Revisiting Counting Solutions for the Global Cardinality Constraint

Giovanni Lo Bianco giovanni.lo-bianco@imt-atlantique.fr
IMT Atlantique,
4 Rue Alfred Kastler, 44300 Nantes, France

Xavier Lorca xavier.lorca@mines-albi.fr
IMT Mines Albi,
Allée des Sciences, 81000 Albi, France

Charlotte Truchet charlotte.truchet@univ-nantes.fr
UFR de Sciences et Techniques,
2, rue de la Houssinière, BP 92208, 44322 NANTES CEDEX 3, France

Gilles Pesant gilles.pesant@polymtl.ca
Ecole Polytechnique de Montréal,
2900 Boulevard Edouard-Montpetit, Montréal, QC H3T 1J4, Canada

Abstract

Counting solutions for a combinatorial problem has been identified as an important concern within the Artificial Intelligence field. It is indeed very helpful when exploring the structure of the solution space. In this context, this paper revisits the computation process to count solutions for the global cardinality constraint in the context of counting-based search (?). It first highlights an error and then presents a way to correct the upper bound on the number of solutions for this constraint.

1. Introduction

Model counting is a fundamental problem in artificial intelligence. It consists in counting the number of solutions of a given problem without solving the problem and it has numerous applications such as probabilistic reasoning and machine learning (?, ?). Counting solutions of a problem can also be helpful when exploring the structure of the solution space (?).

In the field of Constraint Programming (CP), we aim at solving hard combinatorial problems models called Constraint Satisfaction Problem (CSP). We are given a set of variables, which are the unknowns of the problem, a set of domains, which describe the possible value assignments for each variable and a set of constraints, which are mathematical relations between the variables. A constraint can simply be arithmetical constraints such as $x > y + 1$, or have a more complex structure, called global constraints, such as `alldifferent`. CP is based on the propagation-search paradigm. Each constraint has a propagator, that filters inconsistent values from domains. Once every informations have been propagated, a fixed point is reached and an assignment variable/value has to be chosen. And again, this assignment is propagated in the constraints network. If every variable have been instantiated, then a solution is found. If a propagator empty a domain, then the last assignment choice is reconsidered, we call it a backtrack. The propagator "intelligence" can be parametrized, it is called the propagator consistency. The

more consistent is the propagator, the more costly it is to filter the domains. It is sometimes worthy to have less consistency and let the search strategy operates. The search strategy, also called search heuristic, defines how the assignment variable/value is chosen when a fixed point is reached.

Search heuristics have been intensively studied during the last decades within constraint programming (?, ?, ?). While most of these approaches have used generic and/or problem-specific search strategies, the counting-based search heuristics (?) work directly with the combinatorial structure of the problem. The key idea is to guide the resolution process toward the most promising part of the search space according to the number of remaining solutions. Obviously, evaluating the number of solutions of a search space is at least as hard as the problem itself. By relaxing the problem, the authors consider the counting problem for each constraint separately, guiding the exploration toward areas that contain a high number of solutions in individual constraints. In other words, such a strategy bets on the future of the search space exploration.

This paper focuses on the global cardinality constraint (*gcc*) (?). Such a constraint restricts the number of times a value is assigned to a variable to be in a given integer interval. It has been known to be very useful in many real-life problems derived from the generalized assignment problems (?), such as scheduling, timetabling, or resource allocation XXXREF. In (?), the authors state an upper bound on the number of solutions for an instance of *gcc*, as well as exact evaluations for other constraints like regular or knapsack. This paper shows that this result is actually not correct and presents a correction that uses a dedicated non-linear minimization problem. We give a detailed algorithm and a complexity study for the computation of the new bound. The corrected bound is then adapted for counting-based search and its behavior is compared to the former result (?). This paper concludes with an experimental analysis of the efficiency of both estimators within the search heuristic *maxSD*, that aim at exploring first the area where there are likely more solutions.

Outline. Section 2 first introduces the required constraint programming (CP) background as well as the required material in graph and matching theory. Section 3 recalls the method proposed by (?) to compute an upper bound on the number of solutions for an instance of *gcc* before presenting a counter-example of it. Section 4.1 presents a direct calculation method based on a non-linear minimization problem and Section 4.2 gives a time complexity study of the computation of this corrected upper bound. Finally Section 5 presents an experimental evaluation of the new upper bound within Counting-Based Search strategies.

2. Background

This section first defines the global cardinality constraint and illustrates its connection to network flow theory. Then, some necessary background related to the graph and matching theory is introduced.

2.1 Global Cardinality Constraint

In the following, we will consider the classical constraint programming framework, where the variables are $X = \{x_1, \dots, x_n\}$, taking their values in finite domains $\{D_1, \dots, D_n\}$. We write D

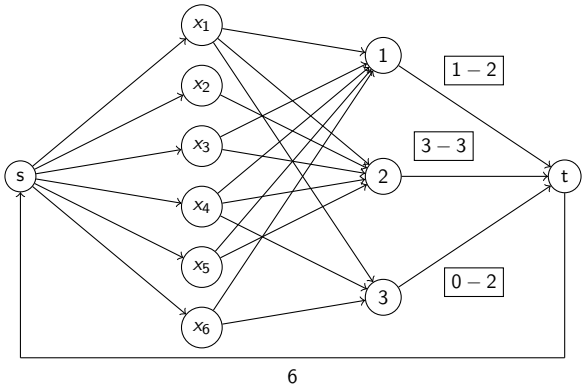


Figure 1: Flow model of the instance of gcc presented in Example 2.2

the cartesian product of the domains and D_X the union of all the domains: $D_X = \bigcup_{i \in \{1,\dots,n\}} D_i = \{y_1, \dots y_m\}$.

In this work, we focus on the global cardinality constraint, written *gcc*, which constrains the number of times a value is assigned to a variable to be in a given integer interval. Given a *gcc* constraint on X over domains $\{D_1, \dots, D_n\}$, we note $T(gcc)$, the set of n-uples that satisfy *gcc* (or its solutions) and $\#gcc = |T(gcc)|$, the number of solutions.

Definition 2.1 (Global Cardinality Constraint (gcc), (?))**.** *Let $l, u \in \mathbb{N}^m$ be two m-dimensions vectors. We define $gcc(X, l, u)$, the constraint satisfaction problem, which search for an assignment of each variable of X such that each value $y_j \in D_X$ must be taken at least l_j times and at most u_j times. More formally:*

$$T(gcc(X, l, u)) = \{(d_1, \dots, d_n) | d_i \in D_i, l_d \leq |\{d_i | d_i = d\}| \leq u_d, \forall d \in D_X\} \quad (1)$$

The state-of-the-art (?) shows that a global cardinality constraint can be represented as a flow-based model on a bipartite graph. Let us take an example:

Example 2.2. *Suppose we have a gcc defined on $X = \{x_1, \dots, x_6\}$ with domains $D_1 = D_4 = \{1, 2, 3\}$, $D_2 = \{2\}$, $D_3 = D_5 = \{1, 2\}$ and $D_6 = \{1, 3\}$; lower and upper bounds for the values are respectively $l_1 = 1, l_2 = 3, l_3 = 0$ and $u_1 = 2, u_2 = 3, u_3 = 2$. We can model this gcc as the flow problem depicted by Figure 1.*

The labels $l_j - u_j$ between each value node y_j and the sink t represent the lower bound and upper bound for the flow between y_j and t. In order to make Figure 1 easier to read, we did not represent the labels for edges between variables and values and between the source s and the variables. The flow between a variable and a value must be 0 or 1 and the flow between the source s and each variable is 1.

Then $(1, 2, 1, 2, 2, 3)$ and $(3, 2, 1, 2, 2, 3)$ are solutions but $(1, 2, 1, 1, 2, 3)$ is not because the value 2 is only taken twice and it must be taken at least three times.

2.2 Matching Theory

We will see in subsection 3.2 that the global cardinality constraint can also be modeled as a maximum matching problem. The next definitions recall some graph and matching theory

definitions that will be used afterwards. In the following, we will consider an undirected bipartite graph written $G = (V, E)$ for a graph, and $V = V_1 \cup V_2$ the two subsets of nodes corresponding to the partitions of G. The graph G is balanced iff $|V_1| = |V_2|$. Unless specified, we take the same framework and notations as (?) and we refer the reader to this book for more details.

Definition 2.3 (Biadjacency matrix of a bipartite graph)**.** *Let $G(V_1 \cup V_2, E)$ be an undirected bipartite graph. We call the biadjacency matrix of G, the matrix $B_G = (b_{ij})$ such that*

$$b_{ij} =: \begin{cases} 1, \text{if } (v_i, v_j) \in E \text{ with } v_i \in V_1 \text{ and } v_j \in V_2 \\ 0, \text{otherwise} \end{cases} \tag{2}$$

Definition 2.4 (Matching)**.** *Let $G(V, E)$ and $M \subseteq E$. M is a matching on G iff no two edges of M have a node in common.*

Definition 2.5 (Maximum matching)**.** *Let $G(V, E)$ and $M \subseteq E$. M is a maximum matching on G iff M is a matching on G and $|M|$ is maximum.*

Definition 2.6 (Perfect matching)**.** *Let $G(V_1 \cup V_2, E)$ and $M \subseteq E$. M is a perfect matching on G if and only iff M is a matching on G and $|M| = |V_1| = |V_2|$.*

Note that a perfect matching can only exist in a balanced graph. Also, a perfect matching is a maximum matching. We note $\#PM(G)$, the number of perfect matchings in G.

The next definition and proposition are the main results that (?) have shown to be relevant to count solutions on a global cardinality constraint. In the following, we write $\mathbb{M}_{n,n}$ the set of square matrices of size n and S_n, the symmetry group over $\{1, \dots, n\}$, that is the group of permutations over $\{1, \dots, n\}$.

Definition 2.7 (Permanent of a matrix)**.** *Let $A = (a_{ij}) \in \mathbb{M}_{n,n}$ be a square matrix, we define the permanent of A a follow:*

$$Perm(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} a_{i,\sigma(i)} \tag{3}$$

is the permanent of A.

The permanent of a matrix looks similar to the determinant of a matrix. However, it is much harder to compute. It is actually a $\#P-$complete problem, as explained in (?).

Proposition 2.8. *Let $G(V_1 \cup V_2, E)$ be a balanced bipartite graph and B_G be its biadjacency matrix, then the number of perfect matchings in G is equal to the permanent of B_G:*

$$Perm(B_G) = \#PM(G) \tag{4}$$

This result is given in (?) and we will not detail its proof. Actually, a perfect matching in G corresponds to a permutation $\sigma \in \mathcal{S}_n$ such that, $\forall i \in \{1, \dots, n\}$, $b_{i\sigma(i)} = 1$, where $B_G = (b_{ij})$ is the biadjacency matrix of G. This result is central in this study, as it gives a mathematical way to compute the number of perfect matchings in a balanced bipartite graph. Since the exact computation of $Perm(B_G)$ is $\#P$-complete, we will use two upper bounds that can be computed in polynomial time.

The first one is the Brégman-Minc upper bound conjectured in (?) and proven in (?).

Proposition 2.9 (Brégman-Minc upper bound). *Let $A \in \mathbb{M}_{n,n}$ and $\forall i \in \{1, \dots, n\}$, r_i the sum of the i^{th} row of A, then*

$$Perm(A) \leq UB^{BM}(A) = \prod_{i=1}^{n} (r_i!)^{\frac{1}{r_i}} \tag{5}$$

The second one is the Liang-Bai upper bound established in (?). An independent proof of this bound has been published in (?) .

