dedicated to solving non-submodular problems using branch and bound search maintaining VAC$_\varepsilon$+EDAC during search. Since it includes FDAC, EDAC can remove values that would not be deleted by VAC$_\varepsilon$. It therefore provides addditional information for variable and value ordering heuristics. In the experiments, the `toulbar2` solver selects the variable with the smallest ratio of current domain size divided by current

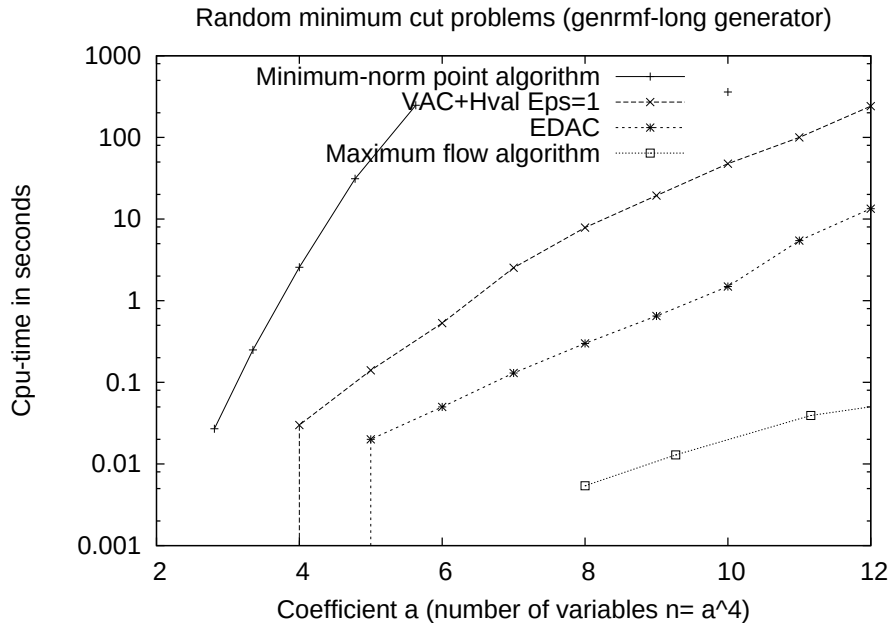

Figure 12: A comparison of the efficiency of algorithms maintaining EDAC and VAC_ε on random minimum cut problems, compared to state-of-the-art maximum flow and general submodular algorithms.

number of constraints involving the variable. Ties are broken by choosing a variable with maximum unary cost.

Frequency assignment. Experiments were performed on the same CELAR instances as mentioned in Section 11.4. During search, VAC_ε was stopped at $\theta = 1000\varepsilon$. Table 5 reports the results on the open instances `graph11` and `graph13` (see fap.zib.de/problems/CALMA) which are solved to optimality for the first time both in their reduced and original formulation, given the best known upper bound. Table 5 also gives the results on the instance `scen06`. The table gives for each problem the number of variables, total number of values, number of cost functions, cpu-time for EDAC alone (a dash for $> 10^4$ seconds), number of nodes explored with VAC_ε, cpu-time with VAC_ε, mean increase of the lower bound observed after one VAC_ε iteration (lb/iter) and total number of VAC_ε iterations (nb. iter). We observed that the value k (number of cost requests) at each VAC_ε iteration can be high, reaching a mean value of 16 in some resolutions of `graph` instances.

This shows that the stronger lower bound provided by VAC_ε clearly pays off on sufficiently difficult problems where a good lower bound is essential to prune a large search tree. VAC_ε is also capable of solving simpler problems, but because of the associated overheads, less computationally expensive techniques such as EDAC may outperform it.

Uncapacitated Warehouse Location Problem (UWLP). In the UWLP, the aim is to decide which facilities should be opened to provide goods to all customers with maximum profit or, equivalently, minimum cost. The cost minimization variant of the UWLP is known to

	nb. var.	nb. val	nb. c.f	EDAC cpu	VAC_ε nodes	VAC_ε cpu	lb/ iter	nb. iter
$gr11_r$	232	5747	792	-	1536	18.2s	2.5	973
gr11	340	12820	1425	-	$2 \cdot 10^5$	217min.	6.63	$2.6 \cdot 10^5$
$gr13_r$	454	13153	2314	-	32	62s	4.8	1893
gr13	458	17588	4815	-	114	254s	0.4	9486
sc06	82	3274	327	39min.	$2 \cdot 10^6$	155min.	96	$3 \cdot 10^6$

Table 5: Maintaining VAC_ε on hard RLFAP instances.

be supermodular (the opposite of a submodular cost function). Minimizing supermodular functions is known to be NP-hard. The precise problem description and WCSP model are given in [40] and [24] respectively.

