Virtual Arc Consistency for Linear Constraints in Cost Function Networks

Pierre Montalbano
LIFAT, UR 6300
Université de Tours, ANITI, INRAE
Tours, France
0000-0001-8126-892X

Simon de Givry
MIAT, UR 875
Université de Toulouse, ANITI, INRAE
Toulouse, France
0000-0002-2242-0458

George Katsirelos
MIA Paris, AgroParisTech
ANITI, INRAE
Paris, France
0000-0002-3727-6698

Abstract—In Constraint Programming, solving discrete minimization problems with hard and soft constraints can be done either using (i) soft global constraints, (ii) a reformulation into a linear program, or (iii) a reformulation into local cost functions. Approach (i) benefits from a vast catalog of constraints. Each soft constraint propagator communicates with other soft constraints only through the variable domains, resulting in weak lower bounds. Conversely, the approach (ii) provides a global view with strong bounds, but the size of the reformulation can be problematic. We focus on approach (iii) in which soft arc consistency (SAC) algorithms produce bounds of intermediate quality. Recently, the introduction of linear constraints as local cost functions increases their modeling expressiveness. We adapt an existing SAC algorithm to handle linear constraints. We show that our algorithm significantly improves the lower bounds compared to the original algorithm on several benchmarks, reducing solving time in some cases.

Index Terms—constraint programming, linear program, duality, soft arc consistency

I. INTRODUCTION

Graphical models provide a powerful framework for modeling a variety of combinatorial problems, addressing tasks that range from satisfaction problems to probabilistic models. [1]. They employ local functions defined over 'small' subset of variables to represent diverse interactions among them. For example, to model the Constraint Satisfaction Problem (CSP) [2], each local function is a constraint evaluating to true (satisfied) or false (falsified). Here, we focus on Cost Function Networks (CFNs), where each local function is a cost function; the task of finding an assignment that minimizes the sum of all cost functions is known as the Weighted Constraint Satisfaction Problem (WCSP). Most methods for finding optimal solutions rely on a branch-and-bound procedure, using either very fast but static and memory-intensive bounds [3] or memory-light ones [4] to compute lower bounds. Here, we focus on the latter, known as *Soft Arc Consistency* (SAC) algorithms. Similar to CSP propagation, they reason about each non-unary cost function individually. Different levels of SAC exist, each offering a trade-off between propagation strength (lower-bound quality) and propagation time. Striking the right balance between the quality of derived lower bounds and the computational time required to construct them is crucial for achieving efficiency. *Virtual Arc Consistency* (VAC) [4] can derive strong lower bounds but is expensive to enforce. The principle of VAC is

to study a CSP, denoted $\text{Bool}(P)$, derived from a WCSP P. For every cost function, only tuples and values with zero cost are allowed in $\text{Bool}(P)$. If $\text{Bool}(P)$ is inconsistent, then the lower bound of P is non-zero. If the inconsistency of $\text{Bool}(P)$ is detected by Generalized Arc Consistency (GAC), then VAC has been designed to extract a lower bound.

CFN also benefits from the flexibility of 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 pruning, propagators for new constraints must also compute a lower bound. This has been done for various global constraints, including AllDifferent, clique, and linear inequality constraints [5]–[7].

Contributions. Motivated by VAC's good performance and the recent introduction of linear constraints in CFN [7], we study how to combine these contributions. Previous approaches to handling linear constraints in CFNs tend to absorb unary costs, which subsequent propagation can no longer exploit. Enforcing VAC enables the identification of sequences of cost moves involving different propagations and facilitates communication between linear constraints. However, enforcing VAC on a linear constraint requires keeping in $\text{Bool}(P)$ only those values that can be part of a zero-cost tuple. For linear inequality constraints, this corresponds to solving a problem similar to the Knapsack problem and is therefore NP-hard. We demonstrate how reduced cost filtering [8] can be used to detect subsets of inconsistent values. This leads to VAC-lin, which enforces an incomplete GAC on $\text{Bool}(P)$. This approach is implemented in the toulbar2 WCSP solver and tested on several benchmarks.

II. BACKGROUND

A. Weighted Constraint Satisfaction Problem

Definition 1. *A Cost Function Network (CFN) P is a tuple $(\boldsymbol{X}, \boldsymbol{D}, \boldsymbol{C}, \top)$ where $\boldsymbol{X} = \{1, \ldots n\}$ is a set of n variables, $\boldsymbol{D}$ a list of finite domains $\boldsymbol{D}_i \in \boldsymbol{D}$ for $i \in \boldsymbol{X}$. $\boldsymbol{C}$ is a set of cost functions. Each cost function $c_{\boldsymbol{S}} \in \boldsymbol{C}$ 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 assignment of value $v \in \boldsymbol{D}_i$ to variable $i \in \boldsymbol{X}$. The *arity* of a cost function is the size of its scope. Unary (resp. binary) cost functions have arity 1 (resp. 2). In this paper, we assume each variable has exactly one associated unary cost function. For $\boldsymbol{S} \subseteq \boldsymbol{X}$, let $\ell(\boldsymbol{S}) = \Pi_{i \in \boldsymbol{S}} \boldsymbol{D}_i$ denote the Cartesian product of variable domains in $\boldsymbol{S}$. An assignment (or tuple) $\tau \in \ell(\boldsymbol{S})$ is an assignment of a value in the domain $\boldsymbol{D}_i$ to every variable $i \in \boldsymbol{S}$. We denote by $\tau[i]$ the value assigned to $i \in \boldsymbol{S}$. If $\boldsymbol{S} = \boldsymbol{X}$, then τ defines a *complete assignment*; otherwise, it is a *partial assignment*.

A cost function $c_{\boldsymbol{S}}$ maps tuples $\tau \in \ell(\boldsymbol{S})$ to costs $c_{\boldsymbol{S}}(\tau)$. The cost of a complete assignment $\tau \in \ell(\boldsymbol{X})$ is given by $c_P(\tau) = \sum_{c_{\boldsymbol{S}} \in \boldsymbol{C}} c_{\boldsymbol{S}}(\tau[\boldsymbol{S}])$, $\tau[\boldsymbol{S}]$ being the projection of τ on $\boldsymbol{S}$. Without loss of generality, we assume all costs are positive integers, bounded by $\top$, a special constant signifying inconsistency. If $c_{\boldsymbol{S}}(\tau) = \top$, then τ is inconsistent. A cost function $c_{\boldsymbol{S}}$ 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 cost functions is a constraint network (CN). In the following, we use the term *cost function* interchangeably with the term constraint.

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 [9]. When the underlying CFN is a CN, the problem reduces to a CSP. Throughout this paper, *WCSP* refers to both the optimization task and the CFN.

Each cost function is represented either in *extension* or *intention*. A cost function defined in extension, also known as a *table constraint*, explicitly lists all the tuples and their associated costs. This is feasible only for low-arity cost functions due to memory usage that 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 an empty scope cost function $c_{\varnothing}$ representing a constant term in the objective function. Since no negative costs exist, $c_{\varnothing}$ serves as a global lower bound for all assignments. It plays a key role in SAC algorithms.