Proposition 2.10 (Liang-Bai upper bound). *Let $A \in \mathbb{M}_{n,n}$ and $\forall i \in \{1, \dots, n\}$, r_i the sum of the i^{th} row of A and $q_i = min(\lceil \frac{r_i+1}{2} \rceil, \lceil \frac{i}{2} \rceil)$, then*

$$Perm(A) \leq UB^{LB}(A) = \prod_{i=1}^{n} \sqrt{q_i(r_i - q_i + 1)} \tag{6}$$

None of these bounds strictly dominates the other. In the following, $UB^{BM}(B_G)$ can also be noted as $UB^{BM}(G)$ (same for $UB^{LB}(B_G)$), where B_G is biadjacency matrix of G.

3. Counting the Solutions to a Global Cardinality Constraint

In this section, we present the method proposed in (?) to compute an upper bound on the number of solutions of a *gcc* instance. This method requires to compute the number of maximum matchings covering one partition of an unbalanced bipartite graph. Subsection 3.1 presents how (?) suggests to enumerate those maximum matchings. The method is then described in Subsection 3.2 before we develop a counter-example to prove it wrong in Subsection 3.3. Subsection 3.4 introduces a new model that will help us in fixing the error.

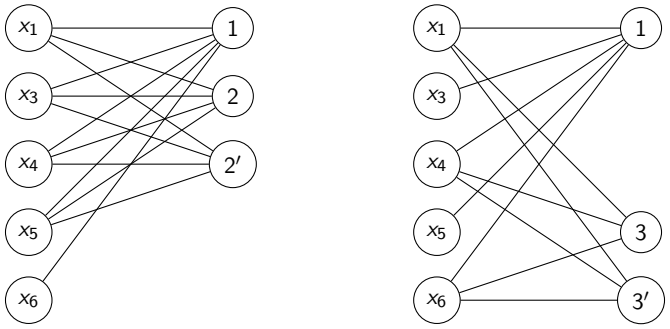
3.1 Number of maximum matchings of an unbalanced bipartite graph

We have seen how to count the number of perfect matchings on a balanced bipartite graph. For unbalanced bipartite graph, we are interested in counting the number of maximum matchings that cover every node of the smaller partition. In the following, if G is not balanced, then $\#PM(G)$ refers to the number of maximum matchings that cover the smaller partition.

We present here the way (?) use to deal with unbalanced graphs. Let us consider $G(V_1 \cup V_2, E)$, such that $|V_1| < |V_2|$. We are interested in computing the number of matching covering every node in V_1. In order to retrieve a balanced graph, first, n_f fake nodes are added to partition V_1.

Notation. *If G is an unbalanced bipartite graph, we note G^{bal} the corresponding balanced bipartite graph, after adding the fake nodes.*

After balancing the graph, the number of perfect matchings on G^{bal} can be computed, as seen in subsection 2.2. Since there are fake nodes, the computed permanent (or upper bound) is an overestimation of the true number of matching covering V_1 on G. (?) proposes to divide the permanent of the balanced graph by the number of permutations among the fake nodes. Indeed, for each matching covering V_1 on G, there are $n_f!$ corresponding perfect matchings on G^{bal}, each fake node being linked to every node in V_2. In the following, if G is not balanced, $\#PM(G)$ refers to the number of maximum matchings covering the smaller partition and we



(a) (b)

Figure 2: Lower Bound Graph (a) and Upper Bound Residual Graph (b) of the *gcc* instance described in Example 2.2

can write $\#PM(G) = \frac{\#PM(G^{bal})}{n_f!}$. Also, we will consider that $UB^{BM}(G)$ is an upper bound over the number of maximum matchings and $UB^{BM}(G) = \frac{UB^{BM}(G^{bal})}{n_f!}$ (same thing for $UB^{LB}(G)$).

A practical case of this method is developed in Example 3.3.

3.2 Upper bounding the number of solutions to a gcc

We present here the method proposed in (?) to compute an upper bound of the number of solutions of a *gcc* instance. The authors first count partial instantiations that satisfy the lower bound restriction. Then, for each of these partial instantiations, they count how many possibilities there are to complete it to a full instantiation satisfying the upper bound restriction.

(?) only considers instances in which every fixed variable (that can be instantiated to only one value) has been removed and the lower and upper bound have been adjusted accordingly. Let $X' = \{x_i \in X |\ x \text{ is not fixed}\}$ be the set of unfixed variables and lower bounds are l' where $l'_d = l_d - |\{x_i | x_i \text{ is fixed and } d \in D_i\}|$ and upper bounds u' are defined similarly. We assume that, $\forall d \in D_X$, $l'_d \geq 0$ and $u'_d \geq 0$.

The first stage of the method counts the partial instantiations satisfying the lower bound restriction. For this purpose, the notion of Lower Bound Graph is introduced:

Definition 3.1 (Lower Bound Graph, (?))**.** *Let $G_l(X' \cup D_l, E_l)$ be an undirected bipartite graph where X' is the set of unfixed variables and D_l, the extended value set, that is for each $d \in D_X$ the graph has l'_d vertices $d^1, d^2, \ldots$ representing d (l'_d possibly equal to zero). There is an edge $(x_i, d^k) \in E_l$ if and only if $d \in D_i$.*

Figure 2a represents the Lower Bound Graph for the instance described in Example 2.2 and its computation is detailed in Example 3.3.

By construction, a matching covering every vertex of D_l corresponds to a partial instantiation of the variables satisfying the lower bound restriction. The matching $\{(x_1, 2), (x_4, 2'), (x_5, 1)\}$ corresponds to the partial assignment $(2, 2, \_, 2, 1, \_)$ (x_2 is already instantiated) and this partial instantiation satisfies the lower bound restriction. Note that the matching $\{(x_1, 2'), (x_4, 2), (x_5, 1)\}$ leads to the same partial instantiation. Switching two duplicated values does not change the resulting partial instantiation. Actually, every combination of permutations among duplicates of

a same value must be considered. There are $\frac{\#PM(G_l)}{\prod_{d \in D_X} l'_d!}$ partial instantiations satisfying the lower bound restriction.

The authors are now interested in counting every possibility to complete a partial instantiation to a full instantiation. Similarly, the Upper Bound Residual Graph is introduced:

Definition 3.2 (Upper Bound Residual Graph, (?))**.** *Let $G_u(X' \cup D_u, E_u)$ be an undirected bipartite graph where X' is the set of unfixed variables and D_u the extended value set, that is for each $d \in D_X$, the graph has $u'_d - l'_d$ vertices $d^1, d^2, \dots$ representing d (if $u'_d - l'_d$ is equal to zero then there is no vertex representing d). There is an edge $(x_i, d^k) \in E_u$ if and only if $d \in D_i$ and $u'_d - l'_d > 0$.*

Figure 2b represents the Upper Bound Residual Upper Graph for the instance described in Example 2.2. At this point, $K = \sum_{d \in D_X} l'_d$ variables are already instantiated (without counting fixed variables). They are removed from the Upper Bound Residual Graph and, by construction, a matching covering the remaining variables corresponds to a completion of the partial assignment. On the partial instantiation $(2, 2, \_, 2, 1, \_)$, x_3 and x_6 remain to instantiate. The matching $\{(x_3, 1), (x_6, 3')\}$ leads to the full instantiation $(2, 2, 1, 2, 1, 3)$, which satisfies both the lower bound and upper bound restrictions.

However, in general, there is an exponential number of partial assignments satisfying the lower bound restriction. It is not reasonable to compute the number of maximum matchings on G_u for each of them. (?) suggests an over-approximation of the number of possibilities to complete each partial assignment: the K variables already instantiated are the variables that contribute the less to the combinatorial complexity of the problem. They remove the K variables such that the number of maximum matchings in G_u covering the remaining variables is maximized. The resulting graph is noted $\overline{G_u}$ (see Figure 3). Like in the Lower Bound Graph, there are symmetries among duplicated values in D_u. There are at most $\frac{\#PM(\overline{G_u})}{\prod_{d \in D_X} (u'_d - l'_d)!}$ ways to complete a partial instantiation satisfying the lower bound restriction to a full instantiation also satisfying the upper bound restriction.

(?) concludes on the following upper bound:

$$\#gcc(X, l, u) \leq \frac{\#PM(G_l) \cdot \#PM(\overline{G_u})}{\prod_{d \in D_X} l'_d! \cdot (u'_d - l'_d)!} \tag{7}$$

In practice, we do not compute directly the number of perfect matchings, but we use the Bregman-Minc or Liang-Bai upper bound (Subsection 2.2). Also we do not compute exactly $\overline{G_u}$, because it would require to consider $\binom{|X'|}{K}$ graphs and to compute the number of perfect matchings for each of them. Instead, we remove the K variables that contribute the less to the computation of the Bregman-Minc or Liang-Bai upper bound (*i.e.* the ones having the smaller factors in the product). Example 3.3 details this method:

Example 3.3. *We consider the same gcc as in Example 2.2: $X = \{x_1, \dots, x_6\}$ with domains $D_1 = D_4 = \{1, 2, 3\}, D_2 = \{2\}, D_3 = D_5 = \{1, 2\}$ and $D_6 = \{1, 3\}$; lower and upper bounds for the values are respectively $l_1 = 1, l_2 = 3, l_3 = 0$ and $u_1 = 2, u_2 = 3 u_3 = 2$.*

Considering that $x_2 = 2$, the lower and upper bounds for the value 2 are respectively $l'_2 = 2$ and $u'_2 = 2$. The Lower Bound Graph is shown in Figure 2a: variable x_2 is fixed and thus

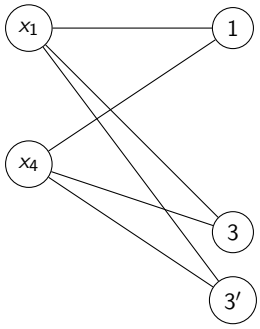
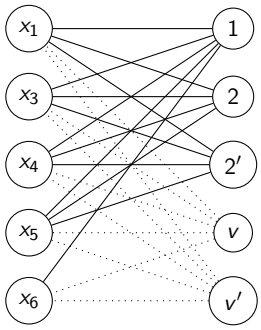


Figure 3: Upper Bound Residual Graph after removing x_3, x_5 and $x6$: $\overline{G_u}$



(a) Lower Bound Graph after adding fake vertices, G_l^{bal}

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 \end{pmatrix}$$

(b) Biadjacency matrix of G_l^{bal}

Figure 4: Lower Bound Graph after adding fake vertices and its biadjacency matrix

does not appear in the graph, value vertex 2 is represented by two vertices because $l'_2 = 2$ *; finally value vertex 3 does not appear because* $l'_3 = 0$*. We construct similarly the Upper Bound Residual Graph (Figure 2b).*

We already have instantiated 3 variables at this stage (4, if we count x_2*, which is fixed). To construct* $\overline{G_u}$*, we remove* x_3*,* x_5 *and* x_6 *from* G_u*, which contribute the less in the combinatorial complexity of the problem (we actually could have chosen* x_1 *or* x_4 *instead of* x_6*).* $\overline{G_u}$ *is such as in Figure 3.*

First, we compute $\#PM(G_l)$*. We add two fake vertices on the value partition,* v *and* v'*, in order to balance* G_l *(Figure 4a) and we compute the permanent of its biadjacency matrix (Figure 4b). The permanent of* $B_{G_l^{bal}}$ *is 72. Then we divide by* $2!$ *to deal with the combinatorial complexity induced by the fake vertices, there are then* $\#PM(G_l) = 36$ *matching covering the value partition in* G_l*.*

Similarly, we can compute $\#PM(\overline{G_u}) = 6$*. Then, we can conclude that:*

$$\#gcc(X, l, u) \leq \frac{36 \cdot 6}{2! \cdot 2!} = 54$$

The true number of solutions of this problem is 19. Scaling and also fake vertices used with the permanent bounds are factors that degrade the quality of the upper bound.