We tested both EDAC and VAC_ε preprocessing followed, in both cases, by maintaining EDAC during search on instances capmq1-5 (600 variables, up to 300 values per variable and $90,000$ cost functions) and instances capa, capb and capc ($1,100$ variables, around 90 values per variable and $101,100$ cost functions). We report solving time to prove optimality (initial upper bound equal to optimum) in seconds in Table 6 (a dash for $> 10^4$ seconds). VAC_ε outperforms EDAC on 6 out of the 8 problems.

	mq1	mq2	mq3	mq4	mq5	a	b	c
EDAC	2508	3050	2953	7052	7323	6179	-	-
VAC_ε	2279	3312	2883	4024	8124	3243	4343	2751
CPLEX	622	1022	415	1266	2357	3	4.5	13

Table 6: Comparison of EDAC, VAC_ε and CPLEX 11.0 on different uncapacitated warehouse location problems.

Instances were also solved using the ILP solver CPLEX 11.0 and a direct formulation of the problem. On these problems, CPLEX is more efficient than VAC_ε. Note, however, that given the floating point representation of CPLEX and the large range of costs in these problems, the proof of optimality of CPLEX is questionable here. OSAC results are not given because LP generation overflows on these instances.

11.7. Conclusion

The lower bounds produced by VAC_ε are stronger than those produced by EDAC but weaker than those produced by OSAC. Our experiments have conclusively demonstrated that there are some problems for which maintaining VAC_ε during search is the best strategy. This is particularly true of difficult problems (such as the two frequency assignment benchmark problems closed for the first time using VAC_ε). Clearly EDAC will outperform VAC_ε whenever the time devoted by VAC_ε to finding a better lower bound is not compensated by sufficient pruning of the branch and bound search tree. This may occur for various reasons: this phenomenon has been observed in (s,t)-mincut problems reported here, but also in the extraction of an optimal plan from a planning graph [17]. It is worth pointing out that our current implementation of VAC_ε leaves room for considerable optimization.

Our experiments have confirmed the theoretical relationship between VAC and submodularity. Although VAC_ε is only an approximation to VAC, it is nevertheless capable of taking advantage of the submodular nature of cost functions to provide a good lower bound. It is also no doubt because EDAC can be considered as an approximation to VAC, that explains the rapidity of EDAC on certain submodular problems. An interesting outcome of our experiments was that VAC_ε performs well on problems containing a high proportion of submodular cost functions (to which specialized submodular algorithms are inapplicable).

12. Discussion

12.1. Virtual arc consistency by diffusion

A much simpler (but slower) algorithm, known as MIN-SUM diffusion, can also be used as an alternative to our VAC algorithm described in Section 9. MIN-SUM diffusion consists in iterating until convergence the following operation: for each $S \in C$, $i \in S$ and $a \in d_i$, call Project (i, a, S, α) where

$$\alpha \quad = \quad \frac{1}{2} \min\{c_S(t) : t \in \ell(S) \text{ such that } t_i = a\} - c_i(a)$$

Rather than sending as much cost as possible towards the unary constraint c_i, MIN-SUM diffusion equalizes costs between unary and higher-arity constraints, in the sense that after the above call of Project,

$$c_i(a) \quad = \quad \min\{c_S(t) : t \in \ell(S) \text{ such that } t_i = a\}$$

If after each iteration we establish node consistency, it is easy to see that whenever MIN-SUM diffusion converges, the resulting VCSP is VAC. MIN-SUM diffusion has been generalized to the tree-reweighting (TRW) algorithm which performs exact equalizations on trees rather than on single edges [55, 35]. In trials on binary problems from low-level computer vision, MIN-SUM diffusion was found to converge several times slower than both the TRW algorithm (where the trees corresponded to the rows and columns of the image) and the "Augmenting DAG" algorithm which is similar to our VAC algorithm described in Section 9 [39, 56].

12.2. Beyond arc consistency

It should be mentioned that forms of higher-order consistency have been proposed for VCSPs [14] which can find a better lower bound than any SAC transformation. This is at the cost of introducing higher-order cost functions. Consider the optimization version of the graph coloring problem on a triangle with two colors, equivalent to the VCSP in Figure 13, where a line represents a cost of 1. The aim is to assign a color to each node so as to minimize the number of pairs of nodes joined by an edge and assigned the same color. No SAC transformation applied to this VCSP increases $c_\varnothing$, whereas soft 3-consistency produces a lower bound $c_\varnothing = 1$ [14]. One disadvantage of establishing soft 3-consistency is that some weights are now stored in ternary cost functions.

Bool(P) is a classical CSP which has a solution if and only if the VCSP P has a solution of cost $c_\varnothing$. In the same way that virtual arc consistency uses inconsistencies detected when establishing arc consistency in Bool(P) to determine a sequence of soft arc consistency operations which increase the lower bound $c_\varnothing$ in P, other virtual consistency techniques could be defined based on other notions of consistency in Bool(P).

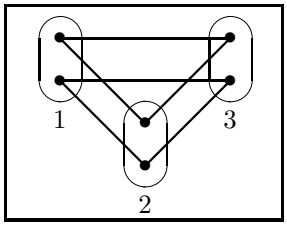

Figure 13: A VCSP corresponding to the 2-color graph-coloring optimization problem on a triangle.

13. Conclusion

We have presented new techniques for finding improved lower bounds in the finite-domain optimization problem VCSP, based on the notions of optimal and virtual arc consistency.