B. Soft Arc Consistency

Soft Arc Consistency (SAC) algorithms examine small subsets of cost functions sequentially. Beyond removing the locally inconsistent values, they compute a lower bound by increasing $c_{\varnothing}$. To achieve this, they rely on reparametrization: a reparameterization P' of a WCSP P is a WCSP with an identical structure, i.e., the set of scopes and variables is 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 set (or 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 higher-arity cost function $c_{\boldsymbol{S}_2}$ to the lower-arity $c_{\boldsymbol{S}_1}$ and the move is called a *projection*, denoted $project(c_{\boldsymbol{S}_1}, c_{\boldsymbol{S}_2}, \tau_1, \alpha)$ with $\tau_1 \in \ell(\boldsymbol{S}_1)$. When $\alpha < 0$, cost moves to the higher-arity cost function $c_{\boldsymbol{S}_2}$ and the move is called an *extension*, denoted $extend(c_{\boldsymbol{S}_1}, c_{\boldsymbol{S}_2}, \tau_1, -\alpha)$. When $\boldsymbol{S}_1 = \emptyset$ and $|\boldsymbol{S}_2| = 1$, with $\boldsymbol{S}_2 = \{i\}$, the move is called a *unary projection*, denoted $unaryProject(c_i, \alpha)$, equivalent to $MoveCost(c_{\varnothing}, c_i, \emptyset, \alpha)$. Extensions are never performed from $c_{\varnothing}$, so it increases monotonically during the algorithm and along each branch of the search tree.

Identifying the cost moves that produce an optimal reparameterization—one that maximizes the lower bound—is nontrivial. It has been shown that any reparameterization can be derived by a set of local cost moves [10] and that the optimal reparameterization (with α rational) can be found from the optimal dual solution of a linear relaxation of the WCSP [4], whose feasible region is the *local polytope*.

However, solving this LP to optimality is often prohibitively expensive because the worst-case time complexity of an exact LP algorithm is $O((mr + m^2)\sqrt{m})$ [11], where m is the number of linear constraints of the linear relaxation and r their largest arity. This poor asymptotic complexity matches empirical observation [12]. Moreover, the structure of this LP does not allow more efficient algorithms, as solving LPs of this form is as hard as solving arbitrary LPs [13]. Instead, research has focused on producing high-quality—though potentially suboptimal—feasible dual solutions. Various algorithms have been proposed for this, like Block-Coordinate Ascent (BCA) algorithms developed for image analysis [10], [14]–[18] or *soft arc consistencies* in constraint programming [4], [19]–[22]. Notably, the strongest algorithms from both lines of research, such as TRWS [10] and VAC [4] converge on fixpoints with the same properties.

Here, we focus on Soft Arc Consistency (SAC) [4] and define several variants.

Definition 2. *A WCSP P is Node Consistent (NC) [20] if for every variable $i \in \boldsymbol{X}$ there exists a value $a \in \boldsymbol{D}_i$ such that $c_i(a) = 0$ and for every value $b \in \boldsymbol{D}_i$, $c_{\varnothing} + c_i(b) < \top$.*

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

An important SAC algorithm for this paper is *Virtual Arc*

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 $\boldsymbol{S}_2$

Input: scopes $\boldsymbol{S}_1 \subset \boldsymbol{S}_2$
Input: $\tau_1 \in \ell(\boldsymbol{S}_1)$
Input: cost α to move
1 $c_{\boldsymbol{S}_1}(\tau_1) \leftarrow c_{\boldsymbol{S}_1}(\tau_1) + \alpha$;
2 **foreach** $\tau_2 \in \ell(\boldsymbol{S}_2) \mid \tau_2[\boldsymbol{S}_1] = \tau_1$ **do**
3 $\quad \mid \quad c_{\boldsymbol{S}_2}(\tau_2) \leftarrow c_{\boldsymbol{S}_2}(\tau_2) - \alpha$;
4 **end**

Consistency (VAC) [4]. 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 inconsistent, no such assignment exists, and the optimum of P has a cost strictly greater than $c_\varnothing$. It has been shown [4] that an inconsistency certificate produced by arc consistency on Bool(P) can be used to derive a reparameterization of P with increased $c_\varnothing$.

In the following, $AC(P)$ denotes the *arc consistent closure* of a CSP P: the unique CSP obtained by removing arc-inconsistent values from domains. An empty AC closure implies inconsistency.

Definition 3 (Virtual Arc Consistency [4]). *A WCSP P is virtual arc consistent if the (generalized) arc consistency closure of the CSP Bool(P) is non-empty.*

Theorem 1 ([4]). *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 is 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] to compute a sequence of EPTs $\boldsymbol{\sigma}$ such that applying $\boldsymbol{\sigma}$ increases $c_\varnothing$ by a cost λ.
3) Apply $\boldsymbol{\sigma}$ to P and go back to phase 1.

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) that has lost all supports on constraint c_{ij} can be viewed as extending $\top$ from each pruned value (j, b) to c_{ij}, which marks all supporting tuples of (i, a) in c_{ij} as forbidden, then projecting $\top$ from c_{ij} to (i, a). By choosing a sufficiently small λ, we can repeat those EPTs in P using λ instead of $\top$, ensuring that no negative costs are introduced. The purpose of step 2 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_θ. 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_θ is equivalent to VAC. Clearly, VAC_θ may discover only a subset of the reparameterizations that VAC can find. But the higher θ is, the higher the costs of P 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

---

[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 during propagation.

---

has an empty AC closure. Thus, VAC_θ is applied by starting with high values for θ to quickly increase the lower bound and gradually decreasing it until $\theta = 1$. In practice, a static schedule based on the original cost distribution is used [4].

Cost function size strongly impacts the VAC enforcement algorithm; 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. A dedicated algorithm is required to enforce a possibly weaker consistency in the presence of global constraints.

Example 1. *Let P be a WCSP with 4 variables, each with domains $\{a, b\}$, and 5 nonzero cost functions $c_1(a) = 1$, $c_4(b) = 1$, $c_{12}(b, a) = 1$, $c_{23}(b, a) = 1$, $c_{34}(b, a) = 1$, all other tuples and $c_\varnothing, c_2, c_3$ having cost 0. The AC closure of Bool(P) is empty, indeed, values $(1, a)$ and $(4, b)$ are directly removed from Bool(P) because $c_1(a) = c_4(b) > 0$. Consequently, value $(2, a)$ has no support on c_{12} and $(3, b)$ has no support on c_{34}; These values can then be removed Finally, $(2, b)$ has no support on c_{23}, and a domain wipe-out occurs at variable 2. By analyzing the trace that led to this conflict, VAC deduces the following sequence of EPTs, increasing $c_\varnothing$ by one.*

1) $extend(c_1, c_{12}, a, 1)$ *5) $extend(c_3, c_{23}, b, 1)$*
2) $extend(c_4, c_{34}, b, 1)$ *6) $project(c_2, c_{23}, b, 1)$*
3) $project(c_2, c_{12}, a, 1)$ *7) $unaryProject(c_2, 1)$*
4) $project(c_3, c_{34}, b, 1)$

The resulting reparameterization P' of P is $c_{12}(a, b) = 1$, $c_{23}(a, b) = 1$, $c_{34}(a, b) = 1$, and $c_\varnothing = 1$, all other tuples and c_1, c_2, c_3, c_4 having cost 0.

