Virtual Arc Consistency for Linear Constraints in Cost Function Networks

Pierre Montalbano
Université Fédérale de Toulouse, ANITI, INRAE, UR 875, 31326 Toulouse, France

Simon de Givry
Université Fédérale de Toulouse, ANITI, INRAE, UR 875, 31326 Toulouse, France

George Katsirelos
Université Fédérale de Toulouse, ANITI, INRAE, MIA Paris, AgroParisTech, 75231 Paris, France

Abstract

Solving combinatorial problems with hard and soft constraints has been an active area of research in Artificial Intelligence for several decades. In Constraint Programming (CP), it gives rise either to the development of soft (global) constraints, to the reformulation into a global (integer or continuous) linear/convex program, or to the reformulation into local cost functions representing constraints and preferences in a unified framework. The first approach benefits from a vast catalog of existing (soft) constraints. However, each soft constraint includes its own preference representation and a dedicated propagator (e.g., a knapsack constraint with assignment costs) that communicates with other soft constraints only through the variable domains, which results in weak lower bounds in minimization problems. Conversely, the second approach provides a global view with strong lower bounds, but the size of the reformulation can be a critical issue when computing bounds (e.g. in Computational Protein Design). Here, we focus on the third approach, within the framework of Cost Function Networks (CFNs) with so-called soft arc consistency algorithms producing lower bounds of intermediate quality between the first two approaches. Recently, the introduction of linear constraints as local cost functions increases the modeling applicability in CFNs. In this work, we adapt an existing soft arc consistency algorithm called Virtual Arc Consistency (VAC) to take into account linear constraints. We call it VAC-lin. In the experimental results, we show that VAC-lin significantly improves lower bounds compared to the original VAC algorithm on the MIPLIB 2017 and XCSP benchmarks. This always helps reduce the initial optimality gap, which is valuable information for a user, and in some cases, it greatly reduces the solving time.

2012 ACM Subject Classification discrete optimization

Keywords and phrases weighted constraint satisfaction problem, linear inequalities, bounds consistency, multiple-choice knapsack problem, integer programming

Digital Object Identifier 10.4230/LIPIcs.Soft.2024.

1 Introduction

Graphical models provide a powerful framework to model combinatorial problems of different natures answering various tasks, going from satisfaction problems to probabilistic models [6]. It employs local functions defined over 'small' subset of variables to represent diverse interactions between the variables. For example, to model the Constraint Satisfaction Problem (CSP) [24], each local function is a constraint evaluating to true (satisfied) or false (falsified). Here we are interested in Cost Function Networks (CFN) where each local function is a cost function evaluating a cost, the task of finding the assignment minimizing the sum of all cost functions is known as the Weighted Constraint Satisfaction Problem (WCSP). Most methods to find optimal solutions rely on a branch and bound procedure relying either on static memory-intensive bounds [11] or on memory-light ones [7] to compute lower bounds. Here, we focus on the latter, known as *Soft Arc Consitency* (SAC) algorithms, because similarly to CSP propagation, they reason on each non-unary cost function individually. Different levels

First Workshop on Discrete Optimization with Soft Constraints (Soft 2024).
Editor: Javier Larrosa and Simon de Givry; Article No. ; pp. :1–:19
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

of SAC exist, each offering a trade-off between strength of propagation (quality of the lower bound) and time to propagate. Finding the correct balance between the quality of derived lower bounds and the time to construct them is crucial to achieving efficiency. *Virtual Arc consistency* (VAC) [7] is a strong level of consistency, it can derive a strong lower bound but can be expensive to enforce. The principle of VAC is to study a CSP Bool(P) derived from a WCSP P. For every cost function, only the tuples and values having a zero cost are allowed in Bool(P). If Bool(P) is inconsistent then the lower bound of P can be increased. If the inconsistency of Bool(P) is detected by Generalized Arc Consistency (GAC), then VAC has been designed to extract a lower bound.
CFN also benefits from the flexibility of the Constraint Programming (CP) with its ability to handle (soft)-global constraints. However, while integrating a global constraint in a CP solver only requires an algorithm to prune inconsistent values, in CFN, in addition to the pruning, propagators for new constraints must also be able to compute a lower bound. This has been done for various global constraints including AllDifferent, clique, and linear constraints [2, 10, 21].

Contributions. Motivated by the good performance of VAC and the new introduction of linear constraints in CFN, we study here how to join those works. Previous approaches handling linear constraints in CFNs tend to absorb unary costs when propagated individually, which can no longer be exploited by other propagation. Enforcing VAC allows finding a sequence of cost moves involving different propagation and makes communication between linear constraints possible. This could greatly increase the computed lower bounds. However, enforcing VAC on a linear constraint requires keeping in Bool(P) only the values that can be part of a zero-cost tuple. For linear constraints, it requires solving a problem similar to the Knapsack problem and thus is NP-complete. We show how we can use reduced costs filtering [13] to detect a subset of inconsistent values. This leads to VAC-lin which enforces an incomplete GAC on Bool(P). This approach is implemented in toulbar2 and tested on several benchmarks.

2 Background

2.1 Weighted Constraint Satisfaction Problem

▶ **Definition 1.** *A Cost Function Network (CFN) P is a tuple $(\boldsymbol{X}, \boldsymbol{D}, \boldsymbol{C}, \top)$ where $\boldsymbol{X}$ is a set of variables, with finite domain $\boldsymbol{D}_i$ for $i \in \boldsymbol{X}$. $\boldsymbol{C}$ is a set of constraints. Each constraint $c_{\boldsymbol{S}} \in \boldsymbol{C}$ is defined over a subset of variables $\boldsymbol{S}$ called its scope ($\boldsymbol{S} \subseteq \boldsymbol{X}$). $\top$ is a maximum cost indicating a forbidden assignment.*

We denote by (i, v) the value $v \in \boldsymbol{D}_i$ of variable $i \in \boldsymbol{X}$. The size of the scope of a constraint is its arity. Unary (resp. binary) cost functions have arity 1 (resp. 2). In this paper, we assume exactly one unary constraint exists for each variable. Let $\boldsymbol{S} \subseteq \boldsymbol{X}$ be a subset of variables, we denote by $\ell(\boldsymbol{S})$ the Cartesian product $\Pi_{i \in \boldsymbol{S}} \boldsymbol{D}_i$ of the domains of the variables in $\boldsymbol{S}$. An assignment (or tuple) $\tau \in \ell(\boldsymbol{S})$ is an assignment of all the variables $i \in \boldsymbol{S}$ to a value of its domain $\boldsymbol{D}_i$. If $\boldsymbol{S} = \boldsymbol{X}$ then τ defines a *complete assignment*, otherwise it is a *partial assignment*. A constraint over a scope $\boldsymbol{S}$ is denoted $c_{\boldsymbol{S}}$. The cost of a tuple $\tau \in \ell(\boldsymbol{S})$ for a constraint $c_{\boldsymbol{S}}$ is denoted $c_{\boldsymbol{S}}(\tau)$. Without loss of generality, we assume all costs are positive integers, bounded by $\top$, a special constant signifying infeasibility. Hence if $c_{\boldsymbol{S}}(\tau) = \top$ then the tuple τ is not a feasible. A constraint $c_{\boldsymbol{S}}$ is hard if for all $\tau \in \ell(\boldsymbol{S})$, $c_{\boldsymbol{S}}(\tau) \in \{0, \top\}$, otherwise it is soft. A CFN P that contains only hard constraints is a constraint network

(CN). In the following, we use the term *cost function* interchangeably with the term constraint. The cost of a complete assignment $\tau \in \ell(\boldsymbol{X})$ is given by $c_P(\tau) = \sum_{c_{\boldsymbol{S}} \in \boldsymbol{C}} c_{\boldsymbol{S}}(\tau)$. The Weighted Constraint Satisfaction Problem (WCSP) asks, given a CFN P, to find a complete assignment τ minimizing $c_P(\tau)$. This task is NP-hard [8]. When the underlying CFN is a CN, the problem is a CSP. In the following, we use WCSP to refer both to the optimization task and the underlying CFN.

Each cost function is either represented in *extension* or in *intention*. A cost function represented in extension, also known as a table constraint, explicitly lists all the tuples and their associated costs. Only low arity cost functions can be written in extension within a reasonable memory size limit because the number of tuples grows exponentially with arity. A cost function given in intention, is defined by a function or a logical expression that specifies the relationship between the variables, for example, global constraints are typically given in intention.

We also assume the existence of a constraint $c_\varnothing$ with empty scope, which represents a constant in the objective function and, since there exist no negative costs, it is a lower bound on the cost of all possible assignments. $c_\varnothing$ will play a primary role in SAC algorithms.

2.2 Soft Arc Consistency

Soft Arc Consistency (SAC) algorithms sequentially examine small subsets of cost functions. On top of removing the locally inconsistent values, it computes a lower bound by increasing $c_\varnothing$. To achieve this they rely on the notion of reparametrization: a reparameterization P' of a WCSP P is a WCSP with an identical structure, i.e., the set of scopes and variables are identical. The costs assigned by each individual cost function may differ, but $c_P(\tau) = c_{P'}(\tau)$ for all complete assignments τ. We say that a reparametrization is better if it has a higher $c_\varnothing$. A reparametrization can be obtained through a sequence of local *Equivalence Preserving Transformations* (EPTs). Let $\boldsymbol{S}_1 \subset \boldsymbol{S}_2$ be two scopes with corresponding cost functions $c_{\boldsymbol{S}_1}$ and $c_{\boldsymbol{S}_2}$. Procedure MoveCost describes how a cost α moves between the corresponding cost functions.