In order to establish optimal soft arc consistency (OSAC), after the propagation of infinite costs, a linear program is solved to determine a set of soft arc consistency operations (shifting of costs between unary and non-unary cost functions) which produces an equivalent instance with a maximum value of the constant cost term. This constant cost term represents a natural lower bound and plays an essential role in branch and bound search. When all costs are finite, the resulting constant cost term is optimal among all equivalent instances with the same set of constraint scopes. Experimental trials have demonstrated the potential utility of establishing OSAC during preprocessing.

Virtual arc consistency (VAC) can be seen as an approximation to OSAC that can be applied either during preprocessing or at every node of a search tree. If a VCSP is virtual arc consistent, then this means that no sequence of soft arc consistency operations could increase the lower bound $c_\varnothing$. In particular, the previous state-of-the-art soft consistency technique EDAC (Existential Directional Arc Consistency) [43] cannot increase $c_\varnothing$ for any variable order.

Virtual arc consistency can be tested in $\mathcal{O}(ed^2)$ time in the case of a binary VCSP using an optimal arc consistency algorithm, such as AC-2001 [6] in the CSP Bool(P). It can also be established in polynomial time by simply establishing OSAC. The main aim of soft consistency techniques is to rapidly find a good (but not necessarily optimal) lower bound. Therefore, in our experimental trials we used an algorithm with guaranteed low-order polynomial time complexity which established a relaxed version VAC_ε of virtual arc consistency, in order to avoid problems of convergence generated by the introduction of smaller and smaller fractional weights. Applying VAC_ε during branch and bound search allowed us to close two longstanding open frequency assignment problems.

Acknowledgments. We would like to thank Arie Koster and Achemi Bennaceur for discussions on the OSAC lower bound and its relation, by duality, to the linear relaxation of the ILP formulation of weighted CSP given in [37]. The presentation of the paper was greatly improved thanks to the remarks of the anonymous reviewers. This research was partly funded by the Agence Nationale de la Recherche (STALDECOPT project).

References

[1] Affane, M. S., Bennaceur, H., 1998. A weighted arc consistency technique for Max-CSP. In: Proc. of the 13^{th} ECAI. Brighton, United Kingdom, pp. 209–213.

[2] Ahuja, R. K., Magnanti, T., Orlin, J., 1993. Network Flows: Theory, Algorithms, and Applications. Prentice Hall.

[3] Apt, K., 1999. The essence of constraint propagation. Theoretical computer science 221 (1-2), 179–210.

[4] Bennaceur, H., Osmani, A., 2003. Computing lower bounds for Max-CSP problems. In: Developments in Applied Artificial Intelligence. 22718. Springer, pp. 217–240.

[5] Bessière, C., 1991. Arc-consistency in dynamic constraint satisfaction problems. In: Proc. of AAAI'91. Anaheim, CA, pp. 221–226.

[6] Bessière, C., Régin, J.-C., 2001. Refining the basic constraint propagation algorithm. In: Proc. IJCAI'2001. pp. 309–315.

[7] Boros, E., Hammer, P., 2002. Pseudo-Boolean Optimization. Discrete Appl. Math. 123, 155–225.

[8] C. Lecoutre, L. Saïs, S. Tabary, V. Vidal, 2009. Reasoning from last conflict(s) in constraint programming. Artificial Intelligence 173 (18), 1592–1614.

[9] Cabon, B., de Givry, S., Lobjois, L., Schiex, T., Warners, J., 1999. Radio link frequency assignment. Constraints 4, 79–89.

[10] Cohen, D. A., Cooper, M. C., Jeavons, P. G., Krokhin, A. A., 2004. A Maximal Tractable Class of Soft Constraints. Journal of Artificial Intelligence Research 22, 1–22.

[11] Cohen, D. A., Cooper, M. C., Jeavons, P. G., Krokhin, A. A., Aug. 2006. The complexity of soft constraint satisfaction. Artificial Intelligence 170 (11), 983 – 1016.

[12] Cooper, M. C., 2003. Reduction operations in fuzzy or valued constraint satisfaction. Fuzzy Sets and Systems 134 (3), 311–342.

[13] Cooper, M. C., 2004. Cyclic consistency: a local reduction operation for binary valued constraints. Artificial Intelligence 155 (1-2), 69–92.

[14] Cooper, M. C., 2005. High-order consistency in Valued Constraint Satisfaction. Constraints 10, 283–305.

[15] Cooper, M. C., 2008. Minimization of locally-defined submodular functions by Optimal Soft Arc Consistency. Constraints 13 (4).

[16] Cooper, M. C., de Givry, S., Schiex, T., Jan. 2007. Optimal soft arc consistency. In: Proc. of IJCAI'2007. Hyderabad, India, pp. 68–73.

[17] Cooper, M. C., de Roquemaurel, M., Régnier, P., 2009. A weighted CSPapproach to cost-optimal planning. Tech. Rep. RR-2009-28-FR, IRIT, Toulouse, France.

[18] Cooper, M. C., Jeavons P., Salamon A., 2008. Hybrid tractable CSPs which generalise tree structure. In: Proc. ECAI'08. pp. 530–534.

[19] Cooper, M. C., Schiex, T., 2004. Arc consistency for soft constraints. Artificial Intelligence 154 (1-2), 199–227.