C. Linear Inequality Constraints

Linear constraints are global constraints capturing a linear interaction between variables. They are expressive, compact, and used in various optimization problems, including computer science, operations research, and artificial intelligence [23]. We consider a linear *inequality* constraint c_S of the form: $\sum_{i \in S} \sum_{v \in D_i} w_{iv} x_{iv} \geq C$, where $c_S \in C$, x_{iv} is a 0/1 variable taking value 1 when $v \in D_i$ is assigned to variable $i \in S$. 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, using $\sum_{v \in D_i} x_{iv} = 1$, and equality constraints can be encoded by two inequality constraints. Initially, linear constraints are considered as hard, i.e., $\forall \tau \in \ell(S)$ satisfying the constraint, it holds that $c_S(\tau) = 0$, otherwise $c_S(\tau) = \top$. If EPTs involve a linear constraint, the cost of the allowed tuples can be modified, yielding $0 < c_S(\tau) < \top$.

Recent work introduced a way to represent and propagate linear constraints in a WCSP solver [7] through so-called *delta costs*. A cost δ_{iv} is associated with each assignment $i \in S, v \in D_i$, and it captures the amount of costs moved from the unary cost function c_i to a linear constraint c_S. A cost $\alpha > 0$ moved from c_S to $c_i(v)$ decreases δ_{iv}, and is denoted $project(c_i, c_S, v, \alpha) = LinMoveCost(c_i, c_S, v, \alpha)$. Thus, we can have negative δ costs. When $\alpha < 0$, costs move in the opposite direction and increases δ_{iv}; we denote this by $extend(c_i, c_S, v, -\alpha) = LinMoveCost(c_i, c_S, v, \alpha)$. By $\delta_\emptyset$, we denote the cost moved

from $c_{\boldsymbol{S}}$ to $c_{\varnothing}$ ($LinProject(c_{\boldsymbol{S}}, \alpha)$), which is necessarily positive. Compared to MoveCost, LinMoveCost allows cost moves in constant time and space, rather than exponential in $|\boldsymbol{S}| - 1$.

After any sequence of EPTs, the cost of an assignment τ is defined by:

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

Initially, no cost moves have been performed, and all the δ costs are 0. In [7], the authors define a SAC algorithm that moves the costs between a linear constraint $c_{\boldsymbol{S}}$, unary cost functions $c_i, \forall i \in \boldsymbol{S}$, and $c_{\varnothing}$ by solving the following linear program, denoted $\mathcal{LP}_{\boldsymbol{S}}$, where x_{iv} are relaxed to real values:

$$\min z = \sum_{i \in \boldsymbol{S}, v \in \boldsymbol{D}_i} (\delta_{iv} + c_i(v)) x_{iv} - \delta_{\emptyset} \tag{1a}$$

$$\sum_{i \in \boldsymbol{S}, v \in \boldsymbol{D}_i} w_{iv} x_{iv} \geq C \tag{1b}$$

$$\sum_{v \in \boldsymbol{D}_i} x_{iv} = 1, \qquad \forall i \in \boldsymbol{S} \tag{1c}$$

$$x_{iv} \in [0, 1], \quad \forall i \in \boldsymbol{S}, v \in \boldsymbol{D}_i \tag{1d}$$

This is the linear relaxation of a Multiple-Choice Knapsack Problem (MCKP), which can be solved efficiently in $O(\sum_{i \in \boldsymbol{S}} |\boldsymbol{D}_i|)$ [24]. Its dual, denoted $\mathcal{LD}_{\boldsymbol{S}}$, is:

$$\max z = Cy_{cc} + \sum_{i \in \boldsymbol{S}} y_i - \delta_{\emptyset} \tag{2a}$$

$$w_{iv} y_{cc} + y_i \leq \delta_{iv} + c_i(v), \quad \forall i \in \boldsymbol{S}, v \in \boldsymbol{D}_i \tag{2b}$$

$$y_{cc} \geq 0 \tag{2c}$$

where y_{cc} (resp. y_i) are real variables associated to constraints (1b) (resp. (1c)) of the primal. $\mathcal{LP}_{\boldsymbol{S}}$ and $\mathcal{LD}_{\boldsymbol{S}}$ have the same optimum (*strong duality property*). From an optimal solution $z^*, \mathbf{x}^* = \{x_{iv}^* \mid i \in \boldsymbol{S}, v \in \boldsymbol{D}_i\}$ of $\mathcal{LP}_{\boldsymbol{S}}$, we can directly deduce an optimal solution $\mathbf{y}^* = \{y_{cc}^*, y_i^* \mid i \in \boldsymbol{S}\}$ of $\mathcal{LD}_{\boldsymbol{S}}$.

Given a dual solution $\mathbf{y}^*$, the reduced cost associated with assignment (i, v) is $rc_{\boldsymbol{S}}^{\mathbf{y}^*}(i, v) = \delta_{iv} + c_i(v) - w_{iv} y_{cc}^* - y_i^*$ and corresponds to the slack of the dual constraint (2b). This value can be interpreted as a lower bound on the difference in the objective between any feasible solution x with $x_{iv} > 0$ and $\mathbf{x}^*$. We have $z - z^* \geq rc_{\boldsymbol{S}}^{\mathbf{y}^*}(i, v)$.

In CFNs, the reduced cost provides a lower bound on the minimal cost tuple $\tau \in \ell(\boldsymbol{S})$ with $\tau[i] = v$ considering only $c_{\boldsymbol{S}}$

Procedure LinMoveCost($c_i, c_{\boldsymbol{S}}, v, \alpha$)
_____
Input: scopes s.t. $i \in \boldsymbol{S}$; $v \in \boldsymbol{D}_i$; cost α to move
1   $c_i(v) \leftarrow c_i(v) + \alpha$;
2   $\delta_{iv} \leftarrow \delta_{iv} - \alpha$

Procedure LinProject($c_{\boldsymbol{S}}, \alpha$)
_____
1   $c_{\varnothing} \leftarrow c_{\varnothing} + \alpha$;
2   $\delta_{\emptyset} \leftarrow \delta_{\emptyset} + \alpha$

---

and the unary costs $c_i, \forall i \in \boldsymbol{S}$. If $z^* > 0$ then it is possible to derive a set of EPTs, $\{extend(c_i, c_{\boldsymbol{S}}, v, c_i(v) - rc_{\boldsymbol{S}}^{\mathbf{y}^*}(i, v)) \mid i \in \boldsymbol{S}, v \in \boldsymbol{D}_i\} \bigcup \{LinProject(c_{\boldsymbol{S}}, z^*)\}$, increasing $c_{\varnothing}$ by z^* [7]. In the following, since we manipulate only one dual solution $\mathbf{y}^*$ at a time, we omit $\mathbf{y}^*$ and write $rc(i, v)$.

III. VAC ON LINEAR CONSTRAINTS

A limitation of the propagation method for linear constraints introduced in [7] is that constraints are propagated one-by-one and communicate only via unary cost functions. Therefore, once a linear constraint has absorbed a cost in some δ, it becomes invisible to other cost functions. Moreover, the quality of the lower bound depends largely on the propagation order. Enforcing VAC_θ allows the detection of a sequence of EPTs resulting from a combination of several constraint propagations without a fixed propagation order. However, VAC_θ requires enforcing GAC on the linear constraints in $\text{Bool}_\theta(\text{P})$. This requires verifying for each $c_{\boldsymbol{S}} \in \boldsymbol{C}, i \in \boldsymbol{S}, v \in \boldsymbol{D}_i$, whether there exists a tuple $\tau \in \ell(\boldsymbol{S}), \tau[i] = v$ such that $c_{\boldsymbol{S}}(\tau) < \theta$. Specifically, each linear constraint $c_{\boldsymbol{S}}$ is transformed in $\text{Bool}_\theta(\text{P})$ into the following hard constraint:

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

Propagating this requires solving a Knapsack problem. This problem is NP-hard, but several approaches have been studied, including dynamic programming for enforcing GAC [25], approximate filtering with a *fully polynomial time approximation scheme* [26], [27], and linear programming-based reduced cost filtering [8], [28].