As a matter of terminology, when $\alpha > 0$, cost moves from the larger arity cost function $c_{\boldsymbol{S}'}$ to the smaller arity $c_{\boldsymbol{S}}$ and the move is called a *projection*, denoted *project*$(c_{\boldsymbol{S}}, c_{\boldsymbol{S}'}, \tau, \alpha)$ with $\tau \in \ell(\boldsymbol{S})$.When $\alpha < 0$, cost moves to the larger arity cost function $c_{\boldsymbol{S}'}$ and the move is called an *extension*, denoted *extend*$(c_{\boldsymbol{S}}, \tau, c_{\boldsymbol{S}'}, -\alpha)$.When $\boldsymbol{S} = \emptyset$ and $|\boldsymbol{S}'| = 1$, with $\boldsymbol{S}' = \{i\}$, the move is called a *unary projection*, denoted *unaryProject*(c_i, α), equivalent to MoveCost$(c_\varnothing, c_i, \emptyset, \alpha)$. We never perform extensions from $c_\varnothing$, so it monotonically increases during the run of an algorithm and as we descend a branch of the search tree.

Finding which cost moves lead to an optimal reparameterization, which means one that derives the optimal increase in the lower bound, is not obvious. It has been shown that any

Procedure MoveCost$(c_{\boldsymbol{S}_1}, c_{\boldsymbol{S}_2}, \tau_1, \alpha)$: Move α units of cost between the tuple τ_1 of scope $\boldsymbol{S}_1$ and tuples τ_2 that extend τ_1 in scope $\boldsymbol{S}_2$

```
Data: Scopes S1 ⊂ S2
Data: τ1 ∈ ℓ(S1)
Data: cost α to move
c_S1(τ1) ← c_S1(τ1) + α ;
foreach τ2 ∈ ℓ(S2) | τ2[S1] = τ1 do
    c_S2(τ2) ← c_S2(τ2) − α ;
```

reparameterization can be derived by a set of local cost moves [16] and that the optimal reparameterization (with α rational) – and, equivalently, the optimal set of cost moves – can be found from the optimal dual solution of a linear relaxation of the WCSP [7], whose feasible region is called the *local polytope*.

However, solving this LP to optimality is often prohibitively expensive because the worst-case complexity of an exact LP algorithm is $O((er+e^2)\sqrt{e})$ [34], where e is the number of cost functions and r the largest arity. The poor asymptotic complexity matches empirical observation [15]. Moreover, the particular structure of this LP does not allow for a more efficient solving algorithm, as it has been shown that solving LPs of this form is as hard as solving any LPs [23]. Instead, work has focused on producing good but potentially suboptimal feasible dual solutions. Various algorithms have been proposed for this, like Block-Coordinate Ascent (BCA) algorithms developed for image analysis [16, 35, 30, 17, 29, 31] or *soft arc consistencies* in constraint programming [26, 18, 9, 36, 7]. Notably, the strongest algorithms from both lines of research, such as TRWS [16] and VAC [7] converge on fixpoints with the same properties.

Here, we are interested in soft arc consistency (SAC) and we define some of them.

▶ **Definition 2.** *A WCSP P is Node Consistent (NC) [18] if for every variable $i \in \boldsymbol{X}$ there exists a value $v \in \boldsymbol{D}_i$ such that $c_i(v) = 0$ and for every value $v' \in \boldsymbol{D}_i$, $c_\varnothing + c_i(v') < \top$.*

In the following, we assume that a WCSP is NC before our propagator runs.

An important SAC algorithm for this paper is *Virtual Arc Consistency* (VAC) [7]. It relies on a particular CSP Bool(P) that can be derived from a WCSP instance P. For every cost function in P, except $c_\varnothing$, only the tuples and values having a zero cost are allowed in Bool(P). Any satisfying assignment of Bool(P) is also feasible for P and by construction has cost $c_\varnothing$, hence that is an optimum assignment of P. On the other hand, if Bool(P) is infeasible, no such assignment exists and the optimum of P has a cost strictly greater than $c_\varnothing$. It has been shown [7] that an infeasibility certificate produced by arc consistency on Bool(P) can be used to derive a reparameterization of P with increased $c_\varnothing$, which is not the case for other cases of infeasibility.

In the following, $AC(P)$ denotes the *arc consistent closure* of a CSP P, the unique CSP that results from removing arc inconsistent values from domains. An empty AC closure implies infeasibility.

▶ **Definition 3** (Virtual Arc Consistency [7])**.** *A WCSP P is virtual arc consistent if the (generalized) arc consistency closure of the CSP* Bool(P) *is non-empty.*

▶ **Theorem 4** ([7])**.** *Let P be a $WCSP$ such that $c_\varnothing < \top$. Then there exists a sequence of EPTs which when applied to P leads to an increase in $c_\emptyset$ if and only if the arc consistency closure of* Bool(P) *is empty.*

The algorithm to enforce VAC can be decomposed into 3 phases:

1. Establish (G)AC on Bool(P). If no conflict occurred, then quit.
2. Given a conflict, perform *conflict analysis*[1] on it to compute a maximal cost λ and corresponding sequence of EPTs σ such that applying σ increases $c_\varnothing$ by λ.
3. Apply σ to P and go back to phase 1.

[1] This is intentionally similar to the term used in SAT, because it uses a post-conflict, reverse chronological order traversal of the operations performed by propagation.

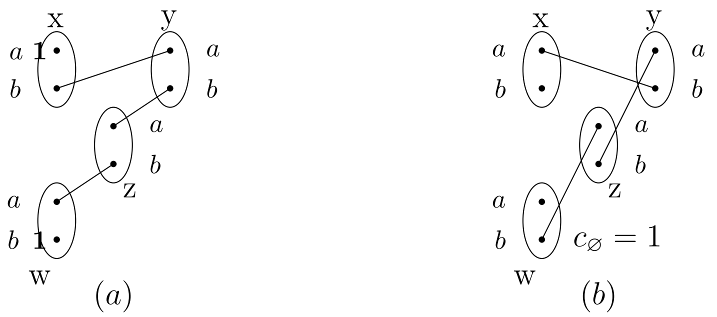

Figure 1 (a) A WCSP with 4 Boolean variables, an edge indicates a cost of 1. (b) An equivalent WCSP verifying VAC.

To see why step 2 is always possible, observe that arc consistency operations in $Bool(P)$ can themselves be viewed as EPTs where the cost moved is always $\top$. For example, pruning a value (i, a) which has lost all supports in constraint c_{ij} can be viewed as extending $\top$ from each support (j, b) of (i, a) in j, which marks all supporting tuples of (i, a) in c_{ij} as forbidden, then projecting $\top$ from c_{ij} to (i, a). If we choose a cost λ small enough, we can repeat those EPTs in P using λ instead of $\top$, so that no negative costs are introduced. The purpose of step 2 then is to identify a maximal value for λ.

From the above, we see that as long as Bool(P) has an empty arc consistency closure, VAC will increase $c_\varnothing$. An additional heuristic variant of VAC that we consider here is VAC$_\theta$. This uses a threshold θ when creating Bool$_\theta$(P) and forbids only the values/tuples with a cost greater than or equal to θ. When $\theta = 1$, VAC$_\theta$ is equivalent to VAC. Clearly, VAC$_\theta$ may discover a subset of the reparameterizations that can be found by VAC. But the higher θ is, the higher the costs that are involved in conflicts discovered by GAC in Bool$_\theta$(P), hence there is a chance that those lead to a higher increase of $c_\varnothing$, although this cannot be guaranteed. On the other hand, the lower θ is, the better the chance that $Bool_\theta(P)$ actually has an empty AC closure. So VAC$_\theta$ is applied by starting with high values for θ in order to quickly increase the lower bound, and gradually decrease it.

The algorithm to enforce VAC is strongly impacted by the size of the cost functions, its time complexity is $O(ned^r)$ per iteration, where n is the number of variables, e the number of cost functions, d the largest domain and r the largest arity. In the presence of global constraints, a dedicated algorithm is required to enforce a possibly weaker consistency.

▶ **Example 5.** Let P be a WCSP with 4 variables x, y, z, w with domains $\{a, b\}$ as depicted in Figure 1 (a). The AC closure of $Bool(P)$ is empty, indeed, values (x, a) and (w, b) are directly removed from Bool(P) because $c_x(a) = c_y(b) > 0$. Consequently, value (y, a) has no support on c_{xy} and (z, b) has no support on c_{zw}, those values can be removed. Finally, (y, b) has no support on c_{yz} and a domain wipe-out occurs at variable y. By analyzing the trace that led to this conflict, VAC produces the following sequence of EPTs and obtains the WCSP verifying VAC depicted in Figure 1 (b).