[20] Cunningham, W., 1985. Minimum cuts, modular functions, and matroid polyhedra. Networks 15 (2), 205–215.

[21] de Givry, S., 1999. Minorants de problèmes de minimisation de violation de contraintes : recherche de bonnes relaxations à l'aide de méthodes incomplètes. In: Proc. of JNPC-99. Lyon, France.

[22] de Givry, S., Heras, F., Jarrosa, J., Rollon, E., Schiex, T., 2003. The SoftCSP and Max-SAT benchmarks and algorithms web site. carlit.toulouse.inra.fr/cgi-bin/awki.cgi/softcsp.

[23] de Givry, S., Schiex, T., Verfaillie, G., 2006. Exploiting Tree Decomposition and Soft Local Consistency in Weighted CSP. In: Proc. of the National Conference on Artificial Intelligence, AAAI-2006. pp. 22–27.

[24] de Givry, S., Zytnicki, M., Heras, F., Larrosa, J., 2005. Existential arc consistency: getting closer to full arc consistency in weighted CSPs. In: Proc. of IJCAI-05. Edinburgh, Scotland, pp. 84–89.

[25] Festa, P., Pardalos, P., Resende, M., 1999. Feedback set problems. In: Handbook of Combinatorial Optimization. Kluwer Academic Publishers, pp. 209–258.

[26] Fujishige, S., Hayashi, T., Isotani, S., 2006. The minimum-norm-point algorithm applied to submodular function minimization and linear programming. Tech. rep., Research Institute for Mathematical Sciences, Kyoto, Japan.

[27] Fujishige, S., Isotani, S., 2003. New maximum flow algorithms by ma orderings and scaling. Journal of the Operations Research Society of Japan 46, 243–250.

[28] Fujishige, S., Patkar, S. B., 2001. Realization of set functions as cut functions of graphs and hypergraphs. Discrete Math. 226, 199–210.

[29] Goldfarb, D., Grigoriadis, M. D., 1988. A computational comparison of the dinic and network simplex methods for maximum flow. Annals of Oper. Res. 13, 83–123.
[30] Hammer, P., Hansen, P., Simeone, B., 1984. Roof duality, complementation and persistency in quadratic 0-1 optimization. Math. Programming 28, 121–155.
[31] Heras, F., Larrosa, J., Oliveras, A., May 2007. MiniMaxSat: A New Weighted Max-SAT Solver. In: Proc. of SAT'2007. No. 4501 in LNCS. Lisbon, Portugal, pp. 41–55.
[32] Jeavons, P., Cooper, M., Dec. 1995. Tractable constraints on ordered domains. Artificial Intelligence 79 (2), 327–339.
[33] Karmarkar, N., 1984. A new polynomial time algorithm for linear programming. Combinatorica 4 (4), 373–395.
[34] Khatib, L., Morris, P., Morris, R., Rossi, F., 2001. Temporal constraint reasoning with preferences. In: Proc. of the 17^{th} IJCAI. Seattle, WA, pp. 322–327.
[35] Kolmogorov, V., 2006. Convergent tree-reweighted message passing for energy minimization. IEEE Trans. on Pattern Analysis and Machine Intelligence 28 (10), 1568–1583.
[36] Koster, A., van Hoesel, S., Kolen, A., 1998. The partial constraint satisfaction problem: facets and lifting theorems. Oper. Res. Lett. 23 (3-5), 89–97.
[37] Koster, A., van Hoesel, S., Kolen, A., 1999. Solving frequency assignment problems via tree-decomposition. Tech. Rep. RM/99/011, Universiteit Maastricht, Maastricht, The Netherlands.
[38] Koster, A. M. C. A., Nov. 1999. Frequency assignment: Models and algorithms. Ph.D. thesis, University of Maastricht, The Netherlands, available at www.zib.de/koster/thesis.html.
[39] Koval, V. K., Schlesinger, M. I., 1976. Dvumernoe programmirovanie v zadachakh analiza izobrazheniy (Two-dimensional programming in image analysis problems). USSR Academy of Science, Automatics and Telemechanics 8, 149–168, in Russian.
[40] Kratica, J., Tosic, D., Filipovic, V., Ljubic, I., 2001. Solving the Simple Plant Location Problems by Genetic Algorithm. RAIRO Operations Research 35, 127–142.
[41] Krause, A., Guestrin, C., 2009. Ijcai tutorial on intelligent information gathering and submodular function optimization. Tech. rep., Caltech/CMU, Pasadena, www.submodularity.org.
[42] Larrosa, J., Sep. 2000. Boosting search with variable elimination. In: Principles and Practice of Constraint Programming - CP 2000. Vol. 1894 of LNCS. Singapore, pp. 291–305.
[43] Larrosa, J., de Givry, S., Heras, F., Zytnicki, M., Aug. 2005. Existential arc consistency: getting closer to full arc consistency in weighted CSPs. In: Proc. of the 19^{th} IJCAI. Edinburgh, Scotland, pp. 84–89.
[44] Larrosa, J., Schiex, T., Aug. 2003. In the quest of the best form of local consistency for weighted CSP. In: Proc. of the 18^{th} IJCAI. Acapulco, Mexico, pp. 239–244.
[45] Li, C. M., Manyà, F., Planes, J., 2005. Exploiting Unit Propagation to Compute Lower Bounds in Branch and Bound Max-SAT Solvers. In: Proc of CP'2005. No. 3709 in LNCS. Sitges, Spain, pp. 403–414.
[46] Mohr, R., Masini, G., 1988. Good old discrete relaxation. In: Proc. of the 8^{th} ECAI. Munchen FRG, pp. 651–656.
[47] Orlin, J. B., 2009. A faster strongly polynomial time algorithm for submodular function minimization. Mathematical Programming Ser. A 118 (2), 237 – 251.
[48] Régin, J.-C., Petit, T., Bessière, C., Puget, J.-F., Dec. 2001. New Lower Bounds of Constraint Violations for Over-Constrained Problems. In: Proc. of CP-01. No. 2239 in LNCS. Paphos, Cyprus, pp. 332–345.
[49] Sanchez, M., Allouche, D., de Givry, S., Schiex, T., 2009. Russian doll search with tree decomposition. In: Proc. of IJCAI'09. Pasadena (CA), USA.
[50] Sanchez, M., de Givry, S., Schiex, T., 2008. Mendelian error detection in complex pedigrees using weighted constraint satisfaction techniques. Constraints 13 (1), 130–154.
[51] Schiex, T., Sep. 2000. Arc consistency for soft constraints. In: Principles and Practice of Constraint Programming - CP 2000. Vol. 1894 of LNCS. Singapore, pp. 411–424.
[52] Schiex, T., Fargier, H., Verfaillie, G., Aug. 1995. Valued constraint satisfaction problems: hard and easy problems. In: Proc. of the 14^{th} IJCAI. Montréal, Canada, pp. 631–637.
[53] Schlesinger, D., Aug. 2007. Exact Solution of Permuted Submodular MinSum Problems. In: Energy Minimization Methods in Computer Vision and Pattern Recognition. No. 4679/2007 in LNCS. pp. 28–38.
[54] Schlesinger, M., 1976. Sintaksicheskiy analiz dvumernykh zritelnikh signalov v usloviyakh pomekh (Syntactic analysis of two-dimensional visual signals in noisy conditions). Kibernetika 4, 113–130.
[55] Wainwright, M., Jaakkola, T., Willsky, A., 2005. MAP estimation via agreement on (hyper)trees: message passing and linear programming approaches. IEEE Trans. on Information Theory 51 (11),