We show in the following section that we can use bounds propagation, an optimal dual solution, and reduced cost filtering [8] to detect a subset of inconsistent tuples in $\text{Bool}_\theta(\text{P})$.

A. Filtering for Linear Constraints with Assignment Costs

Observe that in (3), a tuple τ is forbidden either if it violates the constraint (*i.e.,* the sum of the selected weights is less than the capacity), or if the delta costs are greater than or equal to the VAC threshold ($\sum_{i \in \boldsymbol{S}} \delta_{i\tau[i]} - \delta_{\emptyset} \geq \theta$).

In $\text{Bool}_\theta(\text{P})$, we classify the removal of a value (i, a) by a hard linear constraint $\text{Bool}_\theta(c_{\text{S}})$ as either hard or soft. A removal is *hard* if no feasible tuple exists where variable i takes value a whatever the delta costs and θ values are, *i.e.,* $c_{\boldsymbol{S}}(\tau) = \top, \forall \tau \in \ell(\boldsymbol{S})$ s.t. $\tau[i] = a$. A removal is *soft* if $\forall \tau \in \ell(\boldsymbol{S})$ s.t. $\tau[i] = a$, we have $c_{\boldsymbol{S}}(\tau) \geq \theta$. Strategies for detecting and explaining hard and soft removals differ.

Hard value removals can be detected by enforcing bounds consistency on $\text{Bool}_\top(c_{\text{S}})$, which can be performed in linear time for inequality constraints [29].

To detect a set of values that can be soft removed from $\text{Bool}_\theta(\text{P})$, we solve for each linear constraint $c_{\boldsymbol{S}}$ a modified version of $\mathcal{LP}_{\boldsymbol{S}}$ where $c_i(v)$ has been removed in (1a). This modified problem is called $\widetilde{\mathcal{LP}}_{\boldsymbol{S}}$ and an optimal solution is denoted $\tilde{z}^*, \tilde{\mathbf{x}}^*$. In $\text{Bool}_\theta(\text{P})$, unary cost functions are replaced

by (reduced) domains. After enforcing GAC on $\mathrm{Bool}_\theta(c_i)$, we have $\mathrm{Bool}_\theta(c_i)(v) = 0, \forall v \in D_i$ (other values where $c_i(v) \geq \theta$ have been removed). The dual of $\widetilde{\mathcal{LP}}_{\boldsymbol{S}}$ is obtained by removing $c_i(v)$ in (2b); we call it $\widetilde{\mathcal{LD}}_{\boldsymbol{S}}$. The reduced costs obtained from solving $\widetilde{\mathcal{LP}}_{\boldsymbol{S}}$ and $\widetilde{\mathcal{LD}}_{\boldsymbol{S}}$ allow to filter domains. Specifically, $\tilde{z}^* + rc(i,v)$ provides a valid lower bound on the minimal cost of any tuple $\tau \in \ell(\boldsymbol{S})$ with $\tau[i] = v$, accounting for $c_{\boldsymbol{S}}$ and the current value removals. When $\tilde{z}^* + rc(i,v) \geq \theta$, (i,v) can safely be removed from $\mathrm{Bool}_\theta(\mathrm{P})$.

Note that this method does not enforce complete GAC. Achieving GAC would require solving an MCKP for each value, which, in practice, is too costly when filtering $\mathrm{Bool}_\theta(\mathrm{P})$.

B. Finding Explanations for Value Removals

A further requirement of VAC is an *explanation* for each value removal, primarily during phase 2. An explanation $\langle c_{\boldsymbol{S}}, E \rangle$ for the removal of value (i,a) is a set of values E whose removal implies the removal of (i,a) by arc consistency on a constraint $c_{\boldsymbol{S}}$. An explanation is minimal if no proper subset of E is an explanation.

For each hard value removal, a minimal explanation is generated using conflict explanation [30].

For each soft removal of a value (i,a) by a linear constraint $c_{\boldsymbol{S}}$, we need to identify a subset of the previous value removals that are necessary to ensure that $\tilde{z}^* + rc(i,a) \geq \theta$. Let Q be the list of previous value removals made by VAC; it provides a trivial non-minimal explanation. In the second phase of VAC, we try to improve this explanation, by solving $\widetilde{\mathcal{LP}}_{\boldsymbol{S}}^{ia}$, a modified version of $\widetilde{\mathcal{LP}}_{\boldsymbol{S}}$ with additional constraints $x_{ia} = 1$ and $x_{jb} = 0$ for all $(j,b) \in Q$. An optimal solution is denoted $\tilde{z}^{ia,*}, \tilde{\mathbf{x}}^{\mathbf{ia},*}$. We have $\tilde{z}^{ia,*} \geq \tilde{z}^* + rc(i,a) \geq \theta$. From this optimal primal solution, we can compute an optimal dual solution $\tilde{\mathbf{y}}^{\mathbf{ia},*}$ of the dual of $\widetilde{\mathcal{LP}}_{\boldsymbol{S}}^{ia}$, denoted as $\widetilde{\mathcal{LD}}_{\boldsymbol{S}}^{ia}$. Notice that constraints $x_{jb} = 0$ introduce new dual variables in $\widetilde{\mathcal{LD}}_{\boldsymbol{S}}^{ia}$ resulting in unbounded constraints (*i.e.,* with infinite right hand-side in Eq.(2b)). This prevents meaningful interpretation of the reduced costs associated to the values in Q. Instead, we choose to tighten their dual constraints by keeping δ_{jb} for the right-hand side in Eq.(2b), as in $\widetilde{\mathcal{LD}}_{\boldsymbol{S}}$. Thus, we compute $rc(j,b) = \delta_{jb} - w_{jb}\tilde{y}_{cc}^{ia,*} - \tilde{y}_j^{ia,*}$ for all $(j,b) \in Q$. By doing so, we lose the strong duality property, but still produce valid lower bounds.

Now, if $rc(j,b) \geq 0$, we can be certain that the optimal cost $\tilde{z}^{ia,*}$ is greater than or equal to θ independently of whether (j,b) is removed or not. Consequently, (j,b) cannot be part of a minimal explanation for the removal of (i,a). Otherwise, if $rc(j,b) < 0$, we include (j,b) in the explanation. While this approach may result in a non-minimal explanation, finding a minimal one would require significantly more computation. Indeed, reduced costs account only the additional cost of modifying a single value and do not capture interactions when multiple values are modified together.

C. VAC-lin Subroutines

Here, we define VAC-lin, a local consistency obtained by enforcing in $\mathrm{Bool}_\theta(\mathrm{P})$ the filtering process of Sec. III-A on the linear constraints and GAC on the other constraints. Any value removed by the filtering process of Sec. III-A would have been removed by enforcing GAC on the linear constraints. Thus, the following corollary of Theorem 1 holds.

Corollary 1. *Let P be a WCSP such that $c_\emptyset < \top$. If enforcing an incomplete GAC 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 EPTs which when applied to P leads to an increase in $c_\emptyset$.*

VAC-lin can be enforced by integrating the techniques of Section III-A into the VAC algorithm [4]. Algorithm 1 presents the main VAC algorithm and Algorithm 2 gives the functions specific to VAC-lin. For functions specific to table constraints, the reader can refer to [4].