1) $extend(c_x, a, c_{xy}, 1)$	5) $extend(c_z, b, c_{yz}, 1)$
2) $extend(c_w, b, c_{zw}, 1)$	6) $project(c_y, c_{yz}, b, 1)$
3) $project(c_y, c_{xy}, a, 1)$	7) $unaryProject(y, 1)$
4) $project(c_z, c_{zw}, b, 1)$	

2.3 Linear Constraints

Linear constraints are global constraints capturing a linear interaction between variables. They are expressive and compact and used in a wide range of optimization problems including computer science, operations research, and artificial intelligence [3]. We consider linear inequality constraints of the form: $\sum_{i \in \boldsymbol{S}} \sum_{v \in \boldsymbol{D}_i} w_{iv} x_{iv} \geq C$, where $c_{\boldsymbol{S}} \in \boldsymbol{C}$, x_{iv} is a 0/1 variable taking value 1 when the domain value $v \in \boldsymbol{D}_i$ is assigned to variable $i \in \boldsymbol{S}$. Without loss of generality, we assume the weights w_{iv} and capacity C are positive constants. Any linear constraint can be written in that form. We consider hard linear constraints, i.e., for any assignment $\tau \in \ell(\boldsymbol{S})$ that satisfies the constraint, it holds that $c_{\boldsymbol{S}}(\tau) = 0$, otherwise $c_{\boldsymbol{S}}(\tau) = \top$.

If EPTs involve a linear constraint, the cost of the allowed tuples is modified, and we might get $0 < c_{\boldsymbol{S}}(\tau) < \top$. Recent work introduced a way to represent and propagate linear constraints in a WCSP solver [21] through so-called *delta costs*. A cost δ_{iv} is associated with each value $i \in \boldsymbol{S}, v \in \boldsymbol{D}_i$, and it captures the amount of costs moved from the unary cost functions to the linear constraints. A cost move from $c_i(v)$ to the linear constraint increases δ_{iv}, while a cost move in the opposite direction decreases it. Hence, we can have negative δ costs. We represent by $\delta_\emptyset$ the cost moved from this constraint to $c_\varnothing$. This quantity is necessarily positive. After any sequence of EPTs, the cost of an assignment τ is defined by:

$$c_{\boldsymbol{S}}(\tau) = \begin{cases} \sum_{i \in \boldsymbol{S}} \delta_{i\tau[i]} - \delta_\emptyset & \text{if } \tau \text{ satisfies the constraint} \\ \top & \text{otherwise} \end{cases} \tag{1}$$

Initially, no cost moves have been performed and all the δ costs are 0.
We show how to approach the enforcement of Full $\emptyset$-*Inverse Consistency* (F$\emptyset$IC) on linear constraints.

▶ **Definition 6.** *A WCSP is Full $\emptyset$-Inverse Consistent (F$\emptyset$IC) if for every cost function $c_{\boldsymbol{S}} \in \boldsymbol{C}$ there exists $\tau \in \ell(\boldsymbol{S})$ such that $c_{\boldsymbol{S}}(\tau) + \sum_{i \in \boldsymbol{S}} c_i(\tau[i]) = 0$.*

This can be done by propagating the linear constraints one by one, and each time solving the linear relaxation of the following 0/1LP representation of one linear constraint and the associated unary costs.

$$\min \textstyle\sum_{i \in \boldsymbol{S}, v \in \boldsymbol{D}_i} (\delta_{iv} + c_i(v)) x_{iv} - \delta_\emptyset \tag{2a}$$
$$\textstyle\sum_{i \in \boldsymbol{S}, v \in \boldsymbol{D}_i} w_{iv} x_{iv} \geq C \tag{2b}$$
$$\textstyle\sum_{v \in \boldsymbol{D}_i} x_{iv} = 1, \qquad \forall i \in \boldsymbol{S} \tag{2c}$$
$$x_{iv} \in \{0, 1\}, \qquad \forall i \in \boldsymbol{S}, v \in \boldsymbol{D}_i \tag{2d}$$

For a linear constraint $c_{\boldsymbol{S}}$, let this problem be $\mathcal{P}_{\boldsymbol{S}}$. This problem is the Multiple-Choice Knapsack Problem (MCKP) [22] and its linear relaxation $\mathcal{LP}_{\boldsymbol{S}}$ can be solved efficiently. From the optimal dual solution it is possible to derive a sequence of EPTs increasing $c_\varnothing$ by the cost of the optimal solution. In the following we will also need to find the minimal cost tuple of one linear constraint alone, thus we define $\widetilde{\mathcal{P}}_{\boldsymbol{S}}$ (resp $\widetilde{\mathcal{LP}}_{\boldsymbol{S}}$) as the 0/1LP (resp LP) with modified objective $\min \sum_{i \in \boldsymbol{S}, v \in \boldsymbol{D}_i} \delta_{iv} x_{iv} - \delta_\emptyset$. We can observe that any dual solution of $\widetilde{\mathcal{LP}}_{\boldsymbol{S}}$ is also a dual solution of $\mathcal{LP}_{\boldsymbol{S}}$. A dual solution can be feasible for $\mathcal{LP}_{\boldsymbol{S}}$ and infeasible for $\widetilde{\mathcal{LP}}_{\boldsymbol{S}}$, yet its cost remains the same in both problems. More interestingly, performing sensitivity analysis on both problems provides different information. For example, given a dual solution $\mathbf{y}$, the reduced cost of x_{ia} is defined by the slack of its corresponding dual constraint. This value can be interpreted as a lower bound on the difference of objective value between any feasible solution with $x_{ia} > 0$ and the optimal solution. In the context of CFN, the reduced cost of a variable x_{ia} computed in $\widetilde{\mathcal{LP}_{\boldsymbol{S}}}$ from a dual solution $\mathbf{y}$, denoted $rc^{y}_{\boldsymbol{S}}(i,a)$, gives a lower bound on the minimal cost tuple $\tau \in \ell(\boldsymbol{S})$ in $c_{\boldsymbol{S}}$ verifying $\tau[i] = a$. In the following, since we manipulate only one dual solution $\mathbf{y}$ at a time, we omit $\mathbf{y}$ and write $rc_{\boldsymbol{S}}(i,a)$. Computing reduced costs from problem $\mathcal{LP}_{\boldsymbol{S}}$ gives information on the minimal cost tuple when combining $c_{\boldsymbol{S}}$ with the unary costs. They have been exploited in [21] to enforce $F\emptyset IC$ on linear constraints, however, they are not suited for VAC as it mainly relies on Bool(P) where unary costs are either 0 or $\top$.

3 VAC on Linear Constraints

One problem with the propagation method for linear constraints introduced in [21], is that the constraints are propagated one by one and only communicate with unary costs. Therefore, once a linear constraint has absorbed a cost, it becomes invisible to other cost functions. Moreover, the quality of the lower bound depends largely on the propagation order. Enforcing VAC allows the detection of a sequence of EPTs resulting from a combination of several constraint propagation, without a fixed propagation order. However, in our case, VAC requires enforcing GAC on linear constraints in Bool(P). From equation (1) we know that a linear constraint $c_{\boldsymbol{S}}$ allows in Bool(P) only the tuples $\tau \in \ell(\boldsymbol{S})$ satisfying the linear constraint and $\sum_{\tau[i]=v}(\delta_{iv}) - \delta_\emptyset = 0$. Hence, deciding if a variable is GAC with respect to a linear constraint demands solving a knapsack problem, which is an NP-complete task. Moreover, for each removal, we must be able to provide an *explanation* as VAC needs one to trace-back the removal. An explanation for the removal of value (i,a) is a set of values whose removal implies the removal of (i,a) by arc consistency on a constraint $c_{\boldsymbol{S}}$. We say that an explanation is minimal if none of its subsets is an explanation. We show we can use domain propagation, a dual optimal solution, and reduced cost filtering [13] to detect a subset of the inconsistent tuples in Bool(P) and to produce explanations.

3.1 Filtering for Linear Constraints with Assignment Costs

We show here how to use linear constraints within VAC$_\theta$, rather than the base VAC.

Enforcing GAC in Bool$_\theta$(P) requires to verify for each $\boldsymbol{S} \in \boldsymbol{C}, i \in \boldsymbol{S}, a \in \boldsymbol{D}_i$ if there exists a tuple $\tau \in \ell(\boldsymbol{S}), \tau[i] = a$ such that $c_{\boldsymbol{S}}(\tau) < \theta$. Specifically, each linear constraint is transformed to the following hard constraint in Bool$_\theta$(P). Note that we slightly abuse notation here and use Bool$_\theta$(c$_\text{S}$) to denote the hard constraint that corresponds to the constraint $c_{\boldsymbol{S}}$ in P.

$$\text{Bool}_\theta(\text{c}_\text{S})(\tau) = \begin{cases} 0 & \text{if } \tau \text{ satisfies the constraint and } \sum_{\tau[i]=v} \delta_{iv} - \delta_\emptyset < \theta \\ \top & \text{otherwise} \end{cases} \tag{3}$$

Propagating this requires filtering the Knapsack problem depicted by $\mathcal{P}_{\boldsymbol{S}}$. This is NP-complete, but has been studied before. Algorithms for it include dynamic programming for enforcing GAC [32], approximate filtering with a *fully polynomial time approximation scheme* [27, 28], and linear programming-based filtering [13, 5]. Here, we use linear programming and show how to perform filtering as well as generate explanation, which is needed in order to integrate the constraint into VAC.