3697–3717.
[56] Werner, T., Jul. 2007. A Linear Programming Approach to Max-sum Problem: A Review. IEEE Trans. on Pattern Recognition and Machine Intelligence 29 (7), 1165–1179.
[57] Zytnicki, M., Gaspin, C., de Givry, S., Schiex, T., 2009. Bounds Arc Consistency for Weighted CSPs. Journal of Artificial Intelligence Research 35, 593–621.

Appendix A. Infinite loops while trying to enforce VAC

In this section we give an example of a VCSP instance over the valuation structure $\overline{\mathbb{Q}}^+$ for which our VAC algorithm can enter an infinite loop, increasing $c_\varnothing$ by a smaller and smaller amount at each iteration. We present this example to justify our use of heuristics, described in section 11, in our experimental trials. These heuristics guarantee a low-order polynomial time complexity at the cost of not necessarily completely establishing VAC.

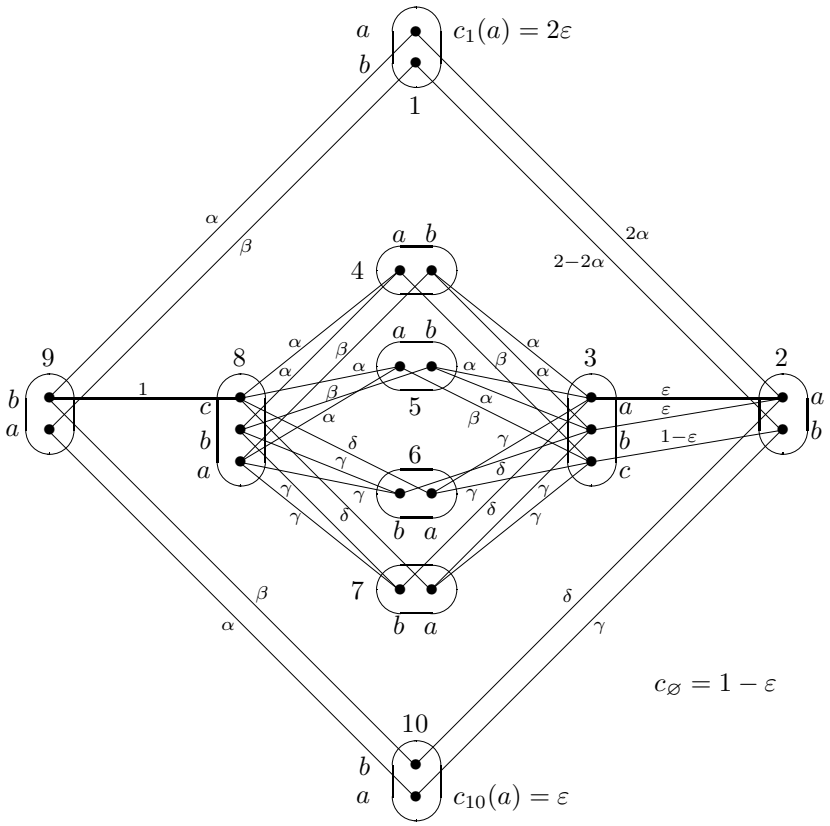

Figure A.14: Example of a VCSP instance for which our VAC algorithm can enter an infinite loop.