Instantiation	x_1	x_2	x_3
ι_1	1	2	1
ι_2	1	2	2
ι_3	1	3	1
ι_4	1	3	2
ι_5	2	2	1
ι_6	2	3	1
ι_7	3	2	1
ι_8	3	3	1

Figure 5: Array of every instantiation

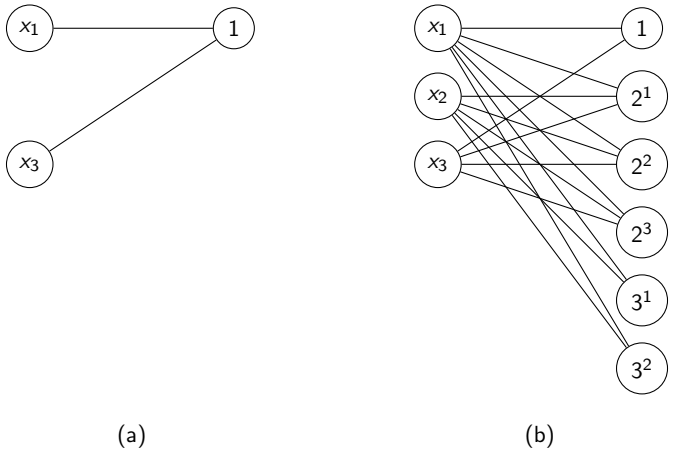


Figure 6: Lower Bound Graph G_l (a) and Upper Bound Residual Graph G_u (b)

3.3 Counter-example

The method described in the subsection 3.2 does not work in general. We develop here a counter-example that proves it wrong and we analyze the error.

Let $X = \{x_1, x_2, x_3\}$, $D_1 = \{1, 2, 3\}$, $D_2 = \{2, 3\}$, $D_3 = \{1, 2\}$ and $l = \{1, 0, 0\}$ and $u = \{2, 3, 2\}$. A list of every instantiation is given by Figure 5.

The Lower Bound Graph and Upper Bound Residual Graph are presented by Figure 6. Only value node "1" is in the Lower Bound Graph because $l_1 = 1$, $l_2 = 0$ and $l_3 = 0$. Also the variable x_2 cannot take the value 1, this is why it is not in the Lower Bound Graph. There is only one value in the Lower Bound Graph, which means that only one variable is instantiated at this stage, then we need to delete one variable from G_u. We choose to delete x_3, as this is the variable which have less impact on the combinatorial complexity. It ensures that we are computing an upper bound. After deleting one variable from G_u, we have $\overline{G_u}$ presented by Figure 7.

If we apply directly the upper bound of Equation (7) (see subsection 3.2), we have:

$$\#gcc(X, l, u) \leq \frac{\#PM(G_l) \cdot \#PM(\overline{G_u})}{\prod_{d \in D_X} l'_d! \cdot (u'_d - l'_d)!} = \frac{2 \cdot 25}{2! \cdot 3!} = \frac{25}{6} = 4,1666$$

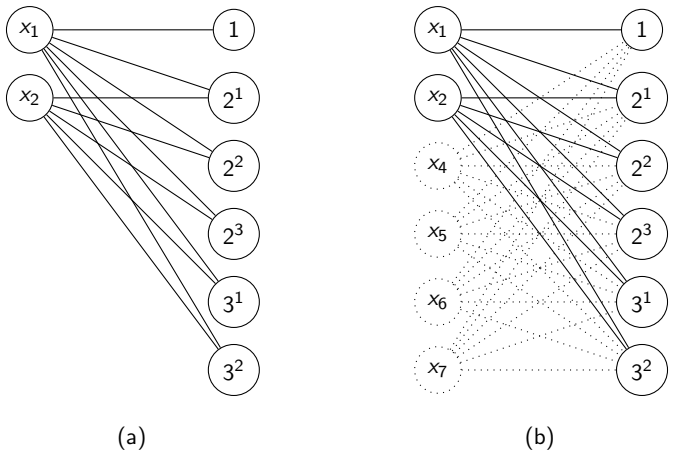


Figure 7: $\overline{G_u}$ (a) and $\overline{G_u}^{bal}$ (b)

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Figure 8: Biadjacency matrix of $\overline{G_u}^{bal}$ (in counter-example)

Perfect Matching	x_1	x_2	x_4	x_5	x_6	x_7
μ_1	1	3^1	2^1	2^2	2^3	3^2
μ_2	1	3^1	2^1	2^3	2^2	3^2
μ_3	1	3^1	2^2	2^1	2^3	3^2
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
μ_{24}	1	3^1	3^2	2^3	2^2	2^1
μ_{25}	1	3^2	2^1	2^2	2^3	3^1
μ_{26}	1	3^2	2^1	2^3	2^2	3^1
μ_{27}	1	3^2	2^2	2^1	2^3	3^1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
μ_{48}	1	3^2	3^1	2^3	2^2	2^1

Figure 9: Array of every perfect matchings on $\overline{G_u}^{bal}$

It is obvious that $\#PM(G_l) = 2$, as for $\#PM(G_u)$, we compute the permanent of the biadjacency matrix of $\overline{G_u}^{bal}$ (Figure 8) , $Perm(\mathcal{B}(\overline{G_u}^{bal})) = 600$, and, as we added 4 fake variables, we divide by 4!. Thus, we get $\#PM(\overline{G_u}) = \frac{600}{4!} = 25$.

But, we know that $\#gcc(X, l, u) = 8$, as we enumerated every instantiation in Figure 5. There is an error when computing the number of matchings covering x_1 and x_2 in $\overline{G_u}^{bal}$. Figure 9 lists every perfect matching of $\overline{G_u}$ corresponding to $\{x_1 = 1, x_2 = 3\}$.

We can notice, for example, that μ_1 and μ_2 are symmetric by transposition of x_5 and x_6 and also by transposition of 2^2 and 2^3. In the method proposed, we first deal with the fake variables symmetry and then with the duplicated values symmetry. The symmetry between μ_1 and μ_2 is thus counted twice instead of once. Fake variables symmetry and duplicated values symmetry may actually offset each other and should not be treated separately.

However, the problem does not appear for the Lower Bound Graph, as the number of variables is greater than or equal to the number of values (including the duplicates). In that case, the fake vertices symmetry comes from the same partition than the duplicated values symmetry. It is then impossible to have two perfect matchings being symmetric in both ways. In section 3.4, we modify the model to fit the case where the duplicated values symmetry and fake variables symmetry are conflictual.

3.4 A model that fits our specific problem

In the previous subsection, we saw that the method does not work when there are fewer variables than duplicated values. In this subsection, we formally present and purify the problem to fit this case. This model and notations will be retained in the next sections.

As before, let $X = \{x_1, \dots, x_n\}$ be the set of variables. For $i \in \{1, \dots, n\}$, D_i is the domain of x_i and we note $D_X = \bigcup_{i=1}^{n} D_i = \{y_1, \dots, y_m\}$. Let $\omega \in \mathbb{N}^m$ be the vector of occurrences, such that $\sum_{j=1}^{m} \omega_j \geq n$. In order to satisfy the global cardinality constraints, the variables of X must be instantiated in such a way that each value $y_j \in D_X$ is taken at most ω_j times. As in previous works (?) and in Section 3.2, in order to work with balanced bipartite graph, we add fake variables that can take any value from D_X. Let $\overline{X} = \{x_{n+1}, \dots x_{n+n'}\}$ be the set of fake variables, such that $n + n' = \sum_{j=1}^{m} \omega_j$ and $\forall x_i \in \overline{X}, D_i = D_X$. We introduce now

the ω-multiplied value graph, which is the value graph, in which each value node y_j has been duplicated ω_j times:

Definition 3.4 (ω-multiplied value graph)**.** *Let $G(X,\omega) = ((X \cup \overline{X}) \cup D_\omega, E_\omega)$ be the $\omega-$multiplied value graph with $E_\omega = \{(x_i, y_j^k) | y_j \in D_i\}$ and $D_\omega = \{y_1^1, \dots y_1^{\omega_1}, \dots, y_m^1, \dots y_m^{\omega_m}\}$.*

Figure 10a represents the ω-multiplied value graph for the instance described in Example 3.8. We will prove that, by construction, a perfect matching in the ω-multiplied value graph, corresponds to an instantiation over X satisfying the restriction on the occurrences. We first define formally the set of perfect matchings in $G(X,\omega)$ and the set of instantiations over X satisfying the restriction on the occurrences:

Definition 3.5 (Set of perfect matchings)**.** *We note $\mathcal{PM}(X,\omega)$, the set of perfect matchings in $G(X,\omega)$:*

$$\mathcal{PM}(X,\omega) = \{\{e_1, \dots, e_{n+n'}\} \subseteq E_\omega | \forall a, b \text{ if } a \neq b,\ e_a = (x_{i_a}, y_{j_a}^{k_a}) \text{ and } e_b = (x_{i_b}, y_{j_b}^{k_b}),$$
$$\text{then } x_{i_a} \neq x_{i_b} \text{ and } y_{j_a}^{k_a} \neq y_{j_b}^{k_b}\}$$

Definition 3.6 (Set of instantiations over X)**.** *We note $\mathcal{I}(X,\omega)$, the set of instantiations over X that satisfy the restriction on the occurrences ω_j:*

$$\mathcal{I}(X,\omega) = \{(d_1, \dots, d_n) | d_i \in D_i \text{ and } |\{d_i | d_i = y_j\}| \leq \omega_j\}$$

To simplify the notations, $\mathcal{PM}(X,\omega)$ and $\mathcal{I}(X,\omega)$ will be referred to as $\mathcal{PM}$ and $\mathcal{I}$, when there is no ambiguity on the considered instances. By construction of the ω-multiplied value graph, a perfect matching of $\mathcal{PM}$ corresponds to an instantiation of $\mathcal{I}$:

Proposition 3.7. *Let $\mu = \{(x_1, y_{j_1}^{k_1}), \dots, (x_{n+n'}, y_{j_{n+n'}}^{k_{n+n'}})\} \in \mathcal{PM}$, then $\iota = (y_{j_1}^{k_1}, \dots, y_{j_n}^{k_n})$ is an instantiation from $\mathcal{I}$.*

Proof. Let $\mu = (e_1, \dots, e_{n+n'}) \in \mathcal{PM}$. For a $i \in \{1, \dots, n+n'\}$, we write the edge e_i as $e_i = (x_i, y_{j_i}^{k_i})$. Let $\iota = (d_1, \dots, d_n)$, such that $\forall i \in \{1, \dots, n\}, d_i = y_{j_i}^{k_i}$.

We have $\forall i \in \{1, \dots, n\}, e_i = (x_i, y_{j_i}^{k_i}) \in \mathcal{PM} \subseteq E_\omega$, thus $d_i = y_{j_i}^{k_i} \in D_i$.