Filtering Phase: The function `VAC-Filter` in Alg. 1 corresponds to the filtering phase, it considers $\mathrm{Bool}_\theta(\mathrm{P})$ and enforces an incomplete GAC. It ends when no more values can be removed or when a conflict appears. It uses a queue R containing all constraints that require propagation. Initially, R includes every constraint. Whenever a value (i,a) has no support on a constraint $c_{\boldsymbol{S}}$, it is removed and added to second queue Q. Additionally, we record the constraint responsible for this removal in a dedicated structure called killer. All constraints other than $c_{\boldsymbol{S}}$ that involve the removed value (i,a) must then be propagated again. Both the queue Q and structure killer are useful only for the second phase `VAC-Tracer`.

The function `Filter(Bool`$_\theta$`(c`$_S$`))` applies a filtering algorithm to the cost function $c_{\boldsymbol{S}}$. It returns a cost and a set of variables describing either a conflict or a set of inconsistent values. Filtering of linear constraints is presented in function `LinFilter(Bool`$_\theta$`(c`$_S$`))` of Algorithm 2 and relies on the techniques described in Section III-A. Given a linear constraint $c_{\boldsymbol{S}}$, propagation starts by enforcing bounds arc consistency and solving $\widetilde{\mathcal{LP}}_{\boldsymbol{S}}$. If it is conflicting (optimal cost $\geq \theta$ or inconsistent constraint), then the values involved in the conflict are identified by analysing the reduced costs, or using conflict explanation [30]. Otherwise, if no conflict occurs, reduced costs obtained from $\widetilde{\mathcal{LP}}_{\boldsymbol{S}}$ are analyzed to remove inconsistent values.

Tracing Phase: Either the first phase ends with a conflict in $\mathrm{Bool}_\theta(\mathrm{P})$ or VAC-lin terminates. The purpose of `VAC-Tracer` is to identify a minimal subset of value removals sufficient to explain this conflict. To achieve this, values involved in the conflict are marked using a Boolean function M. The objective is to identify a set of values/tuples with non-zero costs that can serve as *sources* to move cost to the marked values. Initially, only values returned by `VAC-Filter` with zero unary cost are marked. Then `VAC-Tracer` exploits the queue Q and the killer data structure to rewind the propagation history. Values are popped one by one from Q and if it is marked, we trace back the cause of its deletion.

Identifying possible sources for a marked value (i,a) can be done by computing an explanation $\langle c_{\boldsymbol{S}}, E \rangle$ for its removal in $\mathrm{Bool}_\theta(\mathrm{P})$. According to Corollary 1, moving costs

Algorithm 1: VAC general algorithm

```
   // 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.
 1 Function(VAC-Filter())
 2    R ← C ;
 3    while R ≠ ∅ do
 4       c_S ← R.Pop() ;
 5       ⟨z̃*, E⟩ ← Filter(Bool_θ(c_S)) ;
 6       if z̃* ≥ θ then return ⟨z̃*, c_S, E⟩ ;
 7       foreach (i, a) ∈ E do
 8          Delete a from D_i ;
 9          killer(i, a) ← c_S ;
10          Q.Push(i, a) ;
11          if D_i = ∅ then return ⟨⊤, c_i, {(i, b) | b ∈ D_i^copy}⟩;
12          else R ← R ∪ {c_S' | c_S' ∈ C, c_S' ≠ c_S, i ∈ S'} ;
13    return ⟨0, ∅, ∅⟩ ;
   // Compute λ the maximal cost movable to c_∅.
14 Function(VAC-Tracer())
15    Q ← ∅, D^copy ← D ;
16    ⟨λ, c_S, E⟩ ← VAC-Filter() ;
17    D ← D^copy ;
18    if c_S ≠ ∅ then k_{c_S} ← 1 ;
19    foreach (i, a) ∈ E do
20       k(i, a) ← 1, M(i, a) ← true ;
21       if c_i(a) > 0 then M(i, a) ← false, λ ← min(λ, c_i(a)) ;
22    while Q ≠ ∅ do
23       (i, a) ← Q.Pop() ;
24       if M(i, a) then
25          c_S ← killer(i, a) ;
26          E ← Explanation(c_S, (i, a)) ;
27          foreach (j, b) ∈ E do
28             k(j, b) ← k(j, b) + k(i, a) ;
29             k_{c_S}(j, b) ← k_{c_S}(j, b) + k(i, a) ;
30             if c_j(b) = 0 then M(j, b) ← true ;
31             else λ ← min(λ, c_j(b)/k(j,b)) ;
32    return λ ;
```

from the values in E to the constraint c_S enables subsequent cost transfers from c_S to $c_i(a)$. The constraint responsible for the removal of (i, a) in VAC-Filter is stored in killer(i, a). Explanations are computed by function Explanation$(c_S, (i, a))$. For linear constraints, this is presented in function LinExplanation$(c_S, (i, a))$ and relies on techniques described in III-B. Specifically, the linear program $\widetilde{\mathcal{LP}}_S^{ia}$ is solved. If it is infeasible, an explanation is generated using conflict explanation techniques [30], otherwise, the explanation is based on reduced costs analysis. Whenever values in the explanation set E have zero unary cost, they are marked for further consideration. Since values in queue Q are visited in reverse chronological order of their removal from Bool$_θ$(P), all values in any explanation E will still be present in Q and will be visited later in the process. This ensures that the algorithm can systematically reconstruct a minimal explanation for the original conflict.

The maximum amount of cost λ movable to $c_∅$ depends on the costs available at each source and the number of operations involving the marked values. Indeed, a single value can contribute to multiple removals, causing its available cost to be distributed among several cost functions. To keep track of this, we maintain three counters. The counter $k(i, a)$ records

the number of request made by value (i, a), while $k_{c_S}(i, a)$ tracks the number of request that (i, a) must extend to c_S. These counters are initialized to 0 and updated each time (i, a) is involved in a removal. Ultimately, a sequence/set of EPTs can be obtained by projecting a cost $k(i, a) \times \lambda$ from killer(i, a) to $c_i(a)$ and extending a cost $k_{c_S}(i, a) \times \lambda$ from $c_i(a)$ to c_S, for all (i, a) with nonzero k structure. Then, a final EPT is to project a cost λ from the conflicting constraint found by VAC-Filter to $c_∅$. These EPTs will be applied in VAC phase 3. Ideally, to ensure that no negative costs are introduced, it would be useful to maintain a third counter keeping track of the number of request made by each tuple in each linear constraint. However, since the number of tuples increases exponentially with the arity of the constraint, this approach is impractical for constraints of large arity. Instead, we introduce a counter k_{c_S} giving an upper bound on the maximal number of request made by all the tuples of c_S. This counter is only used to guarantee the correctness of λ, it is initialized 0 and updated only when the constraint is involved in a conflict.