Denote by P_i (for all integers $i \geq 0$) the 10-variable VCSP instance shown in Figure A.14 in which the values of α, β, γ, δ, ε are given by

$$\alpha = \tfrac{2}{3}(1 - 4^{-i}) \qquad \beta = 1 - \alpha = \tfrac{1}{3}(1 + 2(4^{-i}))$$
$$\gamma = \tfrac{1}{2}\alpha = \tfrac{1}{3}(1 - 4^{-i}) \qquad \delta = 1 - \gamma = \tfrac{1}{3}(2 + 4^{-i})$$
$$\varepsilon = 4^{-i}$$

A non-zero binary cost $c_{ij}(u, v) = \rho$ is represented by a line labelled ρ joining (i, u) and (j, v). Non-zero unary costs are given explicitly. The original problem P_0 is somewhat simpler than the problem P_i shown in Figure A.14 since, when $i = 0$, $\alpha = \gamma = 0$ and hence

many edges have zero weight. We will show that two iterations of our VAC algorithm can transform P_i into P_{i+1} (for $i \geq 0$). Hence, it is possible that our algorithm enters an infinite loop producing the sequence P_0, P_1, $P_2 \ldots$, and hence never actually establishes virtual arc consistency.

There are different sequences of SAC operations that can be applied to P_i which would allow us to increase $c_\varnothing$. In particular, it is possible to increase $c_\varnothing$ by ε by shifting a weight of ε from $c_{10}(a)$ to variable 1 via variable 9, using the following sequence of SAC operations:

$\mathsf{Extend}(10,a,\{10,9\},\varepsilon)$, $\mathsf{Project}(\{10,9\},9,b,\varepsilon)$,
$\mathsf{Extend}(9,b,\{9,1\},\varepsilon)$, $\mathsf{Project}(\{9,1\},1,b,\varepsilon)$,
$\mathsf{UnaryProject}(1,\varepsilon)$

This sequence of SAC operations immediately produces a VCSP in which $c_\varnothing = 1$. But our VAC algorithm can equally well successively transform P_i into $P_{i+1}, P_{i+2}, \ldots$; in this case $c_\varnothing$ never actually attains the value 1.

Imagine that among the different $c_\varnothing$-increasing sequences of SAC operations that can be applied to P_i, our algorithm determines that we can increase $c_\varnothing$ by an amount $\varepsilon/2$ by shifting weights through the cycle of variables 1,2,3,4,5,8,9,1. In this sequence σ_1 of SAC operations a weight of ε which is extended from $c_1(a)$ towards variable 2 effectively comes back to $c_1(b)$ as a weight of $\varepsilon/2$ since it has to be split into two at variable 3, half being sent towards variable 4 and half towards variable 5. Weights are sent along these two paths (via variables 4 and 5) to variable 8 in order to increase both $c_8(a)$ and $c_8(b)$, which allows the propagation to continue to variable 9 and then variable 1. The sequence σ_1 of SAC operations applied to P_i is given below:

σ_1: $\mathsf{Extend}(1,a,\{1,2\},\varepsilon)$, $\mathsf{Project}(\{1,2\},2,b,\varepsilon)$,
$\mathsf{Extend}(2,b,\{2,3\},\varepsilon)$, $\mathsf{Project}(\{2,3\},3,a,\varepsilon)$, $\mathsf{Project}(\{2,3\},3,b,\varepsilon)$,
$\mathsf{Extend}(3,a,\{3,4\},\varepsilon/2)$, $\mathsf{Extend}(3,b,\{3,4\},\varepsilon/2)$, $\mathsf{Project}(\{3,4\},4,a,\varepsilon/2)$,
$\mathsf{Extend}(3,a,\{3,5\},\varepsilon/2)$, $\mathsf{Extend}(3,b,\{3,5\},\varepsilon/2)$, $\mathsf{Project}(\{3,5\},5,a,\varepsilon/2)$,
$\mathsf{Extend}(4,a,\{4,8\},\varepsilon/2)$, $\mathsf{Project}(\{4,8\},8,a,\varepsilon/2)$,
$\mathsf{Extend}(5,a,\{5,8\},\varepsilon/2)$, $\mathsf{Project}(\{5,8\},8,b,\varepsilon/2)$,
$\mathsf{Extend}(8,b,\{8,9\},\varepsilon/2)$, $\mathsf{Extend}(8,a,\{8,9\},\varepsilon/2)$, $\mathsf{Project}(\{8,9\},9,b,\varepsilon/2)$,
$\mathsf{Extend}(9,b,\{9,1\},\varepsilon/2)$, $\mathsf{Project}(\{9,1\},1,b,\varepsilon/2)$,
$\mathsf{UnaryProject}(1,\varepsilon/2)$