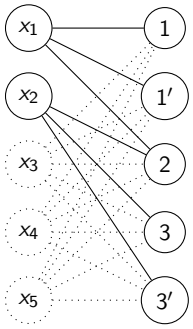
Also, for a $j \in \{1, \dots, m\}$, we have $|\{e_i \in \mu | y_{j_i}^{k_i} = y_j\}| = \omega_j$, since every value nodes is covered once in μ and there are ω_j duplicated nodes for each value. Then $\forall \mu' \subseteq \mu, |\{e_i \in \mu' | y_{j_i}^{k_i} = y_j\}| \leq \omega_j$ and, in particular, $|\{e_i \in \{e_1, \dots e_n\} | y_{j_i}^{k_i} = y_j\}| \leq \omega_j$. As $\forall i \in \{1, \dots, n\}, d_i = y_{j_i}^{k_i}$, we can conclude $\forall j \in \{1, \dots, m\}, |\{d_i | d_i = y_j\}| \leq \omega_j$. Then $\iota \in \mathcal{I}$ □

Notation. *Let $\mu \in \mathcal{PM}$, we note $\pi(\mu) = \iota$ the corresponding instantiation as introduced above.*

Several perfect matchings can lead to a same instantiation. Yes, the size of $\mathcal{I}$ and the size of $\mathcal{PM}$ are correlated. In next section, we present how to compute an upper bound of $|\mathcal{I}|$. Example 3.8 illustrates these new definitions and properties:

Example 3.8. *Let $X = \{x_1, x_2\}, D_1 = \{1, 2\}, D_2 = \{2, 3\}, \omega = \{2, 1, 2\}$ and $\overline{X} = \{x_3, x_4, x_5\}$. Then we obtain the $\omega-$multiplied value graph in Figure 10a. We also list every instantiation of $\mathcal{I}$ in Figure 10b.*

$\{(x_1, 1'), (x_2, 2), (x_3, 3'), (x_4, 1), (x_5, 3)\}$ is a perfect matching that leads to the instantiation $\iota_1 = (1, 2)$. ι_1 is also reached by $\{(x_1, 1), (x_2, 2), (x_3, 3), (x_4, 1'), (x_5, 3')\}$ and $\{(x_1, 1'), (x_2, 2), (x_3, 3), (x_4, 3'), (x_5, 1)\}$.



(a) $\omega-$multiplied Value Graph of the instance described in Example 3.8

instantiations	x_1	x_2
ι_1	1	2
ι_2	1	3
ι_3	2	3

(b) List of every elements of $\mathcal{I}$ for the Example 3.8

Figure 10: ω-multiplied value graph and list of every instantiation for Example 3.8

4. Upper-bound evaluation as a non-linear minimization problem

This section introduces a correction for the bound proposed in (?) and recalled in Section 3.2. Then we propose an algorithmic analysis of this correction

4.1 A New Upper Bound

Basically, the approach is to count how many perfect matchings in $G(X, \omega)$ lead to a specific instantiation, over the true variables, $\iota \in \mathcal{I}$. However, there are an exponential number of such instantiations, so we will find a lower bound of the number of such perfect matchings, by solving a non-linear minimization problem, in order to upper bound the number of instantiations in $\mathcal{I}$.

Proposition 4.1. *Let $\iota = (y_{j_1}, \dots, y_{j_n})$ be an instantiation of $(x_1, ..., x_n)$ and $\forall j \in \{1, \dots, m\}$, $c_j(\iota) = |\{i | y_{j_i} = y_j\}|$, which are the number of occurrences of each value in ι. There are $n'! \cdot \prod_{j=1}^{m} A^{\omega_j}_{c_j(\iota)}$ perfect matchings on $G(X, \omega)$ that lead to the instantiation ι, where $A^{\omega_j}_{c_j(\iota)}$ is the number of possible arrangements of $c_j(\iota)$ objects among ω_j.*

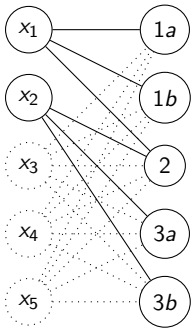
Proof. For each value $y_j \in \bigcup_{i=1}^{n} D_i$, there are $\binom{\omega_j}{c_j(\iota)}$ ways to pick up nodes in G to have as many occurrences of y_j as in ι and there are $c_j(\iota)!$ ways to order these nodes, then there are $\binom{\omega_j}{c_j(\iota)} \cdot c_j(\iota)! = A^{\omega_j}_{c_j(\iota)}$ ways to assign variables of X that are equal to y_j in the instantiation ι.

Each of these choices are independent, thus this result can be extended to every value y_j: there are $\prod_{j \in \{1,\dots,m\}} A^{\omega_j}_{c_j(\iota)}$ ways to assign the variables in X to get the instantiation ι.

We need now to consider fake variables assignment. For each assignment of X leading to the instantiation ι, there are $n'!$ ways to assign every variable of $\overline{X}$, then, there are $n'! \cdot \prod_{j \in \{1,\dots,m\}} A^{\omega_j}_{c_j(\iota)}$ perfect matchings in $G(X, \omega)$, that lead to the instantiation ι. □

The following example illustrates Proposition 4.1.

Example 4.2. *Let $X = \{x_1, x_2\}$, $D_1 = \{1, 2\}$, $D_2 = \{2, 3\}$, and $\omega = \{2, 1, 2\}$, then we must add 3 fake variables, $\overline{X} = \{x_3, x_4, x_5\}$. We obtain the $\omega-$multiplied value graph in Figure 10a. Let us now consider this instantiation of the variables x_1 and x_2 of this problem: $\{x_1 = 1, x_2 = 2\}$,*



(a) $\omega-$multiplied Value Graph

x_1	x_2	x_3	x_4	x_5
$1a$	2	$1b$	$3a$	$3b$
$1a$	2	$1b$	$3b$	$3a$
$1a$	2	$3a$	$1b$	$3b$
$1a$	2	$3a$	$3b$	$1b$
$1a$	2	$3b$	$1b$	$3a$
$1a$	2	$3b$	$3a$	$1b$
$1b$	2	$1a$	$3a$	$3b$
$1b$	2	$1a$	$3b$	$3a$
$1b$	2	$3a$	$1a$	$3b$
$1b$	2	$3a$	$3b$	$1a$
$1b$	2	$3b$	$1a$	$3a$
$1b$	2	$3b$	$3a$	$1a$

(b) List of perfect matchings corresponding to the solution $\{x_1 = 1, x_2 = 2\}$

Figure 11: ω-multiplied graph and list of perfect matchings for Example 4.2

and let us call it ι, we want to count how many perfect matching there are in $G(X, \omega)$ that lead to this instantiation. The array in Figure 11b lists those perfect matchings.

Note that we can remove every symmetric perfect matching by permutation of the variables of $\overline{X}$. There are $3! = 6$ such permutations. After deleting these perfect matchings, there remain 2 possibilities: either $\{x_1 = 1a, x_2 = 2\}$ or $\{x_1 = 1b, x_2 = 2\}$. In general, we need to count every possibility there is to instantiate variables in X. In ι, we chose one value "1" among two, one value "2" among one and zero value "3" among two. Also the order in which we pick these values is important, then there are $3! \cdot A_1^2 \cdot A_1^1 \cdot A_0^2 = 12$ perfect matchings leading to the instantiation ι.

We can directly deduce the following corollary, when extending Proposition 4.1 to every instantiation of $\mathcal{I}$:

Corollary 4.3.

$$\#PM(G(X, \omega)) = n'! \cdot \sum_{\iota \in \mathcal{I}} \prod_{j=1}^{m} A_{c_j(\iota)}^{\omega_j} \tag{8}$$

The following proposition gives an upper bound for $|\mathcal{I}|$:

Proposition 4.4.

$$|\mathcal{I}| \leq \frac{\#PM(G(X, \omega))}{n'! \cdot \min\limits_{\iota \in \mathcal{I}} (\prod_{j=1}^{m} A_{c_j(\iota)}^{\omega_j})} \tag{9}$$

Proof. The proof follows directly from Corollary 4.3. □

Remark. *What suggested the bound used in (?) is that*

$$|\mathcal{I}| \leq \frac{\#PM(G(X,\omega))}{n'! \cdot \prod_{j=1}^{m} \omega_j!}$$

We can check that we have, $\forall \iota \in \mathcal{I}$,

$$\prod_{j=1}^{m} A_{c_j(\iota)}^{\omega_j} \leq \prod_{j=1}^{m} \omega_j!$$

and then,

$$\frac{\#PM(G(X,\omega))}{n'! \cdot \min\limits_{\iota \in \mathcal{I}}(\prod_{j=1}^{m} A_{c_j(\iota)}^{\omega_j})} \geq \frac{\#PM(G(X,\omega))}{n'! \cdot \prod_{j=1}^{m} \omega_j!}$$

Our new objective is to compute $\min\limits_{\iota \in \mathcal{I}}(\prod_{j=1}^{m} A_{c_j(\iota)}^{\omega_j})$. In the following, $c_j(\iota)$ will be just noted as c_j. This problem can be modeled as this minimization problem:

$$IP =: \begin{cases} min \ \prod_{j=1}^{m} \frac{\omega_j!}{(\omega_j - c_j)!} \\ s.t. \sum_{j=1}^{m} c_j = n \\ \forall j \in \{1, ..., m\}, c_j \in \{0, ..., \omega_j\} \end{cases} \tag{10}$$

We can notice that for a high ω_j, choosing a high c_j, makes the objective function rise much more than choosing a high c_j, for a small ω_j. Also choosing a small c_j for a high ω_j makes the objective function rise much more than choosing the same c_j for a smaller ω_j.

Example 4.5. *Let* $\omega_1 = 5$ *and* $\omega_2 = 2$ *and* $n = 4$, *then* $c_1 \in [0, 5]$ *and* $c_2 \in [0, 2]$, *then taking* $c_1 = 4$ *and* $c_2 = 1$, *for example,* $\frac{5!}{(5-4)!} = 120 > 2 = \frac{2!}{(2-2)!}$.
And taking $c_1 = 1$ *and* $c_2 = 1$, *we have* $\frac{5!}{(5-1)!} = 5 > 2 = \frac{2!}{(2-1)!}$
As we must satisfy $c_1 + c_2 = n = 4$, *it seems better to, first, assign the biggest value possible for* c_2 *and then assign the rest to* c_1. *We would have* $c_1 = 2$ *and* $c_2 = 2$ *and* $A_2^2 * A_2^5 = 40$, *which is the minimum.*

This reasoning is generalized in the next proposition.

Proposition 4.6. *If* $\omega_1 \leq ... \leq \omega_m$, *then* $c^* = (\omega_1, \dots, \omega_{k-1}, c_k^*, 0, \dots, 0)$ *is a minimum of IP, with* $c_k^* = n - \sum_{j=1}^{k-1} \omega_j$.

Proof. Let $\omega = (\omega_1, ..., \omega_m)$ and let consider that $\omega_1 \leq \cdots \leq \omega_m$ and

$$f =: \begin{cases} \mathbb{N}^m \to \mathbb{N} \\ c \mapsto \prod_{j=1}^{m} \frac{\omega_j!}{(\omega_j - c_j)!} \end{cases} \tag{11}$$

We want to minimize $f(c)$ such that $\sum_{j=1}^{m} c_j = n$. Then, ω being in ascending order, we want to prove that f reaches a global minimum for:

$$c^* = (\omega_1, ..., \omega_{k-1}, c_k^*, 0, ..., 0) \tag{12}$$

with $c^*_k \in \{0, ..., \omega_k\}$.