The value of λ is the largest amount of cost that can be moved while still satisfying all requests. For example, if a value (i, a) has an available cost of 4 and $k(i, a) = 2$, then $\lambda \leq \frac{4}{2} = 2$. Initially, λ cannot exceed the cost returned by the filtering phase; it is then as counters increase. Finally, in

Algorithm 2: Functions specific to VAC-lin

```
 1 Function(LinFilter(Bool_θ(c_S)))
 2    Hard_Rem ← Bounds-Arc-Consistency(Bool_⊤(c_S)) ;
 3    z̃* ← Solve(𝓛̃𝓟_S) ;
 4    Compute reduced costs from 𝓛̃𝓓_S ;
 5    if 𝓛̃𝓟_S is infeasible then
 6       E ← Conflict-explanation ;
 7       return ⟨⊤, E⟩ ;
 8    if z̃* ≥ θ then
 9       E ← {(j, b) ∈ Q | rc(j, b) < 0} ;
10       return ⟨z̃*, E⟩ ;
11    Soft_Rem ← {(i, v) | i ∈ S, v ∈ D_i, z̃* + rc(i, v) ≥ θ} ;
12    return ⟨0, Hard_Rem ⋃ Soft_Rem⟩ ;
13 Function(LinExplanation(c_S, (i, a)))
14    z̃^{ia,*} ← Solve(𝓛̃𝓟_S^{ia}) ;      /* Assert: z̃^{ia,*} ≥ θ */
15    Compute reduced costs from 𝓛̃𝓓_S^{ia} ;
16    if 𝓛̃𝓟_S^{ia} is infeasible then E ← Conflict-explanation ;
17    else E ← {(j, b) | rc(j, b) < 0} ;
18    k_{c_S} ← k_{c_S} + k(i, a) ;
19    λ ← min(λ, z̃^{ia,*}/k_{c_S}) ;
20    return E ;
```

the third phase, all EPTs are performed according to killer, and the different k structures as explained before. After this sequence of EPTs, we know a cost of λ can be moved to $c_∅$. Example 2 illustrates one iteration of VAC-lin.

The space complexity of VAC-lin is the same as that of the original VAC [4]. As for time complexity, it is dominated by the filtering process. It requires solving for each constraint a relaxed knapsack problem in $O(rd)$ time [24], where r is the arity of the largest linear constraint, and d is the largest domain size. The number of propagations is at most nd,

with n the number of variables. Thus, the total complexity of `VAC-Filter` is $O(mnrd^2)$ where m is the number of linear constraints. Enforcing GAC on e' table constraints requires $O(e'd^{r'})$ time where r' is the arity of the largest table constraints. In practice, VAC is often enforced on binary table constraints ($r' = 2$), thus the complexity is often dominated by the filtering phase of the linear constraints. Therefore, GAC is enforced on table constraints before filtering linear constraints.

Example 2. *Let P be a WCSP with 6 variables, with domains $\{a, b\}$, and 6 constraints $c_{12345} : 7x_{1a} + 7x_{2a} + 3x_{3a} + 3x_{4a} + 3x_{5a} \geq 10$, $c_{14} : x_{1a} + x_{4b} \geq 1$, $c_{246} : x_{2b} + x_{4a} + 2x_{6a} \geq 1$ and $c_1(a) = 2$, $c_3(a) = 2$, $c_6(a) = 2$, all other tuples having cost 0. Propagating the constraints as done in [7] 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 with a threshold $\theta = 1$ increases $c_\varnothing$ by 1.*
In $\mathrm{Bool}_\theta(P)$, $(1, a)$, $(3, a)$ and $(6, a)$ are directly removed, it follows by bounds propagation on c_{12345} that $(2, b)$ can be removed and we set $\mathsf{killer}(2, b) = c_{12345}$. Similarly, $(4, b)$ is removed by bounds propagation on c_{246} and we set $\mathsf{killer}(4, b) = c_{246}$. Finally c_{14} is inconsistent with explanation $\{(1, a), (4, b)\}$; thus, $\mathrm{Bool}_\theta(P)$ is not GAC.
We set $\lambda = \top$ and start tracing back the GAC operations. c_{14} is inconsistent because $(1, a)$ and $(4, b)$ have been removed. The k structures are updated: $k(1, a) = k_{c_{14}}(1, a) = k(4, b) = k_{c_{14}}(4, b) = k_{c_{14}} = 1$. We immediately 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: $M(4, b) = \mathsf{true}$ and must be traced. Value $(4, b)$ has been removed because it has no support on c_{246}, the solver computes the minimal explanation $\{(2, b), (6, a)\}$ using conflict explanation [30]. The k structures are updated: $k(2, b) = k_{c_{246}}(2, b) = k(6, a) = k_{c_{246}}(6, a) = k_{c_{246}} = k(4, b) = 1$. 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 $c_2(b) = 0$; the value is marked: $M(2, b) = \mathsf{true}$ and must be traced. Value $(2, b)$ has been removed because it has no support on c_{12345}, the solver computes the minimal explanation $\{(1, a)\}$ using conflict explanation [30]. We update the k structures: $k(1, a) = k(1, a) + k(2, b) = 2$, $k_{c_{12345}} = k_{c_{12345}}(1, a) = 1$. We update λ: $\lambda = \frac{c_1(a)}{k(1,a)} = 1$. No more values are marked, the conflict has been explained.
We deduce the following EPTs from killer, k structures and λ:

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

After reformulation, we have $\delta_{1a}^{c_{12345}} = \delta_{2b}^{c_{246}} = \delta_{6a}^{c_{246}} = \delta_{4b}^{c_{14}} = \delta_{1a}^{c_{14}} = \delta_{\varnothing}^{c_{14}} = 1$, $\delta_{2b}^{c_{12345}} = \delta_{4b}^{c_{246}} = -1$, and $c_3(a) = 2, c_6(a) = c_\varnothing = 1$, all other tuples and c_1 having cost 0.

IV. EXPERIMENTAL RESULTS

We implemented VAC-lin in toulbar2, an open-source C++ WCSP solver.[2] The original VAC algorithm was already

[2]https://github.com/toulbar2/toulbar2 version 1.2.1.

implemented in the solver, but only for binary cost functions in extension, with VAC maintained incrementally inside each search node [31]. Here, we test three variants of toulbar2. The first, a base version, applies a weaker SAC algorithm (EDAC [21] and partial F∅IC for linear constraints [7]) at every search node of a hybrid best/depth-first branch-and-bound search method [32]. These are the default settings of toulbar2 and we denote this configuration as no-VAC. The second version, denoted VAC, is based on no-VAC and additionally applies the existing version of VAC (which ignores linear constraints) in preprocessing. The third version, denoted VAC-lin, applies during preprocessing VAC with our modifications to make it take linear constraints into account. We compared these three 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 toulbar2 used the same *dom/wdeg* variable ordering heuristic [33] with last conflict [34]. The value ordering heuristic is the minimum domain value for choco and EAC/VAC/VAC-lin *support value* for toulbar2 [4], [35]. In VAC and VAC-lin, this corresponds to choosing first the minimum domain value in Bool(P) after doing the filtering phase. Both solvers use solution phase saving [36].

To test our approach with a large number of linear constraints, we chose integer linear problems from the MIPLIP 2017 benchmark. We also tested the Computational Protein Design (CPD) and Quadratic Assignment Problem (QAPLIB) benchmarks, which have few additional linear constraints, large domains, and several binary cost functions in extension. We also tested 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. For CPD, QAPLIB, and XCSP, we used a support encoding for cplex [12]. We used table constraints for choco to encode the quadratic objective of CPD and QAPLIB.

Experiments on MIPLIB were done 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 / QAPLIB were run on a single core of an Intel Xeon E5-2680 v3 at 2.5GHz with 64GB and $3,600$s / $2,400$ / $1,800$ / $1,200$ limits respectively.[4]

A. MIPLIB 2017 0/1LP

We selected 200 instances from the MIPLIB 2017 collection, containing only Boolean $0/1$ variables. Among them, 184 instances have known feasible solution.[5] We preprocessed them using cplex and applied our methods to the preprocessed instances.[6] In Tab. I, we report the average quality of lower

[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*.