Since the values of α, β and ε (defined above) satisfy the following inequalities

$$\begin{aligned} 2 - 2\alpha &\geq \varepsilon \\ \beta &\geq \varepsilon/2 \\ 1 &\geq \varepsilon \\ 2 - 3\alpha &\geq 3\varepsilon/2 \end{aligned}$$

the above sequence σ_1 of SAC operations produces a VCSP with non-negative costs. Furthermore, $\varepsilon/2$ is the largest increase in $c_\varnothing$ which we can produce by such a sequence due to the fact that $c_{2,3}(a,a) = c_{2,3}(a,b) = \varepsilon$ in the instance P_i. This shows that σ_1 may be the operations actually carried out in one iteration of our VAC algorithm.

Let P_i' denote the VCSP instance which results when the sequence σ_1 of SAC operations is applied to P_i. A sequence of SAC operations can then be applied to P_i' to increase $c_\varnothing$ by $\varepsilon/4$ by shifting weights through the cycle of variables 10,9,8,7,6,3,2,10. This is the rotational symmetry equivalent of the sequence σ_1 of SAC operations with all weights divided by two (and, by rotational symmetry, variable j replaced by variable $11-j$). For completeness, we list the sequence σ_2 of operations below. Again, $\varepsilon/4$ is the largest increase in $c_\varnothing$ which we can produce by such a sequence due to the fact that $c_{9,8}(b,a) = c_{9,8}(b,b) = \varepsilon$ in the instance P_i' and hence σ_2 may be the operations actually carried out by our VAC algorithm.

σ_2: Extend(10,a,$\{10,9\}$,$\varepsilon/2$), Project($\{10,9\}$,9,b,$\varepsilon/2$),
Extend(9,b,$\{9,8\}$,$\varepsilon/2$), Project($\{9,8\}$,8,a,$\varepsilon/2$), Project($\{9,8\}$,8,b,$\varepsilon/2$),
Extend(8,a,$\{8,7\}$,$\varepsilon/4$), Extend(8,b,$\{8,7\}$,$\varepsilon/4$), Project($\{8,7\}$,7,a,$\varepsilon/4$),
Extend(8,a,$\{8,6\}$,$\varepsilon/4$), Extend(8,b,$\{8,6\}$,$\varepsilon/4$), Project($\{8,6\}$,6,a,$\varepsilon/4$),
Extend(7,a,$\{7,3\}$,$\varepsilon/4$), Project($\{7,3\}$,3,a,$\varepsilon/4$),
Extend(6,a,$\{6,3\}$,$\varepsilon/4$), Project($\{6,3\}$,3,b,$\varepsilon/4$),
Extend(3,b,$\{3,2\}$,$\varepsilon/4$), Extend(3,a,$\{3,2\}$,$\varepsilon/4$), Project($\{3,2\}$,2,b,$\varepsilon/4$),
Extend(2,b,$\{2,10\}$,$\varepsilon/4$), Project($\{2,10\}$,10,b,$\varepsilon/4$),
UnaryProject(10,$\varepsilon/4$)

We denote the resulting VCSP instance by P_i'' and its cost functions by $c_\varnothing''$, c_j'', c_{jk}'', with $c_\varnothing$, c_j, c_{jk} denoting the cost functions in P_i. After the two sequences of SAC operations σ_1, σ_2, the new values of the cost functions are given by the following equations.

$$\begin{aligned}
c_\varnothing'' &= c_\varnothing + \varepsilon/2 + \varepsilon/4 \\
c_1''(a) &= c_1(a) - \varepsilon - \varepsilon/2 \\
c_{1,2}''(a,a) &= c_{1,2}(a,a) + \varepsilon \\
c_{1,2}''(b,b) &= c_{1,2}(b,b) - \varepsilon \\
c_{2,3}''(a,a) &= c_{2,3}(a,a) - \varepsilon + \varepsilon/4 \\
c_{2,3}''(b,c) &= c_{2,3}(b,c) + \varepsilon - \varepsilon/4 \\
c_{3,4}''(a,b) &= c_{3,4}(a,b) + \varepsilon/2 \\
c_{3,4}''(c,a) &= c_{3,4}(c,a) - \varepsilon/2 \\
c_{10}''(a) &= c_{10}(a) - \varepsilon/2 - \varepsilon/4 \\
c_{8,7}''(a,b) &= c_{8,7}(a,b) + \varepsilon/4 \\
c_{8,7}''(c,a) &= c_{8,7}(c,a) - \varepsilon/4 \\
c_{8,9}''(c,b) &= c_{8,9}(c,b) - \varepsilon/2 + \varepsilon/2
\end{aligned}$$