Given a linear constraint $c_{\boldsymbol{S}}$, we say that the removal of a value (i,a) is *hard* if there exists no remaining feasible tuple with value a assigned to variable i i.e $\forall \tau \in \ell(\boldsymbol{S}), \tau[i] = a$

we have $c_{\boldsymbol{S}}(\tau) = \top$. Otherwise the removal is *soft* i.e $\forall \tau \in \ell(\boldsymbol{S}), \tau[i] = a$ we have $c_{\boldsymbol{S}}(\tau) \geq \theta$. The strategies to detect and explain a hard/soft removal are different. Hard removal can be detected by enforcing domain consistency. This can be done in linear time and for each removal, a minimal explanation is produced using conflict explanation [14].

In order to detect if a value (i, a) can be soft removed from $Bool_\theta(P)$, we solve for each linear constraint $\boldsymbol{S}$ a variant of $\mathcal{P}_{\boldsymbol{S}}$ (resp $\widetilde{\mathcal{P}}_{\boldsymbol{S}}$), in which the removed values (j, b) are not explicitly forbidden but rather penalized in the objective function by attributing to them a new unary cost $c_j(b) = \top$. We refer to this first variant as $\mathcal{P}'_{\boldsymbol{S}}$ (resp $\widetilde{\mathcal{P}}'_{\boldsymbol{S}} = \widetilde{\mathcal{P}}_{\boldsymbol{S}}$). As we want to find the minimal tuple using (i, a), we also fix $\forall v \in \boldsymbol{D}_i, v \neq a, \delta_{iv} = \top$. We refer to this second variant as $\mathcal{P}^{ia}_{\boldsymbol{S}}$ (resp $\widetilde{\mathcal{P}}^{ia}_{\boldsymbol{S}}$). The use of cost $\top$ for removed values guarantees that those values will not appear in an optimal (relaxed) solution because their costs are too high, but it does so without modifying the primal constraints. Thus, the only difference between $\mathcal{P}_{\boldsymbol{S}}, \mathcal{P}'_{\boldsymbol{S}}, \mathcal{P}^{ia}_{\boldsymbol{S}}, \widetilde{\mathcal{P}}_{\boldsymbol{S}}$ and $\widetilde{\mathcal{P}}^{ia}_{\boldsymbol{S}}$ is the objective function. In particular, an optimal relaxed dual solution $\mathbf{y}$ of $\mathcal{LP}^{ia}_{\boldsymbol{S}}$ is also a valid (nonoptimal) dual solution of $\widetilde{\mathcal{LP}}^{ia}_{\boldsymbol{S}}$ and it provides a lower bound on $\mathcal{P}^{ia}_{\boldsymbol{S}}$.

If the optimal solution of $\mathcal{P}^{ia}_{\boldsymbol{S}}$ has a cost greater than θ then (i, a) can be removed from $\text{Bool}_\theta(\text{P})$. Solving such a problem is NP-complete, but we can obtain partial filtering by solving its relaxation $\mathcal{LP}^{ia}_{\boldsymbol{S}}$ which can be done quite efficiently [22]. To compute an explanation it seems natural to perform dual sensitivity analysis and figure out how the problem behaves without the filtering made in $\text{Bool}_\theta(\text{P})$. However, computing a dual solution $\mathbf{y}$ of $\mathcal{LP}^{ia}_{\boldsymbol{S}}$ and analyzing $\mathcal{LP}^{ia}_{\boldsymbol{S}}$ with it does not provide any useful information because the $\top$ unary costs appearing in the objective function perturb the reduced costs. Those costs don't appear in $\widetilde{\mathcal{LP}}^{ia}_{\boldsymbol{S}}$, hence we compute the reduced costs and an explanation from $\widetilde{\mathcal{LP}}^{ia}_{\boldsymbol{S}}$ and the dual solution $\mathbf{y}$ obtained from $\mathcal{LP}^{ia}_{\boldsymbol{S}}$. Let OPT_{ia} be the optimal relaxed solution of $\mathcal{LP}^{ia}_{\boldsymbol{S}}$, it gives a lower bound on the minimal tuple with $x_{ia} = 1$ in $\mathcal{P}^{ia}_{\boldsymbol{S}}$. If a value (j, b) verifies $rc_{\boldsymbol{S}}(j, b) \geq 0$, then we know that the minimal cost tuple with $x_{ia} = 1$ and $x_{jb} = 1$ costs more than OPT_{ia}, and thus it is not part of a minimal explanation for the removal of (i, a). Otherwise, in our context a value (j, b) having a negative reduced cost indicates that if (j, b) is allowed ($c_j(b) < \top$) then the cost of a tuple with $x_{ia} = 1, x_{jb} = 1$ could be lower than OPT_{ia}, and $rc_{\boldsymbol{S}}(j, b)$ gives the value of the decrease compared to OPT_{ia}. More precisely, if $rc_{\boldsymbol{S}}(j, b) + OPT_{ia} < \theta$ we know that (j, b) is necessarily in the explanation. However, this is not a necessary condition. Indeed, the reduced costs only give a bound on the change of the objective, thus it is possible that there exist $(j, b), (k, c)$ such that $rc_{\boldsymbol{S}}(j, b) + OPT_{ia} \geq \theta$ and $rc_{\boldsymbol{S}}(k, c) + OPT_{ia} \geq \theta$ but the cost of any tuple with both $x_{1a} = 1$, $x_{jb} = 1$, and $x_{kc} = 1$ is $< \theta$. Accounting for this is possible, but only at the cost of additional computation. Instead, we choose to compute a potentially non-minimal explanation with all values (j, b) such that $rc_{\boldsymbol{S}}(j, b) < 0$.

This procedure detecting soft removable values requires solving one LP for each possible value, which is too costly when filtering $Bool_\theta(P)$. We can detect a subset of those values by solving only one time $\mathcal{LP}'_{\boldsymbol{S}}$ and relying on *reduced cost based filtering* [13, 5]. Suppose the optimal relaxed cost of $\mathcal{LP}'_{\boldsymbol{S}}$ is OPT and we compute the reduced costs associated with $\widetilde{\mathcal{LP}}_{\boldsymbol{S}}$ and the dual optimal solution from $\mathcal{LP}'_{\boldsymbol{S}}$. For a value (j, b) such that $rc_{\boldsymbol{S}}(j, b) + OPT \geq \theta$, the minimum cost of a tuple that contains it is at least θ and can be removed from $\text{Bool}_\theta(\text{P})$. For each removal, we can only compute a straightforward explanation containing all the previously removed values. Obtaining a dedicated explanation for a removal requires to solve a new LP as in the previous strategy. Solving $\mathcal{LP}'_{\boldsymbol{S}}$ can also help to directly detect if the linear constraint is conflicting in $\text{Bool}_\theta(\text{P})$ i.e $OPT \geq \theta$. In this case, we can also produce an explanation by analyzing the reduced costs of $\widetilde{\mathcal{LP}}_{\boldsymbol{S}}$.

3.2 VAC-lin Subroutines

Here, we define VAC-lin, a local consistency obtained by enforcing the filtering process described in Section 3.1 on the linear constraints of $\mathrm{Bool}_\theta(\mathrm{P})$ and GAC on the other constraints. Any value removed by filtering techniques described in Section 3.1 would have been removed by enforcing GAC on linear constraints. Thus, the following corollary of theorem 4 is true.

▶ **Corollary 7.** *Let P be a $WCSP$ such that $c_\emptyset < \top$. If enforcing reduced costs filtering and domain consistency on the linear constraints of* $\mathrm{Bool}_\theta(\mathrm{P})$*, and GAC on the other constraints leads to a conflict, then there exists a sequence of soft arc consistency operations which when applied to P leads to an increase in $c_\emptyset$.*

Enforcing VAC-lin can be done by plugging what is presented in Section 3.1 in the VAC algorithm. Specifically, propagation for linear constraints in $Bool_\theta(P)$ is performed in function `VAC-Filter` and tracing in function `VAC-Tracer`.

Filtering Phase

VAC-lin considers $\mathrm{Bool}_\theta(\mathrm{P})$ and applies an incomplete GAC on linear constraints and GAC on other constraints. It uses a queue R containing the constraints that need to be propagated. Initially, R contains every possible pair (line 11), then whenever a value is removed, all the constraints linked to this value need to be propagated again (line 22). It ends when a conflict appears or no more values can be removed (line 21, 6). Whenever a value (i, a) is removed because it has no support on constraint $c_{\boldsymbol{S}}$, this value is added to a queue Q (line 19), the constraint $c_{\boldsymbol{S}}$ and an explanation is recorded in the killer structure (line 20). For a linear constraint $c_{\boldsymbol{S}}$, the filtering process is given in function `LinFilter`, and enforces an incomplete GAC based on Section 3.1. It begins by enforcing domain consistency and solving $\mathcal{LP}'_{\boldsymbol{S}}$. If it is conflicting (optimal cost $\geq \theta$ or unfeasible constraint) then it produces an explanation and goes to the next phase. Otherwise, it studies the reduced costs to remove inconsistent values. To avoid extra computational work, a straightforward explanation containing all the previously removed values is recorded in killer for each removal. A dedicated explanation will be computed in the next phase only when it is required.
To limit the computation time of VAC-lin, we prioritize applying GAC on table constraints and then incomplete GAC on linear constraints.