Let $c = (c_1, ..., c_m)$ be a vector such that $\sum_{j=1}^{m} c_j = n$. We want to prove that $f(c^*) \leq f(c)$. Let us rewrite c as a modification of c^*. We can only decrease (not strictly) c^*_j for $j \in \{1, ..., k-1\}$ and increase (not strictly) c^*_j for $j \in \{k+1, ..., m\}$. As for c^*_k, there are two possibilities. We first deal with the case where c^*_k is increased. We rewrite c:

$$c = (\omega_1 - d_1, ..., \omega_{k-1} - d_{k-1}, c^*_k + u_k, u_{k+1}, ..., u_m) \tag{13}$$

with $\forall j \in \{1, ..., m\}, d_j \geq 0$ and $u_j \geq 0$ and $\sum_{j=1}^{m} u_j = \sum_{j=1}^{m} d_j$. We consider that $\forall j \in \{k, ..., m\}, d_j = 0$ and $\forall j \in \{1, ..., k-1\}, u_j = 0$. The last condition states that everything that has been removed, has also been added. Thus, we still have $\sum_{j=1}^{m} c_j = n$.

Now, we will see a method to transform vector c^* to c in several steps such that, at every step, the function f increases. Let c^j be the vector at step j.

For each $j \in \{1, k-1\}$, we remove d_j from c^j_j, and we redistribute on the variables of the end, that need to be increased, starting redistributing from index k to m. Let us take an example:

Let us consider a vector $\omega = (2, 3, 4, 4, 7)$ and $n = 7$. Then, let $c^* = (2, 3, 2, 0, 0)$ and $c = (0, 1, 3, 1, 2)$. Here are the steps following the method to get c from c^*:

- Step 0: $c^0 = c^* = (2, 3, 2, 0, 0)$
- Step 1: $c^1 = (0, 3, 3, 1, 0)$, we remove 2 from c^0_1 and we add 1 to c^0_3 and 1 to c^0_4.
- Step 2: $c^2 = c = (0, 1, 3, 1, 2)$, we remove 2 from c^1_2 and we add 2 to c^1_5.

In a general way, we have:

- Step 0: $c^0 = c^* = (\omega_1, ..., \omega_{k-1}, c^*_k, 0, ..., 0)$.
- ...
- Step j: $c^j = (\omega_1 - d_1, ..., \omega_j - d_j, \omega_{j+1}, ..., \omega_{k-1}, c^*_k + u_k, u_{k+1}, ..., u_{q-1}, r_q, 0, ..., 0)$.
- ...
- Step k-1 : $c^{k-1} = c = (\omega_1 - d_1, ..., \omega_{k-1} - d_{k-1}, c^*_k + u_k, u_{k+1}, ..., u_m)$.

with $r_q \in \{1, ..., u_q\}$ (or $\{x^*_k, ..., x^*_k + u_k\}$, if $q = k$), which is the residue for a particular index q. This index does not necessarily increase from one step j to the following one, as d_j may be smaller than $u_q - r_q$.

The objective, now, is to prove that $f(c^j) \leq f(c^{j+1}), \forall j \in \{1, ..., k-2\}$. We have:

$$c^j = (\omega_1 - d_1, ..., \omega_j - d_j, \omega_{j+1}, ..., \omega_{k-1}, c^*_k + u_k, u_{k+1}, ..., u_{q_a - 1}, r_{q_a}, 0, ..., 0)$$

and

$$c^{j+1} = (\omega_1 - d_1, ..., \omega_{j+1} - d_{j+1}, \omega_{j+2} ..., \omega_{k-1}, c^*_k + u_k, u_{k+1}, ..., u_{q_b - 1}, r_{q_b}, 0, ..., 0)$$

with $q_b \geq q_a$.

Now, we study $\frac{f(c^j)}{f(c^{j+1})}$.

First case: $q_a < q_b$

$$\begin{aligned}\frac{f(c^j)}{f(c^{j+1})} &= \frac{\omega_{j+1}!}{\frac{\omega_{j+1}!}{d_{j+1}!}} \cdot \frac{\frac{\omega_{q_a}!}{(\omega_{q_a}-r_{q_a})!}}{\frac{\omega_{q_a}!}{(\omega_{q_a}-u_{q_a})!}} \cdot \frac{1}{\frac{\omega_{q_b}!}{(\omega_{q_b}-r_{q_b})!}} \\ &= d_{j+1}! \cdot \frac{(\omega_{q_a}-u_{q_a})!}{(\omega_{q_a}-r_{q_a})!} \cdot \frac{(\omega_{q_b}-r_{q_b})!}{\omega_{q_b}!} \\ &= \frac{(d_{j+1}-r_{q_b})! \cdot (d_{j+1}-r_{q_b}+1) \cdot \ldots \cdot d_{j+1}}{(\omega_{q_a}-r_{q_a}) \cdot \ldots \cdot (\omega_{q_a}-u_{q_a}+1) \cdot \omega_{q_b} \cdot \ldots \cdot (\omega_{q_b}-r_{q_b}+1)}\end{aligned}$$

And, $(d_{j+1}-r_{q_b}+1) \cdot \ldots \cdot d_{j+1}$ and $\omega_{q_b} \cdot \ldots \cdot (\omega_{q_b}-r_{q_b}+1)$ are products of r_{q_b} consecutive terms. Moreover, $\omega_{q_b} \geq \omega_{j+1} \geq d_{j+1}$, then:

$$\frac{(d_{j+1}-r_{q_b}+1) \cdot \ldots \cdot d_{j+1}}{\omega_{q_b} \cdot \ldots \cdot (\omega_{q_b}-r_{q_b}+1)} \leq 1$$

We can notice that, in the case where redistributing d_{j+1} makes the index of the residue grow, $d_{j+1} = (u_{q_a} - r_{q_a}) + r_{q_b}$. In other words, what has been removed has been re-added. Then $(d_{j+1}-r_{q_b})!$ is the product of $u_{q_a}-r_{q_b}$ consecutive terms, the same as $(\omega_{q_a}-r_{q_a}) \cdot \ldots \cdot (\omega_{q_a}-u_{q_a}+1)$. We also know that $\omega_{q_a} \geq u_{q_a}$, then $d_{j+1} - r_{q_b} = u_{q_a} - r_{q_a} \leq \omega_{q_a} - r_{q_a}$, then:

$$\frac{(d_{j+1}-r_{q_b})!}{(\omega_{q_a}-r_{q_a}) \cdot \ldots \cdot (\omega_{q_a}-u_{q_a}+1)} \leq 1$$

Then,

$$\frac{f(c^j)}{f(c^{j+1})} \leq 1$$

Second case: $q_a = q_b$ We consider here the case where the index of the residue remains unchanged, but this does not mean that the residue is the same for c^j and c^{j+1}. Then we have $q_b = q_a$ and $r_{q_b} \geq r_{q_a}$. But we have $\omega_{q_a} = \omega_{q_b}$.

$$\begin{aligned}\frac{f(c^j)}{f(c^{j+1})} &= d_{j+1}! \cdot \frac{\frac{\omega_{q_a}!}{(\omega_{q_a}-r_{q_a})!}}{\frac{\omega_{q_b}!}{(\omega_{q_b}-r_{q_b})!}} \\ &= d_{j+1}! \cdot \frac{(\omega_{q_b}-r_{q_b})!}{(\omega_{q_a}-r_{q_a})!} \\ &= \frac{d_{j+1}!}{(\omega_{q_a}-r_{q_a}) \cdot \ldots \cdot (\omega_{q_a}-r_{q_b}+1)}\end{aligned}$$

What has been removed, has also been re-added: $d_{j+1} = r_{q_b} - r_{q_a}$ then $d_{j+1}!$ is a product of $r_{q_b} - r_{q_a}$ consecutive terms, like the product $(\omega_{q_a} - r_{q_a}) \cdot \ldots \cdot (\omega_{q_a} - r_{q_b} + 1)$. Moreover, $d_{j+1} = r_{q_b} - r_{q_a} \leq \omega_{q_a} - r_{q_a}$. Thus we have,

$$\frac{f(c^j)}{f(c^{j+1})} \leq 1$$

We have proved that, $\forall j \in \{1, \ldots, k-2\}$, $f(c^j) \leq f(c^{j+1})$. Then,

$$f(c^*) = f(c^0) \leq f(c^1) \leq \ldots \leq f(c^{k-1}) = f(c)$$

There remains one case to deal with. We have proved that $f(c^*) \leq f(c)$, for every c such that $c_k \geq c_k^*$. In the case where $c_k \leq c_k^*$, we need to adapt the method described above. In the previous case, we started removing d_j for j going from 1 to $k-1$ in that order. In this case, we first remove d_k and only, then we remove d_j for j going from 1 to $k-1$, as before. We will not detail the calculations for this case as they are very similar.

It follows that $f(c^*) \leq f(c)$ for every c such that $\sum_{j=1}^{m} c_j = n$ and $\forall j \in \{1, ..., m\}, 0 \leq c_j \leq \omega_j$. Then c^* is a minimum of our problem. □

From this proposition, we can deduce a method to get an upper bound of $\mathcal{I}$. We first sort the occurrences vector ω, then we can compute the vector c^* and compute $\min_{\iota \in \mathcal{I}}(\prod_{j=1}^{m} A_{c_j(\iota)}^{\omega_j}) = \prod_{j=1}^{m} A_{c_j^*}^{\omega_j}$. Then we get an upper bound of $|\mathcal{I}|$.

Proposition 4.7. *If $\omega_1 \leq ... \leq \omega_m$, then*

$$|\mathcal{I}| \leq UB^{IP}(X, \omega) = \frac{\#PM(G(X, \omega))}{n'! \cdot \prod_{j=1}^{m} A_{c_j^*}^{\omega_j}} \tag{14}$$

We have now corrected the method presented in subsection 3.2. We can use this new result to upper bound correctly the Lower Bound Graphs and Upper Bound Residual Graphs. This new upper bound requires sorting ω and in practice, we will use the Bregman-Minc or Liang-Bai upper bound instead of computing $\#PM(G(X, \omega))$. We also consider that we can compute the factorial function in constant time. This bound is then polynomially computable. A time complexity study is given in subsection 4.2.

4.2 Computation and Complexity

In this subsection, we present Algorithm 1, which details how to proceed to compute UB^{IP}. Then, a time complexity study is given.

At Line 1, we compute n', the number of fake variables. At Line 2, we sort the ω vector in ascending order. We have a function *zeros*(m) that builds a vector of size m filled with 0 in constant time. From Line 4 to Line 15, we simply fill c^*, such as it is defined. At Line 16, we compute an Bregman-Minc upper bound thanks to a function, that can compute it in $\mathcal{O}(n+n')$ operations. Finally, at Line 17, we return $UB_2(X, \omega)$. We consider that we can compute the number of arrangements and the factorial function in a constant time. We can conclude on the time complexity for the computation of UB_2:

Proposition 4.8. *The computation of UB^{IP} has a time complexity in*

$$\mathcal{O}(n + n' + m \cdot \log(m))$$

Proof. Computing n' and $\prod_{j=1}^{m} A_{c_j^*}^{\omega_j}$ requires $\mathcal{O}(m)$ operations. Also filling c^* requires $\mathcal{O}(m)$ operations (the worst case being when we have to fill entirely c^*). Sorting the ω vector requires $\mathcal{O}(m \cdot \log(m))$. Computing *nbPM* requires to compute a product over $n + n'$ factors, one for each variable node in $G(X, \omega)$, then it requires $\mathcal{O}(n + n')$ operations. Thus, we have a time complexity in $\mathcal{O}(n + n' + m \cdot \log(m))$ □

Computing UB^{IP} is linear in the number of variables and semi-linear in the number of values. In Section 5, we test the efficiency of the corrected bound within Counting-Based search strategy: maxSD (?).