[4]For CPD, we add the option *-d:* in toulbar2 to remove its default dichotomic branching rule.

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

[6]We transformed the original real cost values in fixed-precision integer costs using 3 digits after decimal.

benchmark	total	# av.	no-VAC	VAC	VAC-lin	LP
MIPLIB 2017	184	147	59.51% (157)	59.23% (158)	65.09% (147)	**82.20% (179)**
CPD	30	25	95.56% (**30**)	97.37% (**30**)	97.74% (**30**)	**98.13%** (25)
PB'2007	77	77	62.67% (**77**)	63.65% (**77**)	83.93% (**77**)	**86.44% (77)**
XCSP'2022	158	100	19.72% (**136**)	21.48% (**136**)	21.78% (**136**)	**41.47%** (100)
XCSP'2023	155	88	28.17% (**137**)	27.65% (**137**)	27.87% (132)	**53.65%** (93)
QAPLIB	132	132	5.16% (**132**)	5.16% (**132**)	11.07% (**132**)	**12.77% (132)**

TABLE I: Quality of lower bounds per benchmark averaged over the number of instances (column *# av.*) where all methods produced a lower bound in space and time limits. In parentheses, number of instances where a particular method produced a lower bound.

benchmark	total	choco	cplex	toulbar2-no-VAC	toulbar2-VAC	toulbar2-VAC-lin
MIPLIB 2017	184	14	**100**	17	16	17
CPD	30	0	19	26	**28**	**28**
PB'2007	77	16	**67**	56	58	**67**
XCSP'2022	158	41	**63**	54	54	57
XCSP'2023	155	20	**39**	17	17	22
QAPLIB	132	17	22	31	31	**32**

TABLE II: Number of solved instances per benchmark.

bounds for our three variants, no-VAC, VAC, and VAC-lin, and also for the continuous linear relaxation found by cplex (column LP in Tab. I).[7] As expected, the linear relaxation provides 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 15% of the instances, indicating that substantial engineering work remains to reach the same efficiency level as a commercial state-of-the-art LP solver. Although the original VAC algorithm is not advantageous on this benchmark due to the limited number of arity-2 linear constraints, our VAC-lin significantly improves the initial bound, going from 59% to 65% on average. However, it was insufficient to solve more instances for this benchmark (17 solved instances in total, whereas cplex solved 100). Choco performed poorly, solving 14 instances (Tab. II and Supp. Fig. 1.a).

B. Computational Protein Design with Diversity Constraints

As in [7], we selected 30 CPD instances with 23-97 variables and largest domain sizes 48-194. For each instance, ten diverse solutions with a Hamming distance ten were generated using toulbar2 with a dual encoding [37]. 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 0.22%. VAC and VAC-lin produced almost the same results, solving 28 instances optimally. VAC-lin improved the initial lower bound found by VAC in one-third of the instances (Tab. I). The absolute initial gap was reduced by 9.93%, when moving from VAC to VAC-lin. However, it did not significantly reduce the number of search nodes or solving time. A different behavior was observed between VAC and VAC-lin when applying the additional upper-bound preprocessing RASPS [35]. VAC-rasps solved 29 and VAC-lin-rasps

solved 30 instances. Finally, no-VAC solved 26 and cplex 19 instances.

C. Pseudo Boolean 2007 OPT-SMALLINT-LIN-Other

We ran experiments on 77 instances introduced at PB 2007 Evaluation. They are unweighted Max-SAT instances with 66.3% of arity-2 clauses, 25% of arity-3, and the remainder from arity 4 up to 3,140.[8] Cplex obtained the best results, solving 67 instances within the CPU-time limit of 1,800s; VAC-lin also solved 67 instances but was much slower than CPLEX (Supp. Fig. 1.c).For this benchmark, VAC-lin clearly dominates VAC and no-VAC, which solved 58 and 56 instances respectively.[9] The largest instance solved by VAC-lin (*aksoy/normalized-fir08_area_delay*) has 124,856 variables and 521,620 clauses of maximum arity 1,023. Choco did not perform well, solving only 16 instances. However, it found better solutions on the unsolved *aksoy/decomp* instances than the other competitors.

D. XCSP 2022 and 2023 MiniCOP Competition

We restricted experiments to the mini COP category of the 2022 and 2023 XCSP competitions.[10]

Although the lower bound quality of VAC-lin is slightly better than VAC (Tab. I), it is much higher in some particular families (XCSP22/CoinsGrid, XCSP23/Auctions) where the solving time was greatly reduced compared to VAC. Thus, VAC-lin solved slightly more instances than VAC or no-VAC (Tab. II). It also performed better than or similar to choco depending on the benchmark.[11]

The strong results obtained by cplex are not surprising. They were already observed in past MiniZinc Challenges.

E. Quadratic Assignment Problem Library

We took 132 instances from the QAPLIB.[12] For a problem of size N, we expressed the quadratic objective function as a binary Weighted CSP with N variables of domain size N. The permutation constraint is encoded for toulbar2 and cplex as forbidden tuples for any pair of two variables (i.e., they cannot take the same value), and N redundant (generalized) linear constraints of arity N are added to ensure that each value is assigned to at least one variable. In Choco, it is encoded as an AllDifferent constraint.

Within the CPU-time limit of 1,200 seconds, compared to no-VAC and to VAC, VAC-lin solved the same subset of 31 instances twice as fast (48.37s on average for VAC-lin, 80.8s for VAC and 84.5s for no-VAC) and it solved one additional

---

[7]The quality of an initial lower bound l for a given instance with best-known solution value b and trivial lower bound $t < b$ (computed as the sum of the minimum of each cost function) is defined by $(l - t)/(b - t)$. We report average quality over the number of successful instances producing a bound at the root search node for all the tested methods.

[8]http://www.cril.univ-artois.fr/PB07/benchs/PB07-OTHER.tar

[9]VAC and no-VAC didn't solve *normalized-f20c10b_001_area_delay* whereas VAC-lin solved it in 25.7s and cplex in 14.5s.

[10]https://xcsp.org/competitions

[11]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 added solution phase saving.

[12]http://coral.ise.lehigh.edu/wp-content/uploads/2014/07/qapdata.tar.gz, excluding four instances (*esc128, tai150b, tai256c, tho150*) of size strictly greater than 100.

instance (*chr25a*).[13] Although it has a theoretically stronger lower bound, cplex solved only 22 instances. Choco solved 17 instances.

V. Conclusion

Although VAC-lin improved the initial lower bound compared VAC, in many cases it was insufficient to obtain significant speedups, except on PB07, QAPLIB and on some particular categories of XCSP. Applying a stronger soft arc consistency algorithm during the search can pay off for some difficult instances [39]. Testing VAC-lin in such situations remains future work. In the future, we plan to apply the methodology developed for linear constraints to other global constraints, such as AllDifferent.

Acknowledgment

References

[1] M. Cooper, S. de Givry, and T. Schiex, "Graphical models: queries, complexity, algorithms," *LIPIcs*, vol. 154, pp. 4–1, 2020.

[2] F. Rossi, P. Van Beek, and T. Walsh, *Handbook of constraint programming*. Elsevier, 2006.

[3] R. Dechter and I. Rish, "Mini-buckets: A general scheme for bounded inference," *Journal of the ACM*, vol. 50, no. 2, pp. 107–153, 2003.