Tracing Phase

In the second phase, we want to trace back the operations leading to a conflict in $\mathrm{Bool}_\theta(\mathrm{P})$ and collect a minimal subset of value deletions that is sufficient to explain it. We use a Boolean function M to mark the values with a zero cost in P necessary to explain the conflict. Our objective is to identify a set of values/tuples with non-zero costs that can be used as a source to move costs to the marked values. Initially, only the values in the explanation returned by the filtering process are marked (line 20). Let (i, a) be a marked value ($M(i, a) = true$). We can find a source for this value by studying an explanation for this removal. The filtering phase gave a first explanation and stored it in killer(i, a). The function `Find-Explanation`$_{ia}$`&OPT` tries to refine this explanation. For linear constraints, this is done by `LinExplanation`$_{ia}$. Let the pair $\langle E, c_{\boldsymbol{S}}\rangle$ be a set of value removals and cost function, respectively, explaining the removal of (i, a), in the sense that if we can move costs from the values in E to $c_{\boldsymbol{S}}$, then it is possible to move a cost from $c_{\boldsymbol{S}}$ to (i, a). The algorithm looks at each value (j, b) in the explanation. If $c_j(b) \geq \theta$ then (j, b) can be a source. Otherwise, if

Algorithm 1 VAC-lin iteration - Phase 1: Filtering

// Propagate a linear constraint in $\text{Bool}_\theta(\text{P})$. Return the set of removed values or the minimum cost greater than θ, and their explanation.

Function(LinFilter($c_{\boldsymbol{S}}$))

$R_D \leftarrow$ Domain-Consistency($c_{\boldsymbol{S}}$) ;

$OPT \leftarrow$ Optimal-Relaxed-Solution($c_{\boldsymbol{S}}$) ;

if $OPT \geq \theta$ *or* $c_{\boldsymbol{S}}$ *is not satisfiable* **then**

    $Expl \leftarrow$ Find-Explanation() ;

    // If $c_{\boldsymbol{S}}$ is not satisfiable then we consider that $OPT = \top$

    return $(OPT, \emptyset, Expl)$;

$Expl \leftarrow \{(i,a) \mid (i,a)$ has been removed in $\text{Bool}_\theta(\text{P})\ \}$;

$Removed \leftarrow \{(i,a) \mid rc_{\boldsymbol{S}}(i,a) + OPT \geq \theta\} \bigcup R_D$; // Reduced cost filtering

return *(0,Removed,Expl)* ;

// Propagate all the constraints and record the reason for each value removal. Stop when a conflict occurs or when no more values can be removed.

Function(VAC-Filter())

$R \leftarrow \{c_{\boldsymbol{S}} \mid c_{\boldsymbol{S}} \in \boldsymbol{C}\}$;

while $R \neq \varnothing$ **do**

    $c_{\boldsymbol{S}} \leftarrow R.Pop()$;

    $(OPT, Removed, Expl) \leftarrow$ Filter($c_{\boldsymbol{S}}$) ;

    if $OPT \geq \theta$ **then**

        return $(OPT, c_{\boldsymbol{S}}, Expl)$;

    foreach $(i,a) \in Removed$ **do**

        delete a from $\boldsymbol{D}_i^{curr}$;

        $Q.Push(i,a)$;

        killer$(i,a) \leftarrow (c_{\boldsymbol{S}}, Expl)$;

        if $\boldsymbol{D}_i^{curr} = \varnothing$ **then return** $(\top, \emptyset, \bigcup_{a \in \boldsymbol{D}_i}\{(i,a)\})$;

        else $R \leftarrow R \cup \{c_{\boldsymbol{S}'} \mid c_{\boldsymbol{S}'} \in \boldsymbol{C}, c_{\boldsymbol{S}'} \neq c_{\boldsymbol{S}}, i \in \boldsymbol{S}'\}$;

return $(0, \emptyset, \emptyset)$;

$c_j(b) = 0$ then (j, b) can't directly provide costs to $c_{\boldsymbol{S}}$, this value is marked $M(j, b) = true$ and will need to be traced back. This is done in the function `Update-Counters()`. The values are visited by following the queue Q and starting from the last inserted value. Thus, the deleted values are explored in anti-causal order: a deleted value is always explored before any of the removals that caused its deletion.

The algorithm also computes λ the maximal cost movable to $c_\varnothing$ without introducing negative costs. The value of λ depends on the quantity of costs available at each source and the number of operations involving the marked values. Indeed, each value can be the cause of multiple removals and must send costs to multiple cost functions. To keep track of this, Cooper et .al [7] maintain three counters: $k(i, a)$ is the number of quanta requested by a value (i, a), $k_{c_{\boldsymbol{S}}}(i, a)$ is the number of quanta that (i, a) must extend to $c_{\boldsymbol{S}}$ and $k(c_{\boldsymbol{S}}, \tau)$ is the number of quanta requested by tuple $\tau \in \ell(S)$. We have $k(i, a) = \sum_{c_{\boldsymbol{S}} \in \boldsymbol{C}, i \in \boldsymbol{S}} k_{c_{\boldsymbol{S}}}(i, a)$. Initially, the values in the explanation returned by the filtering process request 1 quantum (line 20). We choose λ to be the maximal cost satisfying all the requests. For example, if a cost of 4 is available on value (i, a) and $k(i, a) = 2$, then $\lambda \leq \frac{4}{2} = 2$. Thus, initially, λ can't be greater than the optimal relaxed solution returned by the filtering phase (line 17), nor the unary costs of the values in the explanation (line 5). However, the number of elements $k(c_{\boldsymbol{S}}, \tau)$ grows exponentially with the size of the arity of the constraints. It would be very costly to maintain in large arity linear constraints. Instead, we introduce a counter $k_{c_{\boldsymbol{S}}}$ giving an upper bound on the maximal number of quanta requested by one tuple: $k_{c_{\boldsymbol{S}}} \geq k(c_{\boldsymbol{S}}, \tau) \quad \forall \tau \in \ell(\boldsymbol{S})$. This counter starts at 0 for every constraint and is updated only for linear constraints in function $\texttt{LinExplanation}_{ia}$. This counter is also set to 1 when a linear constraint is responsible for the conflict. We present here how those structures are updated when a linear constraint is involved in a value removal.

If a value (i, a) has been removed due to a linear constraint $c_{\boldsymbol{S}}$, we want to compute the minimal cost tuple with value a assigned to variable i in the linear constraint along an explanation. This is done in function $\texttt{LinExplanation}_{ia}$ by solving $\mathcal{LP}^{ia}_{\boldsymbol{S}}$. If (i, a) has been hard removed then all the tuples have a cost $\top$ and none of them may limit the value of λ. An explanation is computed using conflict explanation [14]. Otherwise, let OPT_{ia} be the optimal cost of $\mathcal{LP}^{ia}_{\boldsymbol{S}}$. If (i, a) has been soft removed, then we increase $k_{c_{\boldsymbol{S}}}$ by $k(i, a)$ (line 13) and update λ according to OPT_{ia} and $k_{c_{\boldsymbol{S}}}$: $\lambda \leq \frac{OPT_{ia}}{k_{c_{\boldsymbol{S}}}}$ (line 28). We add in the explanation the values (j, b) for which $rc_{\boldsymbol{S}}(j, b) < 0$ as described in 3.1. Finally, in both cases, the k structure of the values within the explanation are updated. For each value (j, b) in the explanation then $k(j, b) = k(j, b) + k(i, a)$. Those values are also marked (structure M) if their unary cost is null (line 4), otherwise we update λ (line 5) if necessary.

Finally, all EPTs are performed according to killer and R structures where the queue R contains all the marked values, and their minimal explanations have been saved in killer. After this sequence of EPTs, we know a cost of λ can be moved to $c_\varnothing$. Example 8 illustrates how to enforce VAC-lin.

The space complexity of VAC-lin is dominated both by the killer structure, for each variable we associate to each value an explanation with maximal size $d(r-1)$, where d is the maximum domain size and r is the largest linear constraint, and by the delta costs δ_{ia} associated to every cost function (total number of e functions). Hence the space complexity is $O(nrd^2 + erd)$. As for time complexity, the filtering process requires solving a relaxed knapsack problem in $O(rd \log(rd))$-time for each constraint, which can be propagated up to rd times. Thus, the total complexity is $O(er^2d^2 \log(rd))$. Concerning the tracing phase, there are at most nd values in the queue Q, it needs at most $O(rd \log(rd))$ to find an explanation of a removal caused by a linear constraint. Thus, the total time complexity of this phase is

Algorithm 2 VAC-lin iteration - Phase 2: Computing λ

Function(Update-Counters($(i,a), c_{\mathbf{S}}, KillerSet$))
foreach $(j,b) \in KillerSet$ **do**
    $k(j,b) \leftarrow k(j,b) + k(i,a)$;
    if $c_j(b) = 0$ **then** $M(j,b) \leftarrow$ true ;
    else $\lambda \leftarrow \min(\lambda, \frac{c_j(b)}{k(j,b)})$;

// Computes an explanation for the removal of value (i,a) along with an approximation of the minimal cost tuple with value a assigned to variable i.