Algorithm 1 Compute $UB^{IP}(X, \omega)$

1: $n' \leftarrow \sum_{j=1}^{m} \omega_j - n$
2: $\omega_{sorted} \leftarrow \omega.sortAsc()$
3: $c^* \leftarrow zeros(m)$
4: $count \leftarrow n + n'$
5: $idx \leftarrow 0$
6: **while** $count > 0$ **do**
7: **if** $count > \omega_{sorted}(idx)$ **then**
8: $count \leftarrow count - \omega_{sorted}(idx)$
9: $c*(idx) \leftarrow \omega_{sorted}(idx)$
10: **else**
11: $count \leftarrow 0$
12: $c^*(idx) \leftarrow \omega_{sorted}(idx) - count$
13: **end if**
14: $idx \leftarrow idx + 1$
15: **end while**
16: $nbPM \leftarrow UB^{BM}(G(X, \omega))$
17: **return** $nbPM/n'! \cdot \prod_{j=1}^{m} A_{c_j^*}^{\omega_j}$

5. Experimental analysis : Efficiency of the new upper bound within Counting-Based Search

The bound proposed in (?) is wrong. Though, the published result is correlated to the real number of solutions on a *gcc* instance. For this reason, the *maxSD* heuristic may make the same choices using the corrected upper bound or the previous wrong one. In this section, we first compare the instantiations order given by these two estimators and, then, we compare their efficiency within the *maxSD* heuristic.

5.1 Impact of the new upper bound on the instantiations order

In this subsection, we compare the instantiations order given by the former bound (the PQZ bound) and its correction. In Example 5.1, we compute, for a given instance of a *gcc*, the density of solutions for each possible instantiation, for the PQZ bound, its correction and the exact computation. We show that changing the way to estimate the number of solutions has a big influence on the instantiations order.

Example 5.1. *For this example, we randomly pick an instance of gcc, whose Value Graph is represented in Figure 12, with* $n = 10$ *variables,* $m = 10$ *values and an edge density* $p = 0,3$*, which means that each edge between a variable and a value has a probability* $p = 0,3$ *to exist. For each value, we also pick uniformly randomly a possible interval* $l_j - u_j$*, considering the number of neighbors of each value node in the Value Graph. These intervals are represented on the right of each corresponding value node.*

For each non-instantiated variable x_i*, and each value* $y_j \in D_i$*, we propagate the instantiation* $x_i \rightarrow y_j$ *and we compute an estimation of the number of remaining solutions, with the previous*

bound (PQZ bound) and the corrected one (we also compute the exact number of remaining solutions). From these estimations, we can compute the solution density for each instantiation.

The heuristic maxSD chooses first the instantiation associated with the highest solution density. With any of the estimators, the maxSD heuristic would choose first the instantiation $x_6 \to 0$. *We sort the instantiations by their solution density in descending order for the three estimators. If two instantiations have the same solution density, then we sort it with the lexicographic order.*

- *PQZ bound ((?)) :* $(x_6 \to 0, x_8 \to 0, x_4 \to 0, x_{10} \to 2, x_9 \to 0, x_3 \to 8, x_2 \to 8, x_7 \to 9, x_2 \to 1, x_3 \to 7, x_9 \to 2, x_7 \to 3, x_7 \to 5, x_{10} \to 6, x_{10} \to 8, x_{10} \to 3, x_8 \to 9, x_8 \to 8, x_4 \to 4, x_4 \to 9, x_4 \to 6, x_4 \to 5, x_6 \to 8, x_4 \to 8, x_7 \to 7)$
- *Corrected bound :* $(x_6 \to 0, x_8 \to 0, x_{10} \to 2, x_4 \to 0, x_9 \to 0, x_3 \to 8, x_2 \to 8, x_7 \to 3, x_7 \to 5, x_2 \to 1, x_3 \to 7, x_7 \to 9, x_7 \to 7, x_9 \to 2, x_4 \to 8, x_{10} \to 8, x_4 \to 4, x_4 \to 5, x_4 \to 6, x_4 \to 9, x_{10} \to 6, x_8 \to 8, x_6 \to 8, x_{10} \to 3, x_8 \to 9)$
- *Exact computation :* $(x_6 \to 0, x_8 \to 0, x_{10} \to 2, x_9 \to 0, x_3 \to 8, x_4 \to 0, x_2 \to 8, x_2 \to 1, x_3 \to 7, x_7 \to 9, x_7 \to 5, x_7 \to 3, x_9 \to 2, x_7 \to 7, x_8 \to 8, x_6 \to 8, x_4 \to 8, x_{10} \to 8, x_4 \to 6, x_8 \to 9, x_4 \to 5, x_4 \to 4, x_{10} \to 6, x_4 \to 9, x_{10} \to 3)$

We can see that the instantiation order is different depending on the estimator. The two first instantiations are identical. The length of the common prefix of these three instantiations orders is 2. In the following, we will formalize this measure of the (dis)similarity of instantiation orders, as a way to distinguish heuristics.

In order to measure how different is the behavior of these estimators within *maxSD*, we will use two measures of the similarity between two instantiation orders:

- $m_{LP}(I_1, I_2)$: the length of the common prefix between the instantiation order I_1 and I_2. The first instantiations are more important. We want to measure how many choices will be common in a row at the beginning. This measure is then normalized, dividing it by n.
- $m_{WS}(I_1, I_2)$: the weighted sum of common elements between the instantiations order I_1 and I_2. It measures the number of common elements and favor the common elements that appear in the beginning . The weight of the k^{th} elements among p instantiations is $p - k + 1$.

  Let $I_1 = (\iota_1^1, \dots, \iota_p^1)$, $I_2 = (\iota_1^2, \dots, \iota_p^2)$ and χ, the function such that $\chi(u, v) = 1$ if $u = v$ and $\chi(u, v) = 0$ otherwise. Then:

$$m_{WS}(I_1, I_2) = \frac{\sum_{k=1}^{p}(p - k + 1) \cdot \chi(\iota_k^1, \iota_k^2)}{\frac{1}{2} \cdot p \cdot (p + 1)}$$

  The denominator is here to normalize the measure, so we can make the comparison between the two estimators on different instances of *gcc*.

These two coefficients are normalized, such that if it is equal to 1, then the two instantiation orders are equal and, if it is equal to 0, they are very different. In Example 5.2, we show how to compute these similarity coefficients.

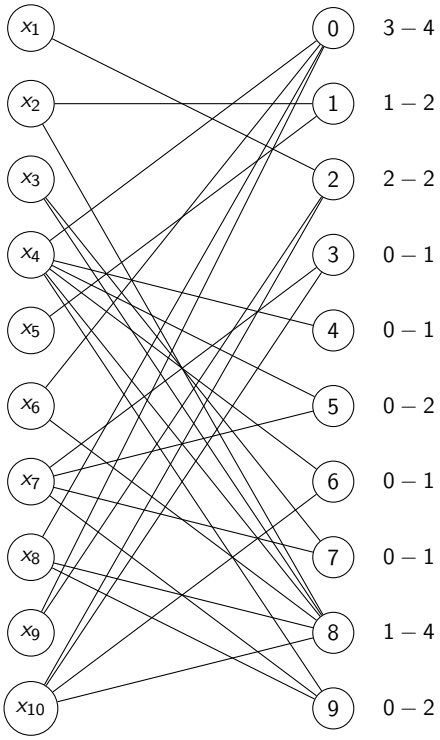


Figure 12: A random instance of *gcc*

Example 5.2. *Taking the three instantiations orders I_{PQZ}, I_{Cor} ans I_{exact} the instantiations orders given by the PQZ bound, the corrected bound and the exact computation in Example 5.1.* [Gilles: On s'intresse davantage aux mesures entre chacun et l'exact; est-ce qu'il est utile de mesurer entre les deux bornes?]

- *The length of the common prefix between the order given by the PQZ bound and its correction is 2, then $m_{LP}(I_{PQZ}, I_{Cor}) = \frac{2}{25} = 8\%$. Also, we have $m_{LP}(I_{PQZ}, I_{exact}) = \frac{3}{25} = 12\%$, as only the first instantiation is identical and $m_{LP}(I_{PQZ}, I_{exact}) = 8\%$.*
- *The instantiations $x_6 \to 0$, $x_8 \to 0$, $x_9 \to 0$, $x_3 \to 8$, $x_2 \to 8$, $x_4 \to 9$, $x_6 \to 8$ are in the same position in the ordering given by the PQZ bound and its correction. Their respective weight are : 26, 25, 22, 21, 20, 6 and 3. Then the weighted sum is 123 and we have $m_{WS}(I_{Cor}, I_{exact}) = \frac{123}{\sum_{k=1}^{25} k} = 37.8\%$. We also have $m_{WS}(I_{PQZ}, I_{exact}) = 25.2\%$ and $m_{WS}(I_{Cor}, I_{exact}) = 30.2\%$.*

Now that we have seen how to compare instantiations orders, we generate randomly 1000 instances of feasible *gcc* and we will compare the orders given by the PQZ estimator, its correction and the exact computation. To generate those instances, we will apply the same method we used for the generation of the instance presented in Example 5.1 with the same parameters, $n = 10, m = 10$ and $p = 0,3$.

Figure 13a shows the number of instances in function of the percentage of similarity between the resulting instantiations orders given by the PQZ bound and its correction, according to the

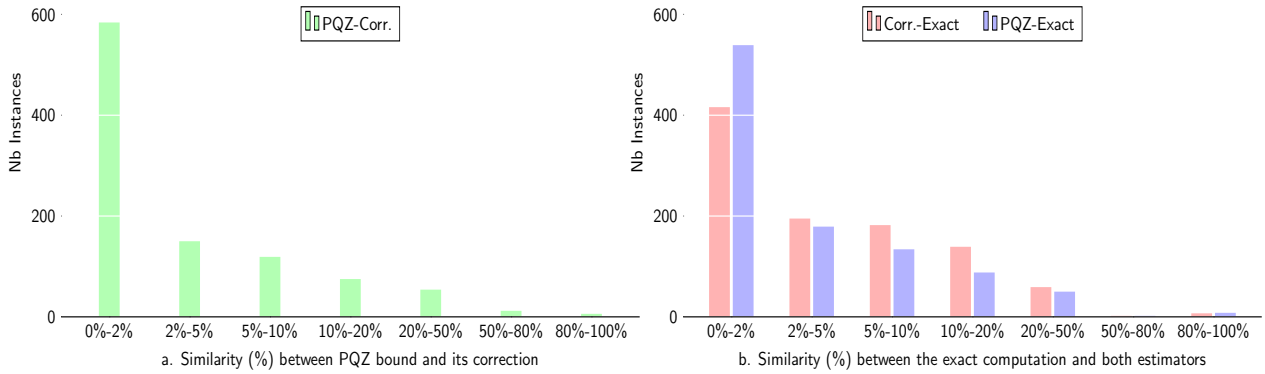


Figure 13: Proportion of instances per percentage of similarity according to m_{LP}

length of common prefix measure. We can notice that almost 600 among the 1000 generated instances, lead to less than 2% similarity. More than 90% of the instances are less than 20% similar. The *PQZ* bound and its correction very often lead to different instantiations orders. Figure 13b shows the number of instances in function of the percentage of similarity between the resulting instantiations orders given by the exact computation and the two other estimators, according to the length of common prefix measure. The corrected bound is similar at less than 2% to the exact bound for 58.9% of all instances, and the PQZ bound for only 51.6%. In general, the corrected bound gives more accurate instantiations orders.