For example, $c_\varnothing$ is increased by $\varepsilon/2 + \varepsilon/4$ due to the combined effect of the operations UnaryProject(1,$\varepsilon/2$) and UnaryProject(10,$\varepsilon/4$), and $c_1(a)$ is decreased by $\varepsilon + \varepsilon/2$ as a result of the operations Extend(1,a,$\{1,2\}$,ε) and UnaryProject(1,$\varepsilon/2$). Reading off the cost function values from Figure A.14 (that is $c_\varnothing = 1-\varepsilon$, $c_1(a) = 2\varepsilon$, $c_{1,2}(a,a) = 2\alpha$, $c_{1,2}(b,b) = 2-2\alpha$, $c_{2,3}(a,a) = \varepsilon$, $c_{2,3}(b,c) = 1-\varepsilon$, $c_{3,4}(a,b) = \alpha$, $c_{3,4}(c,a) = \beta$, $c_{10}(a) = \varepsilon$, $c_{8,7}(a,b) = \gamma$, $c_{8,7}(c,a) = \delta$, $c_{8,9}(c,b) = 1$) and given that $\alpha = \frac{2}{3}(1-4^{-i})$,

$\beta = \frac{1}{3}(1 + 2(4^{-i}))$, $\gamma = \frac{1}{3}(1 - 4^{-i})$, $\delta = \frac{1}{3}(2 + 4^{-i})$ and $\varepsilon = 4^{-i}$, we can deduce that

$$\begin{aligned}
c''_{\varnothing} &= 1 - 4^{-i} + 4^{-i}/2 + 4^{-i}/4 = 1 - 4^{-(i+1)} \\
c''_1(a) &= 2(4^{-i}) - 4^{-i} - 4^{-i}/2 = 2(4^{-(i+1)}) \\
c''_{1,2}(a,a) &= \frac{4}{3}(1 - 4^{-i}) + 4^{-i} = \frac{4}{3}(1 - 4^{-(i+1)}) \\
c''_{1,2}(b,b) &= 2 - \frac{4}{3}(1 - 4^{-i}) - 4^{-i} = 2 - \frac{4}{3}(1 - 4^{-(i+1)}) \\
c''_{2,3}(a,a) &= 4^{-i} - 4^{-i} + 4^{-i}/4 = 4^{-(i+1)} \\
c''_{2,3}(b,c) &= 1 - 4^{-i} + 4^{-i} - 4^{-i}/4 = 1 - 4^{-(i+1)} \\
c''_{3,4}(a,b) &= \frac{2}{3}(1 - 4^{-i}) + 4^{-i}/2 = \frac{2}{3}(1 - 4^{-(i+1)}) \\
c''_{3,4}(c,a) &= 1 - \frac{2}{3}(1 - 4^{-i}) - 4^{-i}/2 = \frac{1}{3}(1 + 2(4^{-(i+1)})) \\
c''_{10}(a) &= 4^{-i} - 4^{-i}/2 - 4^{-i}/4 = 4^{-(i+1)} \\
c''_{8,7}(a,b) &= \frac{1}{3}(1 - 4^{-i}) + 4^{-i}/4 = \frac{1}{3}(1 - 4^{-(i+1)}) \\
c''_{8,7}(c,a) &= \frac{1}{3}(2 + 4^{-i}) - 4^{-i}/4 = \frac{1}{3}(2 + 4^{-(i+1)}) \\
c''_{8,9}(c,b) &= 1 - \varepsilon/2 + \varepsilon/2 = 1
\end{aligned}$$

The remaining cost function values can be deduced from these values, since those edges which have identical labels in Figure A.14 are also identical in P''_i. In other words $c''_{2,3}(a,b) = c''_{2,3}(a,a)$ (edges labeled ε in Figure A.14), $c''_{9,10}(a,a) = c''_{9,1}(b,a) = c''_{4,8}(a,c) = c''_{4,8}(a,b) = c''_{5,8}(a,c) = c''_{5,8}(a,a) = c''_{3,5}(b,b) = c''_{3,5}(a,b) = c''_{3,4}(b,b) = c''_{3,4}(a,b)$ (edges labeled α), $c''_{9,10}(b,b) = c''_{9,1}(a,b) = c''_{4,8}(b,a) = c''_{5,8}(b,b) = c''_{3,5}(c,a) = c''_{3,4}(c,a)$ (edges labeled β), $c''_{2,10}(b,a) = c''_{7,3}(a,c) = c''_{7,3}(a,b) = c''_{6,3}(a,c) = c''_{6,3}(a,a) = c''_{8,6}(b,b) = c''_{8,6}(a,b) = c''_{8,7}(b,b) = c''_{8,7}(a,b)$ (edges labeled γ) and $c''_{2,10}(a,b) = c''_{7,3}(b,a) = c''_{6,3}(b,b) = c''_{8,6}(c,a) = c''_{8,7}(c,a)$ (edges labeled δ). Furthermore, all cost function values which were 0 (represented by the absence of an edge in Figure A.14) are also 0 in P''_i.

The above calculations of the cost functions $c''_{\varnothing}$, c''_j, c''_{jk} show that P''_i is, in fact, exactly the VCSP instance P_{i+1}. It follows that, starting from P_0, our algorithm may find the non-ending sequence of VCSP instances $P_0, P_1, P_2, \ldots$ and hence never halt.