Function(LinExplanation$_{ia}$($c_{\mathbf{S}}$,$OldExpl$,(i,a)))
foreach $(j,b) \in OldExpl$ **do** Fix $c_j(b) = \top$;
Fix $\forall b \neq a, \delta_{ib} = \top$;
$OPT_{ia} \leftarrow$ Optimal-Relaxed-Solution($c_{\mathbf{S}}$) ;
$NewExpl \leftarrow$ Find-Explanation() ;
if $c_{\mathbf{S}}$ *is not satisfiable* **then**
    return $(\max(k(i,a), k_{c_{\mathbf{S}}}) \times \lambda, NewExpl)$;
else $k_{c_{\mathbf{S}}} = k_{c_{\mathbf{S}}} + k(i,a)$;
return $(OPT_{ia}, NewExpl)$;

// Trace the conflict found in VAC-Filter back to values with cost $> \theta$. Compute λ the maximal cost movable to $c_{\varnothing}$.

Function(VAC-Tracer())
$(OPT, c_{conflict}, Expl) \leftarrow$ VAC-Filter() ;
$\lambda \leftarrow OPT$;
if $c_{conflict} \neq \emptyset$ **then** $k_{c_{conflict}} = 1$;
foreach $(i,a) \in Expl$ **do**
    $k(i,a) \leftarrow 1, M(i,a) \leftarrow$ true ;
    if $c_i(a) > 0$ **then** $M(i,a) \leftarrow$ false, $\lambda \leftarrow \min(\lambda, c_i(a))$;
while $(Q \neq \varnothing)$ **do**
    $(i,a) \leftarrow Q.Pop()$;
    if $M(i,a)$ **then**
        $R.Push(i,a)$;
        $c_{\mathbf{S}} \leftarrow$ killer$(i,a).first$;
        $(OPT_{ia}, Expl) \leftarrow$ Find-Explanation$_{ia}$&OPT($c_{\mathbf{S}}$, killer$(i,a).second, (i,a)$) ;
        $OPT_{ia} \leftarrow \frac{OPT_{ia}}{\max(k(i,a), k_{c_{\mathbf{S}}})}$;
        $\lambda \leftarrow \min(\lambda, OPT_{ia})$;
        Update-Counters($(i,a), c_{\mathbf{S}}, Expl$) ;

$O(nrd^2 \log(rd))$. Finally, the number of required EPTs to increase $c_\varnothing$ is at most $O(erd)$.

▶ **Example 8.** Let P be a WCSP with 6 Boolean variables with domain $\{a, b\}$, and constraints $c_{12345} : 7x_{1a} + 7x_{2a} + 3x_{3a} + 3x_{4a} + 3x_{5a} \geq 10$, $c_{14} : x_{1a} + x_{4b} \geq 1$, $c_{246} : 2x_{6a} + x_{2b} + x_{4a} \geq 1$ and $c_1(a) = 2$, $c_3(a) = 2$, $c_6(a) = 2, c_\varnothing = 0$. Propagating the constraints as did in [21] does not increase $c_\varnothing$. The optimal relaxed solution of this problem is 0,824 ($\{x_{1a} = 0,41176, x_{2a} = 0,41176, x_{3a} = 0, x_{4a} = 0,41176, x_{5a} = 1, x_{6a} = 0\}$), we show that enforcing VAC-lin increases $c_\varnothing$ by 1.
If we apply VAC-lin with a threshold $\theta = 1$. In $\text{Bool}_\theta(\text{P})$, $(1, a)$, $(3, a)$ and $(6, a)$ are directly removed, it follows by domain propagation on c_{12345} that $(2, b)$ can be removed and we set $\mathsf{killer}(2, b) = (c_{12345}, \{(1, a), (3, a)\})$. Similarly, $(4, b)$ is removed by domain propagation on c_{246} and we set $\mathsf{killer}(4, b) = (c_{246}, \{(2, b), (6, a)\})$. Finally c_{14} is infeasible with explanation $\{(1, a), (4, b)\}$, thus $\text{Bool}_\theta(\text{P})$ is not GAC.
We set $\lambda = \top$ and start tracing back the GAC operations. c_{14} is infeasible because $(1, a)$ and $(4, b)$ have been removed. The k structures are updated: $k(1, a) = k(4, b) = k_{c_{14}} = 1$. We directly have $c_1(a) = 2$, we can use this cost as a source and update λ: $\lambda = \frac{c_1(a)}{k(1,a)} = 2$. Value $(4, b)$ verifies $c_4(b) = 0$, hence, the value is marked: $M(4, b) = True$ and need to be traced. Value $(4, b)$ has been hard removed because it has no support on c_{246}, the solver computes the minimal explanation $\{(2, b), (6, a)\}$ using conflict explanation [14]. The k structures are updated: $k(2, b) = k(6, a) = k_{c_{246}} = k(4, b) = 1$. We directly have $c_6(a) = 2$, we can use this cost as a source, λ does not need to be modified. Value $(2, b)$ verifies $c_2(b) = 0$, hence, the value is marked: $M(2, b) = True$ and need to be traced. Value $(2, b)$ has been hard removed because it has no support on c_{12345}, the solver computes the minimal explanation $\{(1, a)\}$ using conflict explanation [14]. We update the k structures: $k(1, a) = k(1, a) + k(2, b) = 2$, $k_{c_{12345}} = 1$. We need to update λ: $\lambda = \frac{c_1(a)}{k(1,a)} = 1$. The conflict has been explained.
We deduce the following EPTs from R, killer, and λ:

1) $extend(c_1, a, c_{12345}, 1)$	5) $project(c_4, c_{246}, b, 1)$
2) $project(c_2, c_{12345}, b, 1)$	6) $extend(c_4, b, c_{14}, 1)$
3) $extend(c_2, b, c_{246}, 1)$	7) $extend(c_1, a, c_{14}, 1)$
4) $extend(c_6, a, c_{246}, 1)$	8) $project(c_\varnothing, c_{14}, \emptyset, 1)$

4 Experimental Results

We implemented VAC-lin in toulbar2, an open-source C++ WCSP solver.[2] The original VAC algorithm was already implemented in the solver (only for binary cost functions in extension). Both VAC and VAC-lin are applied in preprocessing only. A weaker SAC algorithm (EDAC [9]) is applied at every search node of a hybrid best/depth-first branch-and-bound search method [1]. We also considered solving without VAC, which corresponds to the default setting in toulbar2 (called no-VAC in the results). We compared the different variants of toulbar2 (no-VAC, VAC, VAC-lin) with choco, an open-source Java CP solver and IBM cplex, a state-of-the-art integer programming solver.[3] Choco and toulba2 used the same *dom/wdeg* variable ordering heuristic [4] with last conflict [19]. The value ordering heuristic is the minimum domain value for choco and EAC/VAC/VAC-lin *support value* for toulbar2 [7, 33]. In VAC and VAC-lin, it corresponds to choosing first the minimum domain value in Bool(P) after doing the filtering phase. Both solvers use solution phase saving [12].

[2] `https://github.com/toulbar2/toulbar2` version 1.2.1.
[3] `https://github.com/chocoteam/choco-solver` version 4.10.14 and cplex version 22.1.1.0 in single-thread mode and with non-premature stop parameters *EPAGAP=EPGAP=EPINT=0*.

bench	total	no-VAC	VAC	VAC-lin	LP
MIPLIB 2017	184	40.14% (150)	40.07% (152)	48.05% (143)	74.33% (179)
CPD	30	99.9635% (30)	99.979% (30)	99.9803% (30)	99.9853% (25)
PB'2007	77	74.43% (77)	74.90% (77)	86.67% (77)	89.14% (77)
XCSP'2022	158	25.54% (136)	27.26% (136)	27.70% (136)	38.71% (100)
XCSP'2023	155	5.92% (134)	5.92% (134)	6.85% (119)	15.57% (92)

Table 1 Quality of lower bounds per benchmark averaged over the number of instances (in parentheses) where a particular method produced a lower bound in memory and CPU-time limits.

To test our approach with a large number of linear constraints, we chose integer linear problems from the MIPLIP 2017 benchmark. On the opposite, we tested on the Computational Protein Design (CPD) benchmark having few additional linear constraints, large domains, and several binary cost functions in extension. We also tested on a selection of the Pseudo-Boolean 2007 Evaluation benchmark (PB07). Last, to show the expressive power of CFNs with linear constraints, we experimented with the XCSP 2022/2023 benchmarks.