In Figure 14a, we represent the number of instances in function of the percentage of similarity according to the weighted sum of common elements measure. With the m_{WS} measure, only about 100 instances have more than 50% similarity between the PQZ bound and its correction. The instantiation orders remain quite different.

In Figure 14b is represented the similarity between the exact computation and the two estimators according to the weighted sum of common elements measure. Though the m_{WS} is a much more flexible measure, there are still more than 100 instances that remain below 50% of similarity for both estimators. The correction seems to give closer instantiation orders than the PQZ estimator, as there are about 100 instances more that are less than 2% similar for the PQZ estimator and about 150 instances more that are between 20% and 50% similar for its correction.

Also, *maxSD* is a strategy that choose the assignment with the highest density. We notice that, for 42.5% of these 1000 instances, the PQZ estimator and its correction would choose the same first decision. For 59.3% of the instances, the exact computation and the corrected estimator would choose the same first decision against 47.0% for the exact computation and the PQZ estimator. And for 32.6% of the instances, the three estimators would choose the same first decision.

In this subsection, we have shown how to compare the influence of different number of solutions estimators within *maxSD* on the instantiations ordering, and we have shown that the PQZ bound and its correction behave in a different way.

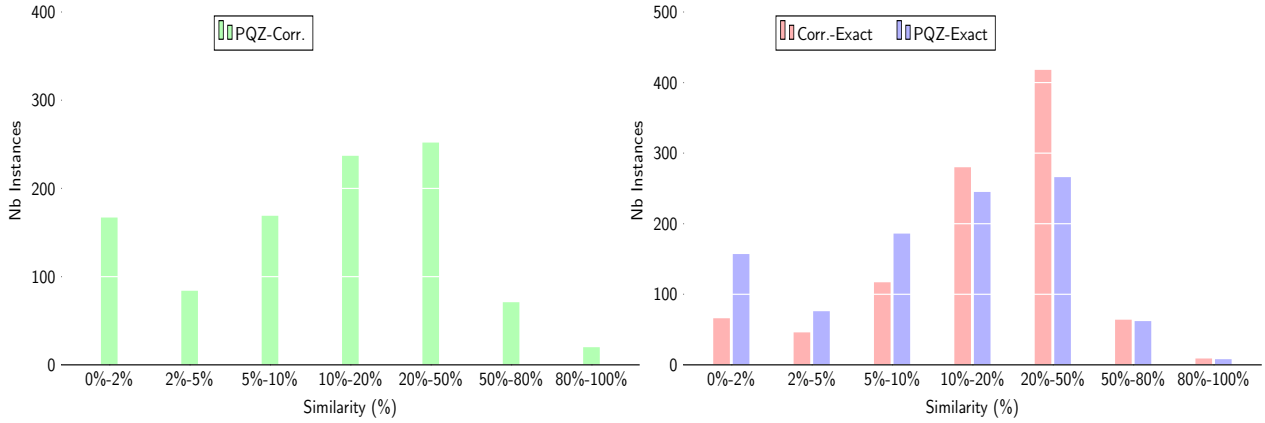


Figure 14: Proportion of instances per percentage of similarity according to m_{WS}

5.2 Performance analysis on generated instances

In this subsection we analyze the performance of each estimator within the *maxSD* strategy. We generate random instances of varying degree of difficulty, on which we run the two variants of the *maxSD* strategy. We first explain the generation process of such random instances and then we analyze the results obtained.

5.2.1 Generation of random CSPs

Let n be the number of variables, m the number of values and p, the edge density of the Value Graph. Each value $y_j \in Y$ have a probability p to be in D_i, for each variable x_i. Then, we add n_C global cardinality constraints to this CSP. The scope of each *gcc* is chosen randomly : each variable has a 50% chance to be picked. We are also given a tightness τ, which defines the length of every occurrence interval of each value. Let $\{x_{i_1}, \dots, x_{i_q}\}$ be the scope of one *gcc*, then the length of every occurrence interval of this *gcc* is $\tau \cdot q$, rounded to the nearest integer. Once the lenght is set, the interval is chosen randomly among every possible interval with such a length. The generated *gcc* are not trivially unsatisfiable, as we also ensure that the generated occurrence intervals, $[l_j, u_j]$, are such that:

$$\sum_{j=1}^{m} l_j \leq q \leq \sum_{j=1}^{m} u_j$$

We also ensure that any scope of a *gcc* is not included in another *gcc*. Indeed, two such *gcc* can be considered as one *gcc*, in which each occurrence interval is the intersection of two occurrence interval. We have thus exactly n_C disjoint global cardinality constraints in the generated *CSP*. This is an important point as we want the number of *gcc* and the tightness to be the two parameters that will directly affect the difficulty of the random instance.

5.2.2 Results Analysis

We generated several random CSP, with the method described above with the following parameters : $n = 20, m = 20$, an edge density $p \in \{0.33, 0.66, 1.0\}$, $\tau \in [0.1, 1]$ by step of 0.1 and $n_C \in [2, 10]$ by step of 1. For each couple (τ, n_C), we generate 50 random instances for a total of 4500 instances. We solve each instance with two different strategies: *maxSD* with the PQZ bound and *maxSD* with the corrected bound. The instances and the strategies are implemented in Choco solver (?) and we set, for each resolution, the time limit to 1 min and run on a 2.2GHz Intel Core i7 with 2.048GB.

The *maxSD* heuristic proposes to compute an estimator of the solution density, for each possible remaining instantiation, each time a fixed point is reached. This is a very costly strategy. Here, we will only compute and store the solution densities in the beginning of the search and each time we reach a fixed point, we will refer to them and choose the instantiation that led to the highest solution density at the beginning of the search and that is still available. We also thought about updating the solution densities each time the size of the domains have decreased enough (have reached a certain threshold), like proposed in (REF). This strategy is not effective on our generated problem, so we have preferred not to run it to gain more time.

$p \backslash \tau$	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
$p = 1.0$	**436**/435	**394**/393	**387**/381	**393**/365	**386**/364	**365**/353	**374**/369	**380**/364	**377**/368	**378**/356
$p = 0.66$	450/450	446/**447**	442/442	**440**/439	436/**438**	427/**434**	433/**434**	416/**420**	364/**400**	450/450
$p = 0.33$	450/450	450/450	450/450	450/450	450/450	450/450	450/450	447/**450**	450/450	450/450

Table 1: Number of solved instances for both estimators (PQZ/Corr.) in function of τ for $p \in \{0.33, 0.66, 1.0\}$

$p \backslash n_C$	2	3	4	5	6	7	8	9	10
$p = 1.0$	492/**493**	481/**482**	**455**/446	**433**/408	**424**/414	**389**/369	**383**/362	**400**/380	**413**/394
$p = 0.66$	500/500	495/495	**491**/489	487/**489**	475/**477**	468/**474**	460/**472**	466/**479**	462/**479**
$p = 0.33$	500/500	500/500	500/500	499/**500**	499/**500**	500/500	500/500	500/500	499/**500**

Table 2: Number of solved instances for both estimators (PQZ/Corr.) in function of n_C for $p \in \{0.33, 0.66, 1.0\}$

Table 1 (resp. Table 2) shows the number of solved instance for both estimators for $\tau \in \{0.1, \dots, 1.0\}$ (resp. $n_C \in \{2 \dots, 10\}$) and $p \in \{0.33, 0.66, 1.0\}$. The hardest instances seems to be for $n_C = 8$ and $\tau = 0.9$. For $p = 1.0$, the PQZ estimator solved more instances: 3870, against 3748 for its correction. For $p = 0.66$, the corrected bound get better results: 4354, against 4304 for PQZ. As for the edge density $p = 0.33$, the corrected bound solved every instance, while the PQZ estimator missed 3 of them. When the edge density is very low ($p \leq 0.33$), the domains are sparser, so there are much less instantiations to test during the computation of the estimators. This is why almost all these instances are solved in less than 1min. One estimator does not perform better than the other in general. To conclude on these tables, when the Value Graph is complete, PQZ estimator gives slightly better results, and when the domains are not

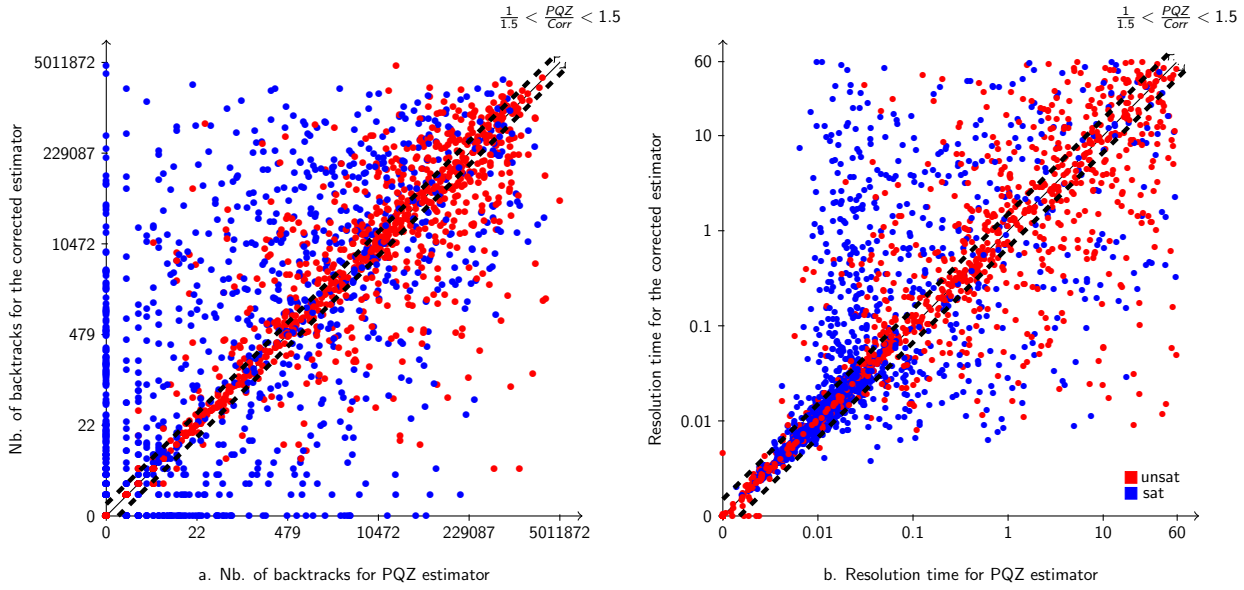


Figure 15: Comparison of the required number of backtracks and resolution time with p=1.0

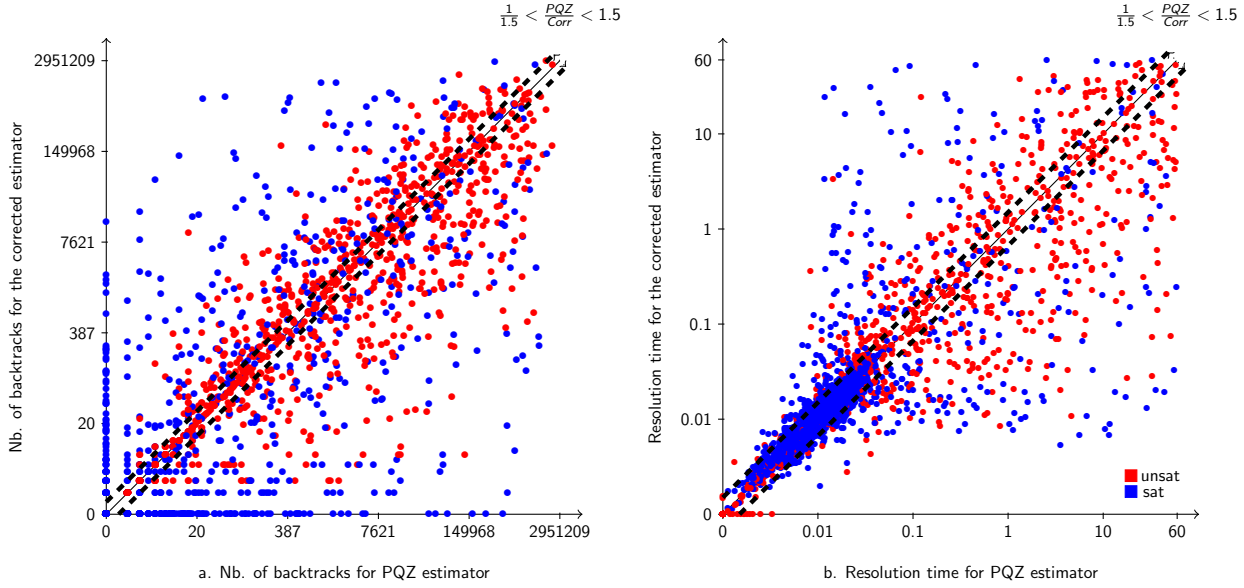


Figure 16: Comparison of the required number of backtracks and resolution time with p=0.66

uniform, the corrected bound solves more instances. It seems that the correction is able to catch more the difference among the domains than its previous version.