[4] M. C. Cooper, S. de Givry, M. Sánchez, T. Schiex, M. Zytnicki, and T. Werner, "Soft arc consistency revisited," *Artificial Intelligence*, vol. 174, no. 7-8, pp. 449–478, 2010.

[5] D. Allouche, C. Bessiere, P. Boizumault, S. De Givry, P. Gutierrez, J. H. Lee, K. L. Leung, S. Loudni, J.-P. Métivier, T. Schiex *et al.*, "Tractability-preserving transformations of global cost functions," *Artificial Intelligence*, vol. 238, pp. 166–189, 2016.

[6] S. de Givry and G. Katsirelos, "Clique cuts in weighted constraint satisfaction," in *Proc. of CP-17*, Melbourne, Australia, 2017, pp. 97–113.

[7] P. Montalbano, S. de Givry, and G. Katsirelos, "Multiple-choice knapsack constraint in graphical models," in *Proc. of CPAIOR-22, Los Angeles, CA, USA*, 2022, pp. 282–299.

[8] F. Focacci, A. Lodi, and M. Milano, "Cost-based domain filtering," in *Proc. of CP-99, Alexandria, VA, USA*, 1999, pp. 189–203.

[9] M. C. Cooper, S. de Givry, and T. Schiex, *Valued Constraint Satisfaction Problems*. Springer International Publishing, 2020, pp. 185–207.

[10] V. Kolmogorov, "Convergent tree-reweighted message passing for energy minimization," *IEEE transactions on pattern analysis and machine intelligence*, vol. 28, no. 10, pp. 1568–1583, 2006.

[11] P. Vaidya, "Speeding-up linear programming using fast matrix multiplication," in *30th Annual Symposium on Foundations of Computer Science*, 1989, pp. 332–337.

[12] B. Hurley, B. O'Sullivan, D. Allouche, G. Katsirelos, T. Schiex, M. Zytnicki, and S. de Givry, "Multi-language evaluation of exact solvers in graphical model discrete optimization," *Constraints*, vol. 21, no. 3, pp. 413–434, 2016.

[13] D. Prusa and T. Werner, "Universality of the local marginal polytope," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Portland, OR, USA*, 2013, pp. 1738–1743.

[14] T. Werner, "A Linear Programming Approach to Max-sum Problem: A Review." *IEEE Trans. on Pattern Recognition and Machine Intelligence*, vol. 29, no. 7, pp. 1165–1179, Jul. 2007.

[15] D. Sontag, T. Meltzer, A. Globerson, Y. Weiss, and T. Jaakkola, "Tightening LP relaxations for MAP using message-passing," in *Proc. of UAI*, Helsinki, Finland, 2008, pp. 503–510.

[16] N. Komodakis, N. Paragios, and G. Tziritas, "MRF energy minimization and beyond via dual decomposition," *IEEE transactions on pattern analysis and machine intelligence*, vol. 33, no. 3, pp. 531–552, 2010.

[17] D. Sontag, D. Choe, and Y. Li, "Efficiently searching for frustrated cycles in MAP inference," in *Proc. of UAI*, Catalina Island, CA, USA, 2012, pp. 795–804.

[18] S. Tourani, A. Shekhovtsov, C. Rother, and B. Savchynskyy, "Taxonomy of dual block-coordinate ascent methods for discrete energy minimization," in *Proc. of AISTATS-20*, Palermo, Sicily, Italy, 2020, pp. 2775–2785.

[19] T. Schiex, "Arc consistency for soft constraints," in *Proc. of CP-00*, Singapore, 2000, pp. 411–424.

[20] J. Larrosa, "On arc and node consistency in weighted CSP," in *Proc. of AAAI-02*, Edmondton, CA, USA, 2002, pp. 48–53.

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

[22] M. Zytnicki, C. Gaspin, S. de Givry, and T. Schiex, "Bounds Arc Consistency for Weighted CSPs," *Journal of Artificial Intelligence Research*, vol. 35, pp. 593–621, 2009.

[23] E. Boros and P. L. Hammer, "Pseudo-boolean optimization," *Discrete applied mathematics*, vol. 123, no. 1-3, pp. 155–225, 2002.

[24] D. Pisinger, "A minimal algorithm for the multiple-choice knapsack problem," *European Journal of Operational Research*, vol. 83, no. 2, pp. 394–410, 1995.

[25] M. A. Trick, "A dynamic programming approach for consistency and propagation for knapsack constraints," *Annals of Operations Research*, vol. 118, pp. 73–84, 2003.

[26] M. Sellmann, "Approximated consistency for knapsack constraints," in *Proc. of CP-03, Kinsale, Ireland*, 2003, pp. 679–693.

[27] ——, "The practice of approximated consistency for knapsack constraints," in *Proc. of AAAI-04, San Jose, CA, USA*, 2004, pp. 179–184.

[28] G. Claus, H. Cambazard, and V. Jost, "Analysis of reduced costs filtering for alldifferent and minimum weight alldifferent global constraints," in *Proc. of ECAI-20, Santiago de Compostela, Spain*, 2020, pp. 323–330.

[29] W. Harvey and J. Schimpf, "Bounds consistency techniques for long linear constraints," in *TRICS CP 2002 Workshop*, Ithaca, NY, USA, 2002, pp. 39–46.

[30] E. Hebrard and M. Siala, "Explanation-based weighted degree," in *Proc. of CPAIOR-17, Padua, Italy*, 2017, pp. 167–175.

[31] H. Nguyen, S. de Givry, T. Schiex, and C. Bessiere, "Maintaining virtual arc consistency dynamically during search," in *Proc. of ICTAI-14*, Limassol, Cyprus, 2014, pp. 8–15.

[32] 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*, Cork, Ireland, 2015, pp. 12–28.

[33] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais, "Boosting systematic search by weighting constraints," in *Proc. of ECAI-04, Valencia, Spain*, vol. 16, 2004, p. 146.

[34] C. Lecoutre, L. Saïs, S. Tabary, and V. Vidal, "Reasoning from last conflict(s) in constraint programming," *Artificial Intelligence*, vol. 173, pp. 1592,1614, 2009.

[35] F. Trösser, S. de Givry, and G. Katsirelos, "Relaxation-aware heuristics for exact optimization in graphical models," in *Proc. of CPAIOR-20*, Vienna, Austria, 2020, pp. 475–491.

[36] 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*, Lille, France, 2018, pp. 99–108.

[37] M. Ruffini, J. Vucinic, S. de Givry, G. Katsirelos, S. Barbe, and T. Schiex, "Guaranteed diversity and optimality in cost function network based computational protein design methods," *Algorithms*, vol. 4, no. 6:168, 2021.

[38] H. Zhang, C. Beltran-Royo, and L. Ma, "Solving the quadratic assignment problem by means of general purpose mixed integer linear programming solvers," *Annals of Operations Research*, vol. 207, pp. 261–278, 2013.

[39] P. Montalbano, D. Allouche, S. De Givry, G. Katsirelos, and T. Werner, "Virtual pairwise consistency in cost function networks," in *Proc. of CPAIOR-23, Nice, France*, 2023, pp. 417–426.

Appendix

See the full-paper version with supplementary Fig. 1 and Tables about solution quality on https://arxiv.org/abs/2509.17706

---

[13]VAC-lin solved *chr25a* in 114sec and 142,042 search nodes. cplex didn't end in 1,200s. The best 0/1LP approach reported in [38] (R-III) solved in 274s and 5,164 nodes using a Pentium 1.7GHz and cplex 9.0.