Experiments on MIPLIB were made on a single thread of a cluster of AMD EPYC 7713 at 2.0/3.7 GHz (turbo) with 8GB and $3,600$-second CPU-time limits. Experiments on CPD / XCSP / PB07 were made on a single core of an Intel Xeon E5-2680 v3 at 2.5GHz with 64GB and $3,600$s / $2,400$ / $1,800$ limits respectively.[4]

4.1 MIPLIB 2017 01LP

We selected 200 instances from the MIPLIB 2017 collection, containing only Boolean 0/1 variables. Among them, 184 instances have a known feasible solution.[5] We preprocessed them using cplex and applied our different methods to these preprocessed instances. In Table 1, we report the average quality of lower bounds for our three variants, no-VAC, VAC, and VAC-lin, and also for the continuous linear relaxation (LP).[6] As expected, the linear relaxation gives the strongest bounds. It is also the most robust with only 5 instances where the dual simplex did not finish in 1 hour. Default toulbar2 (no-VAC) failed to produce an initial lower bound on almost 20% of the instances, showing a lot of engineering work remains to be done to reach the same efficiency level as a commercial state-of-the-art LP solver. Although the original VAC algorithm cannot prove profitable on this benchmark due to the limited number of arity-2 linear constraints, our VAC-lin significantly improves the initial bound, going from 40% to 48% on average. But for this benchmark, it was not sufficient to solve any more instances (15 solved instances in total whereas cplex solved 100). It did not solve one instance solved by VAC or no-VAC in 5.4s (4.3s resp.). This instance has very large costs (*neos-633273*). Here VAC-lin made 181,082 iterations before time-out.[7] cplex solved it in 0.17s. choco performed poorly, as toulbar2, solving 14 instances (Table 2).

---

[4] For CPD, we ran choco on an Intel Xeon E5-2687W v4 at 3GHz with 256GB and $3,600$s. We also add the option *-d:* in toulbar2 to remove its default dichotomic branching rule.

[5] `https://miplib.zib.de/tag_collection.html`

[6] The quality of a nonzero bound l for a given instance with best-known nonzero solution value b is defined by $max(0, min(l/b, b/l))$. The quality is zero otherwise. We report average quality over the number of successful instances producing a bound at the root search node (a different number for each method).

[7] This problem is already known for VAC [7]. Premature termination is a possible workaround.

bench	total	choco	cplex	toulbar2-no-VAC	toulbar2-VAC	toulbar2-VAC-lin
MIPLIB 2017	184	14	**100**	16	16	15
CPD	30	0	19	29	**30**	**30**
PB'2007	77	16	**67**	57	57	66
XCSP'2022	158	41	**63**	57	58	59
XCSP'2023	155	20	**39**	15	14	21

Table 2 Number of solved instances per benchmark.

4.2 Computational Protein Design with Diversity Constraints

As in [21], we selected 30 CPD instances having from 23 to 97 variables and 48 to 194 values in their largest domain. For each instance, ten diverse solutions with a Hamming distance equal to ten were generated using toulbar2 with a dual encoding [25]. Next, we transformed the resulting solutions into ten linear diversity constraints and added them to our original instance. Choco could not solve any CPD instance in 1 hour. It found solutions for half of the instances with an average distance to optimality of 0.2155%. VAC and VAC-lin produced almost the same results, solving all the instances to optimality. VAC-lin improved the initial lower bound found by VAC in one-third of the instances. The absolute initial gap was reduced by 9.87%, going from VAC to VAC-lin. However, it did not reduce the number of search nodes, nor its solving time significantly. The same behavior between VAC and VAC-lin was observed with an additional upper-bound preprocessing called RASPS [33]. no-VAC could not solve one instance to optimality (1BRS) and cplex solved half of the instances (Fig.3.L).

4.3 Pseudo Boolean 2007 OPT-SMALLINT-LIN-Other Competition

We ran experiments on 77 instances introduced at PB 2007 Evaluation. They correspond to unweighted Max-SAT instances with 66.3% of arity-2 clauses, 25% of arity-3, and the rest from arity 4 up to $3,140$.[8] Here cplex obtained the best results, solving 67 instances within the CPU-time limit of $1,800$s. It dominates VAC-lin, which solved 66 instances and was much slower than cplex (see Table 2 and Fig.3.Right).[9] The original VAC algorithm did not improve the baseline (no-VAC and VAC having almost the same results, we draw only no-VAC in Fig.3.R).

For this benchmark, VAC-lin clearly dominates VAC, which solved 57 instances.[10] The largest instance solved by VAC-lin (in 387s, compared to 38s for cplex) has $203,287$ variables and $469,077$ clauses. Choco did not perform well, solving only 16 instances. However, it found better solutions on the unsolved *aksoy/decomp* instances than the other solvers.[11]

4.4 XCSP 2022 and 2023 MiniCOP Competition

We restricted to the mini COP category of the 2022 and 2023 XCSP competitions. [12]

[8] `http://www.cril.univ-artois.fr/PB07/benchs/PB07-OTHER.tar`. 10 *aksoy/decomp* instances contain also capacity constraints and were not solved by any solver in our experiments.

[9] It did not solve instance *aksoy/normalized-fir08_area_delay*. cplex solved it in 18.5 seconds. The best solver in the Max-SAT Evaluation 2023 took 34.56s to solve this instance (WMaxCDCL-S6-HS12).

[10] E.g., VAC and no-VAC did not solve *manquinho/normalized-f20c10b_017_area_delay* whereas VAC-lin solved it in 43s and cplex in 2.3s.

[11] Average objective value of 27.6 by choco, 30.8 by VAC and no-VAC. cplex and VAC-lin did not find a solution for *normalized-matrix_5x3_4* instance.

[12] `https://xcsp.org/competitions`

Although the lower bound quality of VAC-lin is slightly better than VAC, it is much higher in some particular families (XCSP22/CoinsGrid, XCSP23/Auctions) where the solving time was greatly reduced compared to the original VAC algorithm. Thus, VAC-lin solved a few more instances than VAC or no-VAC. It also performed better than or similar to choco depending on the benchmark.[13]

It is not a surprise to see the nice results obtained by cplex. It was already observed in past MiniZinc Challenges.

5 Conclusion

Although VAC-lin improved the initial lower bound compared to the original VAC, in most cases it wax not sufficient to obtain significant speed-up (except on some particular categories of PB07 and XCSP). For some difficult instances, applying a stronger soft arc consistency algorithm during search can pay off [20]. It remains to test VAC-lin in such situations. In the future, we would like to apply the same methodology we made for linear constraints to other global constraints such as AllDifferent.

References

1. D Allouche, S de Givry, G Katsirelos, T Schiex, and M Zytnicki. Anytime Hybrid Best-First Search with Tree Decomposition for Weighted CSP. In *Proc. of CP-15*, pages 12–28, Cork, Ireland, 2015.
2. David Allouche, Christian Bessiere, Patrice Boizumault, Simon De Givry, Patricia Gutierrez, Jimmy HM Lee, Ka Lun Leung, Samir Loudni, Jean-Philippe Métivier, Thomas Schiex, et al. Tractability-preserving transformations of global cost functions. *Artificial Intelligence*, 238:166–189, 2016.
3. Endre Boros and Peter L Hammer. Pseudo-boolean optimization. *Discrete applied mathematics*, 123(1-3):155–225, 2002.
4. F Boussemart, F Hemery, C Lecoutre, and L Sais. Boosting systematic search by weighting constraints. In *ECAI*, volume 16, page 146, 2004.
5. Guillaume Claus, Hadrien Cambazard, and Vincent Jost. Analysis of reduced costs filtering for alldifferent and minimum weight alldifferent global constraints. In *ECAI 2020*, pages 323–330. IOS Press, 2020.
6. Martin Cooper, Simon de Givry, and Thomas Schiex. Graphical models: queries, complexity, algorithms. *Leibniz International Proceedings in Informatics*, 154:4–1, 2020.
7. Martin C Cooper, Simon de Givry, Martı Sánchez, Thomas Schiex, Matthias Zytnicki, and Tomas Werner. Soft arc consistency revisited. *Artificial Intelligence*, 174(7-8):449–478, 2010.
8. Martin C. Cooper, Simon de Givry, and Thomas Schiex. *Valued Constraint Satisfaction Problems*, pages 185–207. Springer International Publishing, 2020.
9. Simon de Givry, Federico Heras, Matthias Zytnicki, and Javier Larrosa. Existential arc consistency: Getting closer to full arc consistency in weighted CSPs. In *Proc. of IJCAI-05*, pages 84–89, Edinburgh, Scotland, 2005.
10. Simon de Givry and George Katsirelos. Clique cuts in weighted constraint satisfaction. In *Proc. of CP-17*, pages 97–113, Melbourne, Australia, 2017.
11. Rina Dechter and Irina Rish. Mini-buckets: A general scheme for bounded inference. *Journal of the ACM (JACM)*, 50(2):107–153, 2003.

[13] Compared to XCSP'2023 official results, choco could not solve BeerJugs-table-07, BeerJugs-table-09, BeerJugs-table-10, Sonet-s2ring02, TravelingSalesman-015-30-00, but solved HCPizza-20-20-2-8-02 and TSPTW-n040w020-1. The different parameter settings can explain this discrepancy. We used *dom/wdeg* instead of *dom/wdeg_cacd* and add solution phase saving.

12 E Demirovic, G Chu, and P J. Stuckey. Solution-based phase saving for CP: A value-selection heuristic to simulate local search behavior in complete solvers. In *Proc. of CP-18*, pages 99–108, Lille, France, 2018.