We now focus only on the instances that have been solved by both estimators. In Figure 15, Figure 16 and 17, we compare the required number of backtracks and the resolution time for each instance that have been solved (proved satisfiable or unsatisfiable) with both estimators for the edge density $p = 1.0$, $p = 0.66$ and $p = 0.33$. Blue dots represent the satisfiable instances and red ones represent the unsatisfiable instances. The area delimited by the black dotted

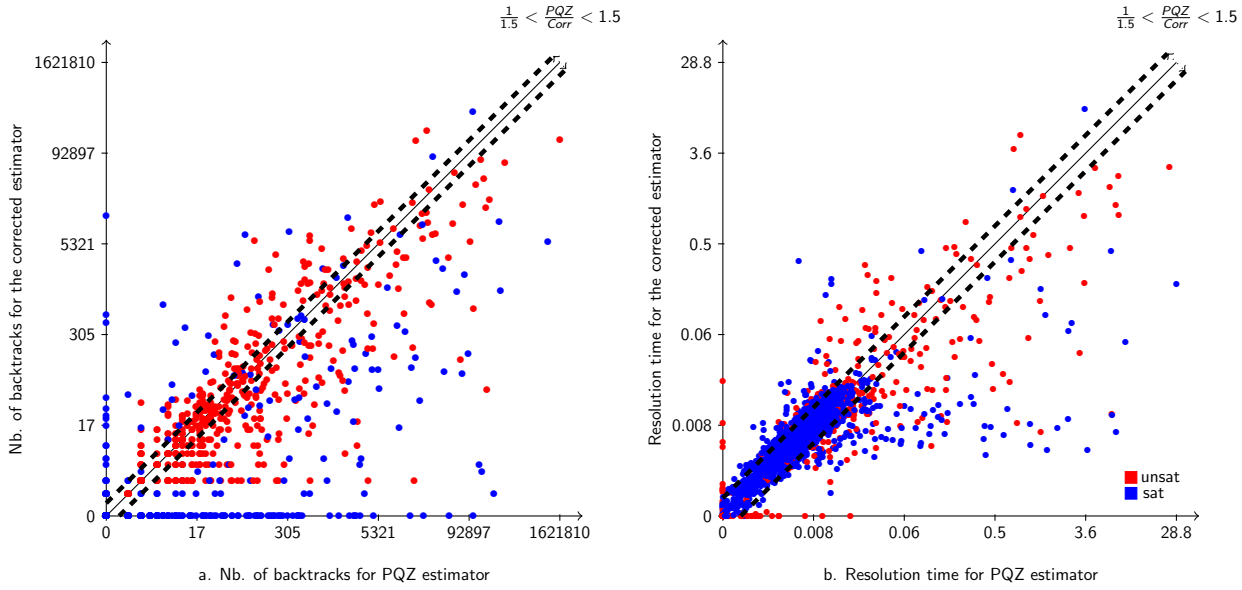


Figure 17: Comparison of the required number of backtracks and resolution time with p=0.33

lines represent the instances for which the ratio between the required number of backtracks (or resolution time) for PQZ and its correction is between $\frac{1}{1.5}$ and 1.5. In other words, we consider that, for the instances inside this area, both estimators have very comparable performances. If a point is below this area, it means that the corrected estimator has better performance, and on the opposite, if a point is above the area, the PQZ estimator has better performance.

For $p = 1.0$, 80.2% of the instances have been solved by both estimator. On Figure 15a, 60.1% of solved instances by both estimators are inside the dotted area. The performance, in terms of number of backtracks, of both estimators on theses instances are very similar. There are 16.1% points below the area and 23.8% points above. The PQZ estimator seems slightly better on these instances. More precisely, 67.6% of unsatisfiable instances are inside the dotted area, 15.4% below and 17.1% above. As for satisfied instances, there are 54.1% blue points inside the dotted area, 16.8% below and 29.2% above. For unsatisfiable instances, the two estimators have equivalent performances, but PQZ estimator seems to have better performance on these satisfiable instances. If we observe Figure 15b, 70.6% of red points and 70.4% of blue points are inside the dotted area. Also, 10.1% of blue points are below the area and 19.3% are above and 14.2% of red points are below and 15.4% are above. In terms of resolution time, both estimators have almost equivalent performance, but again, PQZ estimator is slightly better on satisfiable instances.

For $p = 0.66$, 94.9% of the instances have been solved by both estimator. On Figure 16a, 71.2% of solved instances by both estimators are inside the dotted area. The performance, in terms of number of backtracks, of both estimators on theses instances are again very similar, even more for the edge density. There 16.1% points below the area and 12.7% points above. Here, the corrected estimator seems to have slightly better performance. There is no clear difference here in the behaviour of unsatisfiable and satisfiable instances: 17.0% of blue points and 15.2% of red points below the dotted area and 13.3% of blue points and 12.0% of red points

above. On Figure 16b, 77.7% of red points and 78.1% of blue points are inside the dotted area. Also, 10.1% of the instances are above the area and 11.8% are below. In terms of resolution time, both estimators have very similar performance.

For $p = 0.33$, 99.9% of the instances have been solved by both estimator. On Figure 17a, 86.7% of solved instances by both estimators are inside the dotted area. For most instances, the performances of both estimators are more similar as the density get higher. There are 9.7% of the instances below the area and 3.5% above. The correction performs a little better on this density too. On Figure 17b, we observe that there are a little more points in the bottom part of the plot. It seems like the lower the edge density, the better are the performance of the correction over PQZ estimator.

Our conclusions on this benchmark are the following. As a preliminary remark, both heuristics are meant to tune the search toward areas with many solutions (or a high solution density for the *gcc* constraints). The heuristics are not meant to provoke failures, hence, there are not suited to unsatisfiable instances. In practice, we observe on the benchmark that both heuristics roughly behave the same on unsatisfiable instances, with both a number of backtracks and a resolution time with a similar ratio. In general, we observe that the PQZ heuristic is slightly better than the corrected heuristic for the edge density $p = 1.0$, and for lower density, the corrected heuristic performs a bit better. Our assumption is that the corrected estimator is a little more able to catch the heterogeneity of the domains. As proved in the previous sections, the PQZ is not based on a bound, but our experiments show that in practice it can be considered as an estimator of the solution density. In practice, on our benchmark, this estimator seems to be useful to guide the search.

In conclusion, on all these experiments, there is no clear dominance of the PQZ estimator over our upper bound, when they are used inside a *maxSD* heuristic, in terms of solving efficiency. The number of solved instances are quite distributed around the bisector. Nevertheless, we observe that there is a clear difference between them, when they are used to find an instantiation order. In addition, we also observed that the bug in the PQZ bound indeed happens, in which case the presumed upper bound gives a value below the exact number of solutions. In the end, the benchmark is conclusive on this precise matter: the PQZ estimator and our Corrected bound are not equivalent, although using them in *maxSD* does not reveal this on the solving time.

6. Conclusion

This paper revisited the solution-counting strategies for the well-known global cardinality constraint. It first highlights that in the computation of the initial result provided by (?), a solution can be counted several times and, thus, the alleged upper bound is not an upper bound. We then provide a correct calculation of an upper bound of the number of solutions for a global cardinality constraint. Finally, we build a benchmark for the *gcc* constraint with a original method, which had not been done in the previous work, and we compare heuristics based on both quantities (the previous calculation by (?) and our upper bound) on this benchmark. We show that the two estimators lead to different choices within counting-based strategies. Finally, we solve a number of instances with both heuristics and compare the results. In practice, neither heuristic dominates the other: the counting-based heuristics used with the estimator from (?) performs slightly better on instances with a high density of edges, *i.e.* instances where

the domains are rather large and of comparable sizes. Our bound performs slightly better on instances with a lower density of edges, *i.e.* instances with more heterogeneous domains.

Though the former result is not an upper bound, it can now be seen as an estimator that can be used to guide the search, as an alternative way. That is probably why the error was difficult to spot. Our assumption is the following: when an estimator/bound used inside a counting-based heuristic, the correlation between the estimator and the actual number of solutions is much more important that the estimator accuracy. Now that we do have an upper-bound of the number of solutions, new questions can be investigated: how do different estimators compare to this upper bound? Is it possible to use our methodology (both on the calculation techniques and the benchmark generation) to introduce average-case estimators, or estimators based on probabilistic approaches? All these questions will be investigated as future works.

As further research, it would be interesting to extend our counting methods to other constraints of the same family, *i.e.* cardinality constraints such as `alldifferent`, `atmost` and `atleast`, which can all be seen as specifications of the `global_cardinality` constraint and feature the same data structure, a bipartite graph. An extension of these counting constraint is the `nvalue` constraint, whose generalized arc-consistency is NP-hard: having either a bound or an estimator for this constraint would be very valuable inside a solver. The natural question which then arises is the following: assume that all cardinality constraints come with either upper-bounds of their number of solutions, or estimators. Given a problem featuring several such constraints, how to combine these bounds to efficiently guide the search?

Finally, bound or estimations of solutions can be useful in other context than search heuristics. In particular, one of the current challenges in CP is to produce solutions which widely cover the set of solutions. On practical applications, solvers may be used by people who are not CP experts, and even not computer scientists (there are many examples in medicine (?, ?), computer graphics (?), disaster management (?), computer music (?, ?)...). In this context, users may have to rate the solutions produced by the solver or choose one. Yet, due to the systematic search process, a solver often provides solutions in a given order. In practice, two solutions in a row are often very similar in terms of values of the variables. This is unsatisfactory to the user and may drive him/her away of the technology. A better approach would be to provide solutions uniformly sampled in the solution space. Doing this implies to choose randomly the instanciation at each node of the search space, so that the random draw is uniform within *the set of solutions of the subtree of this node* (and not of the domains of the variables). Counting solutions, or estimating/bounding their number, has thus promising applications to the design of better solvers for non-expert users.