13 Filippo Focacci, Andrea Lodi, and Michela Milano. Cost-based domain filtering. In *Principles and Practice of Constraint Programming–CP'99: 5th International Conference, CP'99, Alexandria, VA, USA, October 11-14, 1999. Proceedings 5*, pages 189–203. Springer, 1999.

14 Emmanuel Hebrard and Mohamed Siala. Explanation-based weighted degree. In *International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 167–175. Springer, 2017.

15 Barry Hurley, Barry O'Sullivan, David Allouche, George Katsirelos, Thomas Schiex, Matthias Zytnicki, and Simon de Givry. Multi-language evaluation of exact solvers in graphical model discrete optimization. *Constraints*, 21(3):413–434, 2016.

16 V Kolmogorov. Convergent tree-reweighted message passing for energy minimization. *IEEE transactions on pattern analysis and machine intelligence*, 28(10):1568–1583, 2006.

17 Nikos Komodakis, Nikos Paragios, and Georgios Tziritas. MRF energy minimization and beyond via dual decomposition. *IEEE transactions on pattern analysis and machine intelligence*, 33(3):531–552, 2010.

18 J. Larrosa. On arc and node consistency in weighted CSP. In *Proc. AAAI'02*, pages 48–53, Edmondton, (CA), 2002.

19 C. Lecoutre, L Saïs, S. Tabary, and V. Vidal. Reasoning from last conflict(s) in constraint programming. *ai*, 173:1592,1614, 2009.

20 Pierre Montalbano, David Allouche, Simon De Givry, George Katsirelos, and Tomáš Werner. Virtual pairwise consistency in cost function networks. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 417–426. Springer, 2023.

21 Pierre Montalbano, Simon de Givry, and George Katsirelos. Multiple-choice knapsack constraint in graphical models. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 282–299. Springer, 2022.

22 David Pisinger and Paolo Toth. Knapsack problems. In *Handbook of combinatorial optimization*, pages 299–428. Springer, 1998.

23 Daniel Prusa and Tomas Werner. Universality of the local marginal polytope. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1738–1743, 2013.

24 Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

25 Manon Ruffini, Jelena Vucinic, Simon de Givry, George Katsirelos, Sophie Barbe, and Thomas Schiex. Guaranteed diversity and optimality in cost function network based computational protein design methods. *Algorithms*, 4(6:168), 2021.

26 T. Schiex. Arc consistency for soft constraints. In *Proc. of CP-00*, pages 411–424, Singapore, 2000.

27 Meinolf Sellmann. Approximated consistency for knapsack constraints. In *International Conference on Principles and Practice of Constraint Programming*, pages 679–693. Springer, 2003.

28 Meinolf Sellmann. The practice of approximated consistency for knapsack constraints. In *AAAI*, pages 179–184, 2004.

29 D Sontag, D Choe, and Y Li. Efficiently searching for frustrated cycles in MAP inference. In *Proc. of UAI*, pages 795–804, Catalina Island, CA, USA, 2012.

30 D Sontag, T Meltzer, A Globerson, Y Weiss, and T Jaakkola. Tightening LP relaxations for MAP using message-passing. In *Proc. of UAI*, pages 503–510, Helsinki, Finland, 2008.

31 Siddharth Tourani, Alexander Shekhovtsov, Carsten Rother, and Bogdan Savchynskyy. Taxonomy of dual block-coordinate ascent methods for discrete energy minimization. In *Proc. of AISTATS-20*, pages 2775–2785, Palermo, Sicily, Italy, 2020.

32 Michael A Trick. A dynamic programming approach for consistency and propagation for knapsack constraints. *Annals of Operations Research*, 118:73–84, 2003.

33 Fulya Trösser, Simon de Givry, and George Katsirelos. Relaxation-aware heuristics for exact optimization in graphical models. In *CPAIOR20P*, pages 475–491, CPAIOR20L, 2020.

34 P.M. Vaidya. Speeding-up linear programming using fast matrix multiplication. In *30th Annual Symposium on Foundations of Computer Science*, pages 332–337, 1989.

35 Tomas Werner. A Linear Programming Approach to Max-sum Problem: A Review. *IEEE Trans. on Pattern Recognition and Machine Intelligence*, 29(7):1165–1179, July 2007.

36 M. Zytnicki, C. Gaspin, S. de Givry, and T. Schiex. Bounds Arc Consistency for Weighted CSPs. *Journal of Artificial Intelligence Research*, 35:593–621, 2009.

bench	choco	cplex	no-VAC	VAC	VAC-lin
MIPLIB 2017	244,882 (82)	1,968 (153)	1,373 (74)	1,238 (75)	1,080 (70)
CPD	788 (30)	298 (19)	675 (30)	604 (30)	589 (30)
PB'2007	308 (77)	315 (76)	283 (77)	280 (77)	243 (76)
XCSP'2022	10,499 (157)	812 (110)	1,971 (123)	1,894 (123)	1,760 (123)
XCSP'2023	12,810 (144)	491 (93)	2,867 (119)	2,664 (119)	2,190 (107)

Table 3 Total number of solutions found by each search method per benchmark (in parentheses, number of instances where at least one solution has been found). E.g., on *CoinsGrid-31-14*, choco found 961 intermediate solutions before reaching the time limit, whereas no-VAC and VAC found 24 intermediate solutions, VAC-lin 3 (optimality proof in 6.37s), and cplex only 1 (optimality in 0.01s).

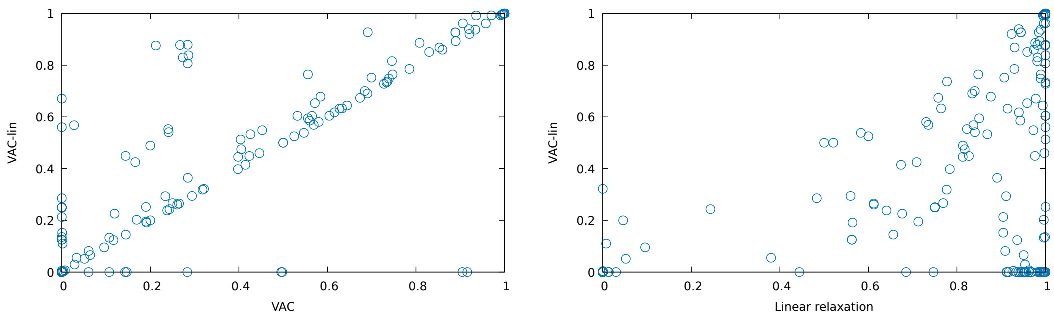

Figure 2 Quality of lower bounds on MIPLIB 2017.

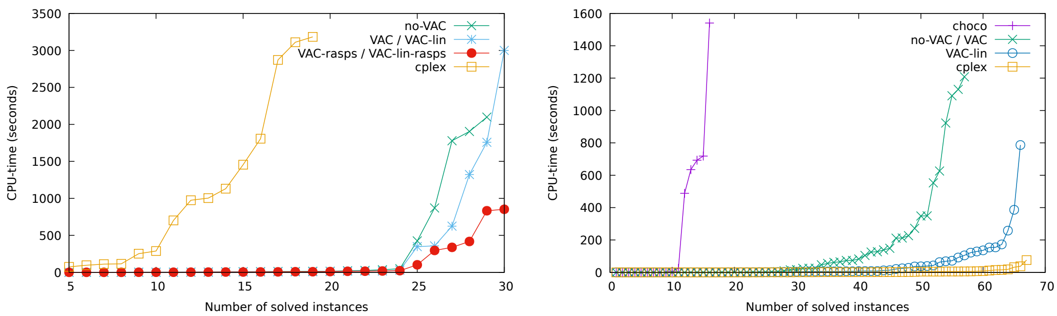

Figure 3 Cactus plot of CPU-time to solve CPD with diversity (Left Fig.) and PB07 (Right).

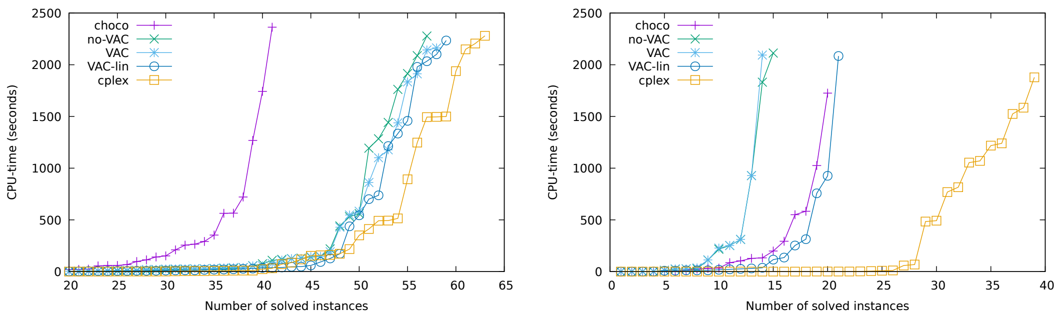

Figure 4 Cactus plot of CPU-time to solve XCSP'2022 (Left Fig.) and XCSP'2023 (Right).